



UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

ANALIZADOR TONAL EN SOFTWARE LIBRE

Proyecto de Fin de Carrera

JUAN CARLOS DE LA FUENTE CÓRDOBA

Dirigido por: MIGUEL RODRÍGUEZ ARTACHO

Curso: CURSO 2009-2010



ANALIZADOR TONAL EN SOFTWARE LIBRE

Proyecto de Fin de Carrera de modalidad Oferta Específica

Realizado por: JUAN CARLOS DE LA FUENTE CÓRDOBA

Dirigido por: MIGUEL RODRÍGUEZ ARTACHO

Tribunal calificador:

Presidente: D./D^a

Secretario: D./D^a

Vocal: D./D^a

Fecha de lectura y defensa:

Calificación:

Resumen

En la música occidental tradicional, o *Música Tonal*, la *armonía* estudia la simultaneidad sonora de agrupaciones de notas, conocidas como *acordes*. La Música Tonal se basa en crear sonidos a partir de un conjunto de reglas consolidadas a lo largo de varios siglos de tradición musical.

En el proceso de enseñanza de la música, abordar la armonía representa un salto cualitativo en complejidad para el alumno. Mientras que en la melodía se tratan los sonidos secuenciales, en la armonía se tratan *en paralelo*. Esto implica ejercitarse la mente para superar la consabida dificultad humana de representación de la simultaneidad.

El alumno debe asimilar las reglas sintácticas de los acordes individuales y, además, aprender las reglas de enlace entre acordes sucesivos. Adquirir estas habilidades requiere repetir numerosos ejercicios. Disponer de ejercicios variados y de un tutor para corregirlos puede resultar complicado.

La pedagogía de la armonía se basa en la "realización en cuatro partes", donde los acordes están formados por 4 notas, o voces, una de ellas repetida. Aquí aparecen una complejidad añadida: las notas de un acorde se disponen en la escala de forma consecutiva, sumando una distancia de 2 notas a la anterior, pero se permiten ciertas licencias: inversiones, añadido de octavas, e incluso omisiones.

La justificación del uso del ordenador como apoyo a los ejercicios de armonía se basa en las siguientes consideraciones:

- Dificultad intrínseca de los ejercicios de armonía, y la consiguiente conveniencia de herramientas de corrección automatizada.
- Viabilidad de expresar formalmente las reglas de la armonía.
- Necesidad de seleccionar y adaptar las reglas aplicadas para adaptarse al nivel del alumno y también para manejar la subjetividad de muchas reglas.
- Escasa disponibilidad de tutores humanos cualificados

En este proyecto se propone el estudio y la realización de una aplicación informática, integrable en una plataforma de aprendizaje musical, que realice la función de reconocer y analizar los acordes introducidos por el alumno, y corregirlos en función de las reglas de la armonía tonal clásica, mostrando indicaciones sobre los errores cometidos.

LenMus es una aplicación de software libre para aprendizaje musical que permite crear libros interactivos en los que la teoría musical se intercala con ejercicios. Estos ejercicios presentan aspectos muy destacables, relacionados con su dinamismo: son configurables y aleatorios, por lo que el profesor los puede adaptar y el alumno puede disponer de una infinita variedad. Además, es destacable el seguimiento y evaluación de los resultados del alumno.

Para el analizador construido, *LenMus* ha aportado una extensa infraestructura para realizar tareas auxiliares de procesamiento musical.

Los resultados han sido enormemente satisfactorios. Se ha conseguido el objetivo principal de reconocer y analizar los acordes de una partitura, y se ha integrado el analizador con el editor de partituras de *LenMus*. Además se ha conseguido sintetizar secuencias correctas de acordes, lo que se ha utilizado para la creación de ejercicios para reconocer y completar acordes previamente generados por la aplicación.

Lista de palabras clave

Se refleja el contenido del proyecto con el fin de facilitar su almacenamiento y recuperación en un sistema de búsqueda de bibliografía.

Principales

Música, armonía, acordes, automatizado, análisis, reconocimiento, generación.

Secundarias

Musical, progresión, armónica, reglas, LenMus, enseñanza, partituras, procesamiento, ejercicios, intervalos, MIR.

Patrones

Acordes-reconocer, acordes-analizar, analizador-armonía, sistema-tonal, procesamiento-automatizado.

Summary

In classical *western music*, so-called "*tonal music*", *harmony* deals with groups of simultaneous notes: the *chords*. Creating *tonal music* requires to follow a standard set of rules, consolidated through several centuries of tradition.

Within the music teaching, learning harmony represents a quantum leap in difficulty for the student; while in *melody* the notes are processed sequentially, in *harmony* they are managed in parallel. This implies to overcome the well-known human difficulty of representing concurrency.

The student is expected to learn the syntactical rules of individual chords as well as the progression rules that control how to link successive chords. Acquiring these skills requires extensive and intensive drilling and practice. The availability of an adequate human tutor for helping in this task might be difficult or even sometimes impossible.

Although the chord creation may seem simple (basically adding an interval to the preceding note), the processing of chords becomes complicated because of the exceptions such as notes repetition, inversions, octave addition or elisions.

The intrinsic difficulty of correcting the harmony exercises is one of the main reasons that justify the use of the computer for this task. Besides, the fact that harmony rules are easily formalized, and the necessity of customizing the rules (to suit the student's level and also to manage the subjectivity of many rules), are other significant motivations.

This work proposes the study and implementation of a software application, integrated into a musical learning platform that performs the recognition and analysis the chord in a score, according to the rules of classical tonal harmony.

LenMus is an interactive, extensible and customizable music education application that allows to create *eBooks* that combine theory with exercises, which can be randomly generated and also configured, allowing the professor tailor it and the student to have an infinite variety. Tools for monitoring and evaluating the results of the student are also included.

LenMus application is completely open and free (GPL-licensed). In addition, it provides an extensive auxiliary infrastructure for processing scores and managing exercises, which allows to concentrate efforts on the task of chords processing.

The results can be considered a remarkable success. Not only has it been achieved the initial objective of recognizing and analyzing chords, but it also has been possible to build up correct sequences of chords. This functionality has been applied to exercises of several types, such as chord *identification*, chord *completion* and chord *notation*.

Keywords

The following keywords reflect the contents of this work in order to facilitate the automatic processing in an automated bibliographic search system.

Main words

Music, harmony, chords, automated, analysis, recognition, generation.

Secondary words

Musical, progression, harmonic, rules, LenMus, scores, teaching, exercises, intervals, MIR.

Patterns

Chord recognition, chord analysis, harmony analyzer, tonal music, automated music processing, music annotation, music information retrieval.

Índice de contenido

1	Planteamiento del problema.....	1
1.1	Objetivos.....	1
1.2	Motivación.....	5
1.3	Dificultades previstas.....	8
2	Contexto.....	11
2.1	Software libre.....	11
2.2	Software musical.....	15
2.3	Software educativo.....	66
2.4	Perspectiva integradora.....	81
3	Desarrollo.....	83
3.1	LenMus.....	83
3.2	Metodología de desarrollo.....	103
3.3	Problemas encontrados.....	110
4	Evaluación y pruebas.....	117
4.1	Metodología de pruebas.....	117
4.2	Pruebas unitarias.....	117
4.3	Pruebas de Integración.....	121
5	Resultados y conclusiones.....	125
5.1	Logros.....	125
5.2	Lecciones aprendidas.....	144

6 Mejoras y extensiones.....	151
6.1 Extensiones recomendables.....	151
6.2 Variantes experimentales	152
7 Bibliografía.....	155
7.1 Bibliografía citada.....	155
7.2 Bibliografía de referencia sobre enseñanza de música con ordenador.....	157
8 Anexos.....	165
8.1 Código fuente.....	165
8.2 LenMus score notation language: LDP	201
8.3 LenMus eMusicBooks' file format.....	226
8.4 La licencia libre de LenMus	246
8.5 Teoría de armonía a cuatro voces de Marcelo Gálvez.....	257
8.6 Curso de teoría de la armonía de Michel Baron.....	270
8.7 Enlaces world wide web.....	274
8.8 Glosario musical español-inglés.....	278

Lista de Figuras

Índice de ilustraciones

Ilustración 1: Variaciones de un acorde de grado I.....	9
Ilustración 2: Mapa conceptual del software libre, según GNU.....	12
Ilustración 3: Modelo de acorde en la ontología OMRAS2 Chord	37
Ilustración 4: Modelo de nota en la ontología OMRAS2 Chord	38
Ilustración 5: Modelo de intervalo en la ontología OMRAS2 Chord	38
Ilustración 6: Ejemplo de descripción de acorde en la ontología OMRAS2 Chord	39
Ilustración 7: 'Red de proceso' del TonalAnalysis de CLAM.....	50
Ilustración 8: Vista 'Key Space' de CLAM, que muestra la probabilidad de los acordes	50
Ilustración 9: El ChordData de CLAM visualiza distintas 'dimensiones' de la armonía.	51
Ilustración 10: Juego de armonización "Harmony-toy", de Music Awareness.....	55
Ilustración 11: Ejemplo del error 'cruzamiento de voces' detectado por HP3.....	58
Ilustración 12: Más ejemplos de errores de progresión armónica detectados por HP3...	59
Ilustración 13: Ejemplo de error en HP3 junto al diccionario de acordes.....	60
Ilustración 14: Conceptos implicados en el "eMusicLearning".....	66
Ilustración 15: LenMus como editor de partituras.....	88
Ilustración 16: LenMus como libro de teoría y ejercicios.....	89
Ilustración 17: Empezando a crear nueva partitura con LenMus.....	92
Ilustración 18: Asistente para crear partituras en LenMus.....	92
Ilustración 19: Aparece la partitura en el editor LenMus e introducimos las notas.....	93

Ilustración 20: Resultado del analizador aplicado a partitura LenMus.....	93
Ilustración 21: Partitura cargada en LenMus.....	94
Ilustración 22: Abriendo una partitura existente en LenMus.....	94
Ilustración 23: Resultado de aplicar el analizador a una partitura LenMus.....	95
Ilustración 24: Abriendo los libros de música disponibles en LenMus.....	96
Ilustración 25: Libro de ejercicios de armonía a cuatro voces en LenMus.....	97
Ilustración 26: Planteamiento del ejercicio de armonización de soprano	98
Ilustración 27: Ejercicio de armonización de soprano completado por el alumno.....	99
Ilustración 28: Ejercicio de armonización de soprano corregido por el analizador.....	99
Ilustración 29: Analizador aplicado a un ejercicio de voz soprano.....	100
Ilustración 30: Ejercicio de cifrado de acordes.....	101
Ilustración 31: Errores de cifrado detectados por el analizador.....	102
Ilustración 32: Ciclo del TDD (Test Driven Development).....	106
Ilustración 33: Un test de integración cargado desde un fichero LDP.....	122
Ilustración 34: Resultados de aplicar el analizador el test de integración.....	123
Ilustración 35: Jerarquía de clases de acordes.....	128

Índice de tablas

Tabla 1: Grados y Funciones Tonales.....	18
Tabla 2: Algunos destacables formatos de representación musical.....	30
Tabla 3: Programas para educación musical destacables y estrategias instructivas.....	44
Tabla 4: Teorías y modelos del aprendizaje.....	78
Tabla 5: Jerarquía de clases de acordes.....	129

"Armonía: unión y combinación de sonidos simultáneos y diferentes, pero acordes"

(D.R.A.E.)

"En realidad resulta que la melodía surge de la armonía y es precisamente de la armonía de donde debemos obtener las reglas de la melodía"

Jean-Philippe Rameau [RAM22]

"En la armonía se acumula tensión disonante que, por su inestabilidad, busca resolverse en el equilibrio consonante.

El juego permanente entre tensión y estabilidad, entre consonancia y disonancia, resulta ser el motor de la armonía musical, y en definitiva, de todo el universo."

Federico Abad [ABA07]

1 Planteamiento del problema

1.1 Objetivos

En el presente trabajo se pretende crear una aplicación informática que dé soporte al aprendizaje de la armonía, en el contexto de la enseñanza de la *música tonal occidental*. En concreto, se pretende crear un módulo capaz de analizar la armonía de una partitura, aplicando las reglas clásicas.

El analizador debe ser capaz de reconocer los acordes y detectar su tipo. Se espera que también pueda realizar un análisis básico de la *progresión armónica*, aplicando algunas de las reglas más reconocidas al respecto.

Adicionalmente se pretende integrar este analizador en un entorno de educación musical para aplicarlo a la creación y corrección de ejercicios de *armonía a cuatro voces*.

Se pretende poder guardar tanto las partituras como los ejercicios en un formato que resulte práctico tanto para el humano como para ordenador.

Dada la previsible dificultad del objetivo propuesto, no se plantean objetivos teóricos, sino tan solo prácticos. Además, se pretende maximizar la utilidad del esfuerzo realizado, por lo que se buscará obtener algoritmos y estructuras reutilizables. También se busca conseguir que el trabajo pueda tener continuidad, por lo que se ha de integrar en un entorno de *software libre*.

La plataforma elegida para desarrollar el analizador es el programa de aprendizaje musical LenMus, ya que por una parte suministra una infraestructura básica de estructuras y algoritmos ya existentes sobre los que construir con menor esfuerzo nuestra aplicación, y por otra parte también dispone de un mecanismo que facilita la creación de ejercicios interactivos, y que puede servir para los ejercicios de armonía a cuatro voces.

1.1.1 Requisitos *no funcionales*

Para cumplir con los objetivos propuestos, se plantea las siguientes restricciones o *requisitos no funcionales*:

- ✓ Que el material producido sea completamente **libre**.
- ✓ Que se utilice un entorno o infraestructura ya existente, y asimismo libre, para ayudar en tareas auxiliares tales como: interfaz de usuario, representación visual, serialización y audición de partituras, uso de notaciones y transformación de formatos, procesamiento básico de música (conversiones de escalas, cálculo de intervalos, operaciones con notas), etc.
- ✓ Que se utilice un lenguaje orientado a objetos para facilitar la reutilización del diseño.

Además, existen requisitos *deseables* para el entorno en el que debe integrarse el analizador. No son, pues, requisitos para el propio desarrollo a realizar.

- ✓ Interoperatividad: compatibilidad con una o varias varias *notaciones musicales* ampliamente difundidas y posibilidad de importar y exportar partituras.
- ✓ Editor de partituras con generación y visualización realista.
- ✓ Reproducción auditiva de las partituras mediante MIDI.
- ✓ Integración sin esfuerzo de la herramienta de análisis de armonía con el resto de la plataforma.
- ✓ Entorno didáctico integrado, que contemple tanto ejercicios como teoría.
- ✓ Extensibilidad y flexibilidad: posibilidad de incorporar y adaptar ejercicios.
- ✓ Seguimiento de los resultados del alumno.

1.1.2 Requisitos funcionales

Debido a que resulta difícil estimar la dificultad del trabajo a realizar y, por tanto, hasta dónde se podrá llegar, los requisitos funcionales se plantean como *incrementos* de funcionalidad ordenados por prioridad. De esta forma garantizamos que se llegará a resultados prácticos siempre, y se conseguirán los objetivos principales.

Por tanto los requisitos se concretan en los siguientes incrementos de funcionalidad:

Fase 1: *Análisis de una partitura LenMus*

1. **Reconocer acordes:** detectar los grupos de 3 o más notas *simultáneas*.
2. Calcular **tipo:** distinguir si los grupos de notas detectados forman acordes *válidos*.
3. **Indicaciones** al usuario: mostrar resultados del procesamiento e indicaciones para el usuario, de forma gráfica, en la partitura visible.
4. Crear una **clase** para gestionar acordes y almacenar la secuencia de la partitura en una lista los acordes.
5. Analizar la secuencia de acordes aplicando al menos dos **reglas de enlace** de acordes importantes:
 - Evitar *movimientos paralelos*.
 - Evitar *cruces* entre voces.
6. Crear un **mecanismo de reglas genérico** que permita fácilmente activar/desactivar y añadir/quitar reglas.
7. Añadir nuevas reglas de enlace:
 - Evitar *llegar a quinta* u octava por movimiento directo.
 - Evitar intervalos mayores de octava.

Fase 2: *Generación de acordes y ejercicios*

8. Conseguir generar **acordes individuales** correctos
9. Conseguir generar **secuencias de acordes** correctas
10. Crear **ejercicios** consistentes en mostrar secuencias de acordes con notas ocultas y dar indicaciones al alumno para completarlas. Verificar la corrección de los datos introducidos y mostrar los mensajes correspondientes.
11. Implementar un mecanismo que permita generar acordes a partir de notación de **bajo cifrado**, descrita en una cadena de texto. También implementar el camino contrario: obtener el bajo cifrado de cualquier acorde.
12. Crear un ejercicio que consista en generar acordes, pedir al alumno que elija el cifrado correspondiente, y verificarlo.

Fase 3: **Refactorización:** favorecer la interoperatividad y reutilización

13. Adaptar las clases creadas a las necesidades del entorno LenMus de forma que puedan resultar útiles para desarrollos futuros en este entorno. A la vez, refinar estructuras y algoritmos para que se puedan reutilizar en cualquier entorno de procesamiento musical.
14. Permitir exportar e importar partituras y ejercicios a una notación musical fácil de entender y usar, y que además sea portable a otras aplicaciones musicales.

1.1.3 Otros objetivos secundarios

Al margen de los objetivos concretos del trabajo, para los que se espera un resultado práctico y visible, existen otros objetivos secundarios relacionados con el conocimiento de los temas relacionados.

Estos objetivos se realizarán en la medida de lo posible, y estarán supeditados a los objetivos de funcionalidad.

1. Adquirir conocimiento en las materias relacionadas:
 - Procesamiento de partituras.
 - Análisis automatizado de armonía.
 - Representación de acordes.
2. Compartir el conocimiento adquirido, plasmándolo en forma de consejos y directrices para ayudar al desarrollo de aplicaciones de procesamiento de la música en un contexto educativo.
3. Estudiar y evaluar programas de utilidad para la enseñanza de música, y más concretamente de la armonía, con énfasis especial en aquellos de tipo software libre, debido a su mejor adecuación al entorno educativo.

1.2 Motivación

La motivación principal de este trabajo surge de la constatación de que, dentro del aprendizaje musical, los ejercicios de armonía constituyen un campo muy adecuado para ser *computerizado* y gestionado mediante el uso de la informática, ya que estos ejercicios presentan características particulares como las que se mencionan a continuación.

Dominio modelable

El conocimiento del dominio de la Música Tonal, o Sistema Tonal, es en gran medida explícito y representable mediante reglas precisas, tradicionalmente aceptadas y reconocidas. Existe un *corpus* de conocimiento consolidado, bien definido y perfilado. En particular, las reglas de la armonía se han estudiado, discutido y, en gran medida, acordado desde hace ya siglos, siendo la referencia principal el libro de Rameau "*Tratado de Armonía reducida a sus principios naturales*" [RAM22].

Personalización

Es cierto que solo una parte de las reglas de armonía están perfectamente definidas y son unánimemente aceptadas sin discusión. Muchas otras son menos objetivas; por ejemplo, algunas son aceptadas por unos expertos y rechazadas por otros, en otras se discute sobre las excepciones aceptables, y en otras se debate sobre ciertos valores o variantes.

La subjetividad de las reglas se puede manejar programáticamente con relativa facilidad mediante el uso de parámetros en los ejercicios u opciones en la configuración.

Además, hay que tener en cuenta el hecho de la distinta relevancia entre las reglas. Mientras que algunas son fundamentales para el aprendizaje de la armonía más básica, hay otras que son secundarias en el sentido de que pertenecen a ciertos enfoques, visiones o especializaciones de la armonía, y, por tanto, solo interesarán ejercitarse en ciertos casos.

Por tanto, dependiendo del nivel del alumno, del plan pedagógico o de otros factores, puede interesar enfocar el adiestramiento a ciertas reglas. El ordenador permite activar y desactivar con facilidad las reglas que se desea ejercitarse.

Mediante la implementación de relativamente sencillos mecanismos de configuración, se consigue personalizar y adaptar la enseñanza a las necesidades, y los planteamientos del tutor y del alumno.

Repetición

El adiestramiento en el reconocimiento y la composición de acordes se basa en la repetición de ejercicios básicos, más que en la memorización teórica. Mediante el empleo del ordenador para generar y corregir estos ejercicios, se consigue que el alumno disponga de una herramienta con capacidad ilimitada de creación de casos distintos sobre de los ejercicios fundamentales.

En general, las tareas repetitivas y bien definidas son ideales para ser realizadas por ordenador, mientras que, por el contrario, a los humanos les resultan monótonas y desmotivantes.

Complejidad de supervisión para el tutor humano

El conjunto de reglas de armonía es suficientemente amplio y complejo como para su aplicación por parte de humanos resulte propensa a errores y omisiones, incluso para los expertos. Las conocidas ventajas inherentes al uso del ordenador son que si las tareas están definidas y programadas correctamente, él realizará su labor de forma infalible e incansable, aplicando todas las reglas y aplicándolas correctamente.

Disponibilidad

Ya se ha comentado la dificultad intrínseca que para un humano representa la tarea de corregir ejercicios de armonía. También se ha mencionado que el aprendizaje de la armonía se basa fuertemente en la repetición de ejercicios básicos. Por tanto, la disponibilidad permanente de un tutor supone una gran ventaja en el aprendizaje. El hecho de no requerir más que un ordenador, y poder ejercitarse el adiestramiento en cualquier lugar y a cualquier hora, representa un factor enormemente valioso.

Integración e interoperabilidad

El estudio de la armonía representa solo una pequeña parte dentro de la enseñanza del Lenguaje Musical y está ligado a otros, como pueden ser la melodía o el ritmo.

Este tutor virtual de armonía se puede programar como un componente integrable en un curso a mayor escala. Este componente puede, por tanto, formar parte de un plan pedagógico interactivo global.

Si el curso de armonía se plantea como un módulo informatizado, se abrirán muchas posibilidades para complementarlo o integrarlo con otros materiales didácticos. Por ejemplo, gracias a la posibilidad de acceder a partituras de piezas musicales en formato interoperable como

MusicXML, el profesor puede plantear ejercicios en los que el alumno debe crear armonización de melodías conocidas; para ello solo tiene que eliminar de la partitura los acordes, dejando la melodía.

Rapidez y realimentación

La evidente ventaja de la velocidad a la hora de corregir los ejercicios, no solo representa un ahorro de tiempo para el profesor y el alumno, sino que también mejora la fluidez de la realimentación de información de los resultados. Cuanto antes se dé cuenta el alumno de sus errores, mejor asimilará su corrección. Esta rápida realimentación o *feedback*, también permitirá al profesor mejorar la adaptación del curso al perfil del alumno. Como lo expresa Horacio Lapidus en [LAP01], páginas. 9-10:

"En definitiva, la Tecnología de la Información provee recursos óptimos para el desarrollo de materiales educativos dotados de niveles de inteligencia artificial, adecuados para el aprendizaje de la armonización de melodías, mediante el apoyo de una combinación de diversas formas de feedback adaptables a los distintos momentos y particularidades del proceso cognitivo de cada estudiante."

Audición

Un módulo de pedagogía de armonía informatizado permite una relativamente sencilla integración con otro módulo que ejecute musicalmente las notas introducidas por el alumno en el ejercicio. El hecho de poder escuchar lo que ha escrito, facilita la asimilación de los ejercicios por parte del alumno, además de otras ventajas, como pueden ser aumentar su motivación o mejorar su capacidad de percepción auditiva musical. Jesús Tejada lo explica así en [TEJ99]:

"Por otra parte, didácticamente era muy conveniente poner a disposición del alumnado una herramienta que le permitiera oír sus propios arreglos y composiciones con el fin de desarrollar el oído melódico y armónico, así como la discriminación tímbrica. Pero, ¿cómo podrían oír sus materiales, si la mayoría de los alumnos que ingresan en estos estudios carecen de las suficientes destrezas musicales e instrumentales para poder interpretarlos en un instrumento armónico? "

1.3 Dificultades previstas

Si se consulta las experiencias de los investigadores en el campos del análisis automatizado de armonía musical (Bryan Pardo, Roger G. Dannenberg, David Temperley, Christopher Harte entre otros), se aprecia claramente que los acordes no resultan nada fáciles de procesar debido a sus múltiples facetas.

Para profundizar en la problemática del análisis de acordes se recomienda el trabajo de Bryan Pardo "*Automated partitioning of tonal music*" [PAR99].

Las reglas de formación de acordes no son sencillas de implementar. A continuación las recordamos para explicarlo.

La regla básica de formación de acordes es engañosamente simple:

Los acordes se forman con notas simultáneas, separadas cada una de la siguiente por una distancia de 2 notas. Un acorde correcto requiere al menos tres notas.

Pero sobre esta sencilla regla se añaden ciertas licencias o excepciones que complican bastante su implementación:

- Se permiten *inversiones*, que consisten en subir la nota más baja (el *bajo*) una o más *octavas*.
- Equivalencia de octavas: a cualquier nota distinta del bajo se le permite *subir* cualquier número de octavas.
- Se permite repetir ilimitadamente cualquiera de las notas, y en cualquier octava, siempre que se mantenga el bajo.
- Además pueden darse circunstancias extraordinarias: notas que faltan (*elididas*) y notas que sobran (notas *ajenas*)

Por tanto un mismo acorde puede encontrarse en estados diferentes, según las inversiones, y a su vez, en cada estado puede representarse con distintas combinaciones de notas.

Para complicar más el problema, hay que considerar que las notas que forman un mismo acorde variarán ligeramente según la *tonalidad* (el grado de la escala del que se parte). Estas variaciones se manifiestan con signos unidos a algunas notas, llamados *alteraciones*, que básicamente son semitonos añadidos o restados a la nota asociada.

Un mismo acorde mantiene siempre las distancias (intervalos) entre sus notas, sin embargo las notas y las alteraciones que lo componen pueden variar según las 24 tonalidades.

Analizador tonal

Variaciones de un mismo acorde

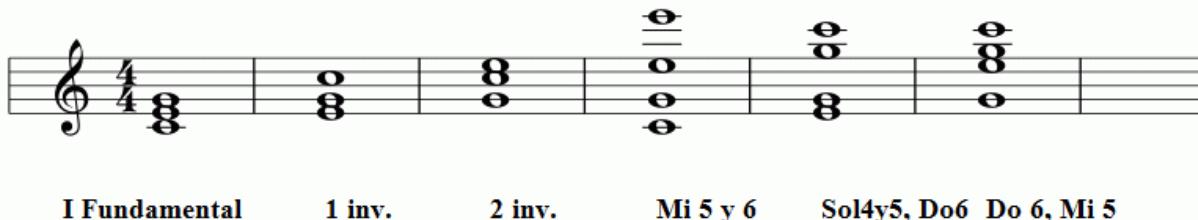


Ilustración 1: Variaciones de un acorde de grado I

La figura ilustra el problema de los distintos estados y representaciones de un mismo acorde.

Existen, además otros problemas más sutiles como son algunas ambigüedades del Lenguaje Musical, por ejemplo respecto al concepto de acorde, hay dos interpretaciones:

- Un acorde se puede entender en sentido *estricto*, como grupo de notas que cumplen las reglas de formación de acordes, o bien en sentido *relajado*, como cualquier grupo de notas simultáneas
- Aunque la mayoría de las reglas de formación son objetivas, hay algunas que permite cierta subjetividad, como el caso de las notas *elididas*, que son notas que faltan. Estas reglas subjetivas plantean varios problemas: por un lado, es preciso realizar un esfuerzo para conseguir aclarar su alcance de aplicación (cuándo, en qué medida y cómo se van a aplicar). No hay unanimidad sobre cuándo se puede omitir una nota de un acorde, y esto debe reflejarse en un mecanismo flexible de aplicación de esta regla. Por otro lado, las reglas subjetivas suelen resultar más complicadas de implementar. En el caso particular de las notas elididas, es fácil de ver que el hecho de permitirlas complica enormemente la detección de acordes válidos, pues requiere considerar los casos posibles de notas omitidas.

2 Contexto

2.1 Software libre

2.1.1 El concepto del software libre

Software libre es aquel que una vez adquirido, ya sea pagando o gratuitamente, puede ser usado con *completa libertad*, es decir que puede ser usado, copiado, estudiado, cambiado y redistribuido libremente.

El software libre requiere disponer del código fuente, porque solo así es posible ejercer completamente la libertad sobre él:

- Libertad de usar el programa, con cualquier propósito.
- Libertad de estudiar cómo funciona el programa y modificarlo, adaptándolo a tus necesidades.
- Libertad de distribuir copias del programa, con lo cual puedes ayudar a tu próximo.
- Libertad de mejorar el programa y hacer públicas esas mejoras a los demás, de modo que toda la comunidad se beneficie.

Es importante destacar la diferencia del software libre con respecto a otros tipos de software con los que a veces se le confunde. El software gratuito no necesariamente permite disponer del código fuente. El software *abierto* (*open source*) puede limitar la distribución de modificaciones.

Mientras que el software *libre* se basa en principios *éticos*, y principalmente en la libertad, el software *abierto* se basa en criterios de ventajas *técnicas* e incluso *económicas*.

2.1.2 Los valores del software libre

Según argumenta la principal asociación defensora del software libre, la *Free Software Foundation*, el software es *conocimiento* y por ello debe poderse difundir sin restricciones. Su ocultación es una actitud *antisocial* y la posibilidad de modificar programas es una forma de *libertad de expresión*.

El software abierto se basa en una filosofía mucho más pragmática, enfocada a poder obtener también beneficios particulares e individuales.

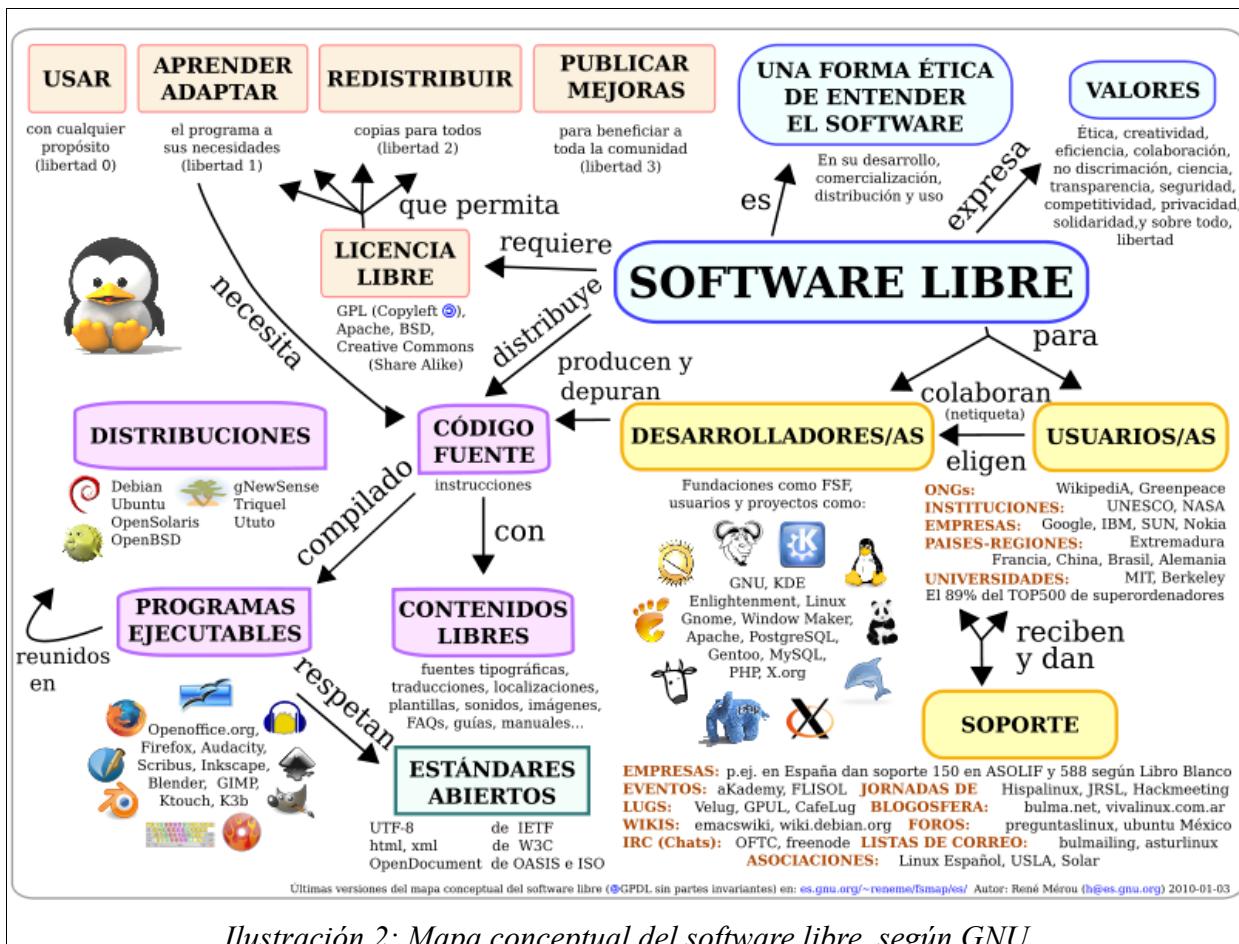


Ilustración 2: Mapa conceptual del software libre, según GNU

2.1.3 Los motivos para el uso del software libre en la educación

El software libre está muy ligado al concepto del conocimiento. La idea es que "el saber" es en sí un bien común y por tanto debe compartirse. En consecuencia, el software libre encuentra su ámbito natural de aplicación dentro del mundo centrado en torno al conocimiento, que es precisamente el mundo de la educación y el aprendizaje.

El software libre tiene todas las ventajas del software abierto, que en mucho casos hacen que tenga mejor calidad que el software privativo o cerrado, y además añade positivos y motivadores sentimientos del altruismo, el beneficio mutuo, y la libertad.

En definitiva el software libre se basa en valores éticos esenciales, que son los que se pretenden transmitir mediante la educación y por tanto es la herramienta más *naturalmente* adecuada para ejercer la educación.

Desde un punto de vista más concreto y práctico, el software libre permite compartir recursos educativos, estándares, herramientas, y en definitiva compartir el conocimiento, lo que crea las mejores condiciones ambientales para que el conocimiento crezca.

Jesús González Barahona expone algunas ventajas en su estudio "*El Software libre en la educación*" (<http://www.miescuelayelmundo.org/spip.php?article305>) [GON04].

- Puede adaptarse a las necesidades docentes concretas.
- El alumno puede reproducir el entorno de prácticas donde quiera.
- Pueden usarse marginalmente muchas herramientas (no hay problemas de coste).
- Todo el material usado puede ponerse a disposición de otros docentes.
- Permite altísimos niveles de adaptación.
- Es neutro frente a fabricantes.

En el *wikilibro "El software libre en la educación"* se defiende el uso de software libre en el contexto educativo. URL: http://es.wikibooks.org/wiki/El_software_libre_en_la_educaci%C3%B3n/Teor%C3%ADA

"... por eso, la primer razón por la cual una universidad, como cualquier institución educativa publica, debe utilizar software libre es justamente que para eso mismo ha surgido: para favorecer la divulgación del ya vasto conocimiento humano y esto es imposible de lograr utilizando sistemas que ponen todo tipo de trabas a la libertad de copiar y utilizar el conocimiento.

Además el uso de software libre en la educación tiene una gran cantidad de beneficios sobre el uso de el software privativo entre las cuales se encuentran:

1. *Crea profesionales independientes de un determinado entorno de software.*
2. *Reduce costos.*
3. *Permite que los alumnos puedan usar el mismo software con el que se les enseña.*
4. *Ofrece control sobre el software.*
5. *Es el futuro de la informática, a lo que debería apuntar una universidad del siglo XXI*
6. *Es una buena herramienta de aprendizaje en el área de informática.*

La pregunta no debe ser 'por qué usar software libre en la educación' sino más bien al contrario: 'por qué usar software privativo' ".

2.1.4 El software libre en la educación en España

El uso del software libre en la educación está promovido por varias leyes en España. Por ejemplo la Orden EDU/2341/2009, de 27 de agosto, por la que se crea el *Centro Nacional de Desarrollo Curricular en Sistemas no Propietarios*, y tiene como finalidad el diseño, el desarrollo y la promoción de contenidos educativos digitales para colectivos educativos específicos, en el ámbito de las Tecnologías de la Información y la Comunicación, que se centra en promocionar y aplicar estrategias dirigidas a poner a disposición de los centros escolares recursos y contenidos digitales de calidad, desarrollados en software libre.

Diversas instituciones educativas regionales promueven el uso del software libre, entre las que destaca el CENATIC (*Centro Nacional de Referencia de Aplicación de las Tecnologías de la Información y la Comunicación (TIC) basadas en fuentes abiertas*, <http://www.cenatic.es/>) que es una Fundación Pública Estatal, promovida por el Ministerio de Industria, Turismo y Comercio (a través de la Secretaría de Telecomunicaciones y para la Sociedad de la Información y la entidad pública Red.es) y la Junta de Extremadura, que además cuenta en su Patronato con varias comunidades autónomas y empresas.

El Decreto 72/2003 de la Junta de Andalucía, de Medidas de Impulso de la Sociedad del conocimiento contempla el software libre como un instrumento facilitador del acceso a la Sociedad del Conocimiento. En el artículo 33 establece entre sus líneas maestras que "*El software empleado en Educación será software libre*".

También algunas universidades promueven activamente en la implantación del software libre en el entorno educativo. Por ejemplo la Universidad Rey Juan Carlos de Madrid ha compilado varios estudios sobre el tema en el documento *Sobre software libre* de Vicente Matellán y otros (<http://gsyc.es/~grex/sobre-libre/libro-libre.pdf>).

Se puede obtener documentación abundante sobre el software libre en <http://www.softwarelibre.org.bo/wiki/info:biblio>

2.2 Software musical

2.2.1 Conceptos básicos de *Música Tonal*

Para comprender la tarea que se pretende realizar en este trabajo, se precisan ciertos conocimiento básicos del *Lenguaje Musical*, por ello se aconseja repasar previamente algún buen libro de teoría de armonía que también introduzca nociones generales de Música Tonal, como el libro *Curso completo de Teoría de la Música*, de Vanesa Cordantonopulos [COR02] (www.lapalanca.com), o el *Armonía Práctica* de Miguel Ángel Mateu [MAT06]. Se precisa estar familiarizado con conceptos fundamentales del Lenguaje Musical como *escalas*, *modos*, *tonalidad*, *grados*, *intervalos*, *acordes* y *propiedades de los acordes*.

No obstante, a continuación se explican someramente algunos de estos conceptos.

Altura de una nota: es la *frecuencia* del sonido asociado.

Escala musical es un rango de *tonos* (sonidos) que se incrementan gradualmente en altura.

La escala **cromática** tiene 12 pasos iguales llamados **semitonos**. Transformando esta escala mediante agrupaciones, se obtiene la escala **diatónica**, la cual tiene pasos (**grados**) de dos alturas distintas: de 1 o de 2 semitonos.

La escala diatónica se puede *sobreponer* sobre la escala cromática comenzando por cualquiera de los 12 pasos de ésta, lo que da origen a otras tantas **tonalidades** distintas.

La *tonalidad* define, por tanto, la nota de referencia en la escala: la nota **tónica**. Dentro de esta escala, a cada grado se le asocia un rol distinto, y dentro de la jerarquía de roles que se establece con los grados, el rol de *tónica* es, con diferencia, el más importante.

La *Música Tonal* se refiere a una forma de componer música basada en el respeto a ciertas normas establecidas respecto a los roles de los grados, como, por ejemplo, que la composición debe finalizar con la nota *tónica*.

Por extensión, el concepto de **tonalidad** se asimila con la jerarquía de roles o grados basados en la nota *tónica*. En este sentido puede también denominarse *tonalismo*.

Una vez fijada la nota *tónica* sobre la escala cromática, para obtener una escala cromática concreta, debe elegirse uno de los dos tipos posibles de agrupamiento de notas, que son los *modos*.

El **modo** define la distribución de alturas para los grados de la escala *diatónica*. Hay dos modos, el *mayor* y el *menor*, y ambos tienen *7 grados*, que son las notas DO, RE, MI, FA SOL, LA, SI. Sin embargo, la diferencia de altura entre estos grados (es decir, el *intervalo*) será distinto en cada uno de los modos.

En cada modo, el *intervalo* entre dos grados consecutivos puede ser de uno o de dos semitonos cromáticos. Si denominamos *S* al intervalo de un semitono y *T* al de dos semitonos (o sea de un tono), las distribuciones de alturas, a partir de la nota *tónica*, para cada uno de los modos se expresan así:

Modo Mayor : T, T, S, T, T, T, S

Modo Menor : T, S, T, T, S, T, T

Al observar ambos patrones de distribución, se constata que uno se puede obtener a partir del otro *desplazando 5 elementos a la izquierda*.

Con la información anterior se puede concluir que existen 24 *tonalidades* distintas para la escala *diatónica*: 12 en modo *mayor* (uno por cada paso de la escala *cromática*) y otros tantos en modo *menor*.

Los *intervalos* se miden por grados de la escala diatónica, y se expresan con números *ordinales*, y además se añade siempre una unidad, con lo que, por ejemplo, un intervalo de 2 grados se denomina de *tercera* (2+1), pero si sumamos dos intervalos de *tercera* (2+2), obtenemos un intervalo de *quinta* (2+2+1).

La escala diatónica es *cíclica*, de forma que la nota siguiente a la última (SI) es de nuevo la primera, DO. Una *octava* es un ciclo de la escala diatónica.

2.2.1.1 Consideraciones sobre el procesamiento de la Música Tonal

En este punto conviene recapitular sobre los conceptos anteriores, pero aplicando ya una perspectiva *informática*. En este sentido, si se pretende procesar música *programáticamente*, conviene ser consciente de ciertos hechos objetivos:

1. Al procesar una partitura, gran parte del esfuerzo se dedica a conocer con exactitud en qué instante suenan las notas y con qué *altura* lo hacen. El Lenguaje Musical, por las características comentadas de ambigüedad y dependencia del contexto, **no** facilita la tarea

de averiguar el *tiempo* en que suenan las notas y la *frecuencia* de las mismas. Por lo tanto, al procesar una partitura se debe **buscar una forma de representar la partitura en el lenguaje destino** (el lenguaje de programación usado) que sí permita conocer con facilidad esos dos atributos esenciales de la música: tiempo y altura.

2. Es muy importante comprender y asimilar la relación entre las dos escalas. La escala *cromática* debe verse como la que tiene la información *verdadera* respecto a la *altura* de los sonidos, mientras que la *diatónica* es una *distorsión* pensada para los *compositores*, pero no para los "*informáticos*". Por tanto, conviene **evitar tratar con la escala diatónica** en la medida de lo posible, ya que, para conocer la altura real, nos exige siempre realizar cálculos que transformen las notas a la escala *cromática*.

2.2.1.2 Conceptos básicos de Armonía Tonal

Las *armonía* se basa en los conceptos de *consonancia* y *disonancia*. El primero se asocia a la sensación de *equilibrio* y relajación, mientras que el segundo se asocia a *tensión* e inestabilidad.

Como se ha comentado, *intervalo* es la diferencia de altura entre dos notas musicales. Si las notas son simultáneas, se denomina intervalo *armónico*, y si son sucesivos, se denomina intervalo *melódico*.

Los *intervalos* tienen asociada la importante propiedad del *tipo de consonancia*, y esta puede ser *consonante* o bien *disonante*. Las notas individuales no tienen por sí mismas *tipo de consonancia*, y por ello se puede decir que los **intervalos son los elementos esenciales sobre los que se construye la armonía**.

Un *acorde* es una combinación de dos o más *intervalos armónicos* que suenan simultáneamente.

Los *intervalos* de los acordes están siempre referidos a la nota más baja.

En los *acordes* se ignoran las *octavas*; es decir, se trabaja con *intervalos entre grados*, y no entre *notas*. A las *notas-independientes-de-octava* que determinan los *intervalos* del acorde, y que por tanto definen el propio acorde, se les suele denominar *factores*.

Un *acorde* está en *estado fundamental* si está formado exclusivamente por *intervalos de tercera* (o sea, de 2 *grados* de diferencia) sucesivamente añadidos a un grado concreto.

Un acorde de **tríada** es el formado por un intervalo de *tercera* y uno de *quinta*. Otra forma de verlo es la adición de dos intervalos de tercera.

Al *grado* sobre el cual se construye un acorde en *estado fundamental* se le denomina **grado del acorde**.

Por tanto, en el *Sistema Tonal* nos encontramos con dos tipos de *grados*: de la escala *diatónica* y de los acordes. Como ya se mencionó, cada grado de la escala se asocia a un rol concreto o *función tonal* en el Sistema Tonal. Sucede que esta asociación de grado se mantiene en los acordes, de forma que podemos establecer la *función tonal* de cada grado (sea nota o acorde) según la siguiente tabla:

Grado	Función Tonal
I	<i>Tónica</i>
II	<i>Supertónica</i>
III	<i>Mediante</i>
IV	<i>Subdominante</i>
V	<i>Dominante</i>
VI	<i>Submediante</i>
VII	<i>Sensible o Subtónica</i>

Tabla 1: Grados y Funciones Tonales

El *intervalo complementario* (o *invertido*) de otro es la *diferencia* con respecto a una *octava*.

Aplicar una **inversión** a un acorde implica reemplazar el primer intervalo por su *complementario*. Desde el punto de vista de las notas, una *inversión* se interpreta como *subir una octava* a la nota más baja.

El *estado* del acorde es *fundamental* si no tiene *inversiones* y, en caso contrario, es el ordinal correspondiente al número de inversiones.

Los acordes conservan su *función tonal* independientemente del *estado*.

2.2.2 Complejidad de modelar el Lenguaje Musical

2.2.2.1 El problema de la *informalidad*

Es un hecho evidente que el Lenguaje Musical dista mucho de ser un lenguaje formal. Quizás sí lo sea en los niveles de *símbolos* y de *sintaxis*, pero no lo es en la *semántica*, ya que es fácil encontrar situaciones en las que la interpretación de los símbolos es ambigua o imprecisa. En general, el Lenguaje Musical tiene la negativa propiedad de ser *dependiente del contexto*.

Un estudio riguroso sobre las propiedades formales del Lenguaje Musical lo tenemos en el trabajo de Bryan Jurish [JUR04] , donde se explica que cuanto más *natural*, es decir, más *informal*, es un lenguaje, mayor capacidad generativa tiene, pero también es mayor la dificultad para su *modelado* y, por tanto, su procesamiento automatizado. Y en este sentido, el *Lenguaje Musical* tiene mucho en común con el *lenguaje natural*, como dice Bryan Jurish:

"Empirical arguments were presented that musical languages exhibit the characteristic properties of the mildly context-sensitive languages, to which natural (spoken) languages are also assumed to belong."

2.2.2.2 El problema de la complejidad

Es así: el Lenguaje Musical es complicado. Es el fruto de una larga evolución, pero los criterios a la hora de definirlo o rehacerlo no han sido generalmente los de optimizarlo, estructurararlo o simplificarlo, sino que más bien ha ido cambiando en función de costumbres o modas.

Probablemente sería posible diseñar un lenguaje con la misma capacidad expresiva pero con mucha menor complejidad. Cuando ya se ha familiarizado con ese lenguaje, un humano puede interpretar sus fórmulas (las *partituras*) con sorprendente facilidad, pero diseñar un procesador de partituras no es una tarea sencilla ni evidente.

Baste como ejemplo resaltar el hecho de que, para interpretar correctamente cualquier parte de una pieza musical, es preciso comenzar a procesar desde el principio mismo. Además, los elementos (*símbolos*) se disponen, no en una, sino en *dos* dimensiones (*tiempo* y *altura*), organizados en *frases* (pentagramas) que discurren paralelas en el eje del tiempo, pero con ciertas

irregularidades que hacen que una nota situada antes en un pentagrama pueda de hecho sonar después.

2.2.2.3 El problema de la subjetividad de la armonía

Dentro de la música, la armonía presenta problemas específicos que dificultan su tratamiento a la hora de intentar modelar sus conceptos y reglas. En especial hay un problema en la misma raíz del concepto armonía.

Para entenderlo, podemos recurrir a la aproximación que nos proporciona Federico Abad [ABA07]:

*"La mayor o menor coincidencia de los ciclos vibratorios de dos sonidos simultáneos da lugar a los conceptos de consonancia y disonancia. Son intervalos armónicamente consonantes los que percibimos juntamente como **agradables** o estables. Son disonantes los que percibimos con una sensación de choque o tensión."*

Y continúa citando ejemplos de intervalos sobre cuya consonancia o disonancia no hay o no ha habido siempre unanimidad, como son los intervalos de 3^a, 6^a y 4^a.

La subjetividad en la armonía se extiende también a la combinación secuencial de acordes, es decir, a las normas de las cadencias (dos acordes consecutivos) y de la *progresión armónica* (secuencia de acordes consecutivos).

Esta subjetividad la expresa Miguel A. Mateu en [MAT06] de esta forma:

"La capacidad intelectiva y cultural para asimilar un intervalo (expresado de otro modo: la educación armónica) hace que lo que para algunos son disonancias, otros lo entienden como consonancias."

2.2.3 Procesamiento de la música mediante ordenador

El campo del procesamiento de la música mediante ordenador, que bien podría denominarse "*eMusic*", es enormemente extenso. Sin embargo, una buena parte de los esfuerzos se dedican al procesamiento digitalizado de la señal de audio en general, usando formatos de bajo nivel y amplio espectro, como el MP3.

Si lo que se pretende es procesar piezas musicales, es preferible acotar la envergadura del problema y comenzar a trabajar sobre una notación a nivel del *Lenguaje Musical*, evitando la tendencia a usar herramientas y formatos demasiado genéricos, y a un nivel alejado del que se pretende tratar.

Es importante darse cuenta de que si se va a procesar música, vale la pena dedicar el tiempo preciso a buscar el entorno adecuado, un entorno que permita trabajar directamente sobre partituras, sea cual sea el formato de estas. La existencia de abundantísimo material especializado en procesamiento de audio y sonido en general resulta tentador, ya que suele estar soportado por una gran comunidad de desarrolladores, usuarios y, por ello, suele disponer de una impresionante y variada funcionalidad. Además, en la documentación a menudo aparecen destacadas capacidades cercanas al nivel del Lenguaje Musical. Por muy atractivo que todo esto pueda parecer, se recomienda dejar de lado las librerías y utilidades de bajo nivel y trabajar con aquellas específicamente pensadas para procesar música como tal, no sonidos en general.

Más adelante, en el apartado de *software musical*, se darán detalles de algunas librerías y formatos comunes para el procesamiento de música.

Hay mucho material disponible y por ello se debe valorar con cuidado cuál es el más adecuado para las necesidades de cada caso.

Un buen lugar para comenzar a informarse sobre el vastísimo mundo del "*eMusic*" puede ser el portal [International Computer Music Association](http://www.computermusic.org/) (<http://www.computermusic.org/>), que organiza regularmente congresos internacionales, como la [International Computer Music Conference](http://www.icmc2010.org/) (<http://www.icmc2010.org/>), y publica gran parte de los trabajos presentados.

Las áreas de interés de la asociación son muy variadas, como por ejemplo la Inteligencia Artificial, el procesamiento de señales, el arte, las interfaces gestuales o 'hápticas', las notaciones, la sintetización de sonidos y un muy largo etcétera. En la Wikipedia realizan un interesante resumen en http://en.wikipedia.org/wiki/International_Computer_Music_Conference:

Otra consideración sobre la *eMusic* es que los campos sobre los que se trabaja bien podrían agruparse en función de la calidad *sintética/analítica*, aspecto sobre el que se incide continuación.

2.2.4 La síntesis musical o composición automatizada

Las actividades de tipo *sintético* agrupan principalmente trabajos relacionados con intentos de *componer* automáticamente mediante ordenador. En gran medida consisten en intentos de aplicar la *Inteligencia Artificial* a la *creatividad musical*.

La composición automatizada de *melodías* choca con el problema de la enorme *subjetividad* de las reglas y de los resultados. Sin embargo, la *armonización* automatizada sobre un melodía dada,

es un campo mucho más propicio a obtener resultados concretos y de una calidad objetivamente medible, gracias a que las reglas de la armonía están en general bastante bien perfiladas y objetivadas. Esto hace que en este campo de la *armonización* automatizada se desarrollem numerosos y variados estudios, con notables resultados.

Según el investigador François Pachet en su estudio "*Musical Harmonization with Constraints: A Survey*" de 2001 [PACH01], el problema de generar **armonías** ya está básicamente "**resuelto**", si bien no se puede decir lo mismo para el problema de la generación de **melodías**:

"The technical problem of four-voice harmonization may now be considered as solved, using constraint satisfaction techniques based on arc-consistency, and an adequate structuring of the problem to handle chord variables properly. This result comes after several years of trials and errors, starting from brute force approaches (e.g. Schottstaedt), to proprietary constraint languages (Ebcioğlu), to arc-consistency techniques (Ovans), augmented with adequate problem structuration (Pachet & Roy). However, what remains unsolved is the problem of producing musically nice or interesting melodies. There are several open issues concerning what makes melodies interesting"

En el anexo de bibliografía se pueden encontrar numerosas referencias de trabajos en este campo de la generación automatizada de música. Algunos ejemplos son los siguientes:

- K. Ebcioğlu. "An Expert System for Harmonizing Four-part Chorales", Computer Music Journal, vol.12, no. 3, 43-51 (1988).
- Yoge, N., & Lerch, A. (2008). "A System for Automatic Audio Harmonization" ,
- Yi, L. (1970). "Automatic Generation of Four-part Harmony"
- Hanlon, M., & Ledlie, T. (2002). "CPU Bach : An Automatic Chorale Harmonization"
- Eigenfeldt, A., & Pasquier, P. (n.d.). "Realtime Generation of Harmonic Progressions Using Constrained Markov Selection". Computer Music Journal.
- Lopez De Mantaras, R., & Arcos, J. L. (2002). "AI and Music From Composition to Expressive Performance". AI Magazine, 43-58.

2.2.5 La análisis automatizado (M.I.R)

El análisis automatizado de música es un campo con un enorme desarrollo y diversidad. En gran parte, este éxito se debe a la creciente disponibilidad de música en formatos digitales y a la consiguiente necesidad por parte del gran público consumidor, de herramientas que faciliten la localización de piezas afines a sus gustos.

Analizar música consiste en extraer rasgos de ella. Gracias a esto se puede caracterizar y clasificar, o bien simplemente anotarla con ciertos atributos, con objeto de pasar los resultados a otra etapa de análisis.

El éxito de este área de investigación ha dado origen a un término propio, MIR (Music Information Retrieval) y a numerosas actividades organizadas, como la creación de congresos, organizaciones, sitios web.

Son muchos los rasgos o aspectos que se pueden considerar de la música. A bajo nivel se pueden estarían cualidades esenciales, las llamadas *dimensiones*, de la música: altura, duración, timbre, intensidad.

A niveles intermedios se pueden considerar aspectos como tipo de métrica, armonías, ritmos, melodía, orquestación. A niveles más altos pueden buscarse *géneros* musicales.

Una vez extraídos los rasgos o cualidades buscados, puede realizarse con facilidad la tarea de la clasificación.

Cuando los rasgos o cualidades no son absolutos, se expresan mediante porcentajes o probabilidades. En estos casos la clasificación puede aplicarse usando el teorema de Bayes:

$$P(A|B) = P(B|A) P(A) / P(B)$$

$$P(\text{Cualidad}|\text{Clase}) = P(A|B) = P(\text{Clase}|\text{Cualidad}) P(\text{Cualidad}) / P(\text{Clase})$$

Una buena introducción a los conceptos de MIR es la realizada por Nicola Orio "Music Retrieval: A Tutorial and Review" [ORI06]. Puede servir de base para adentrarse en este vasto y variado campo del MIR.

Como se ha mencionado, el MIR es un campo muy activo y existe mucha información disponible en la Web. Por ejemplo, hay un portal dedicado en exclusiva al MIR:<http://www.music-ir.org/>. Lo mantiene *J. Stephen Downie*, una de las autoridades académicas más reconocidas en

este campo. En este portal se puede encontrar una extensa bibliografía, así como recomendaciones sobre herramientas, técnicas, congresos y todo tipo de información sobre el tema.

Existen también congresos mundiales sobre MIR que se organizan regularmente y en los que se presentan interesantes y variados artículos, a menudo descargables desde la Web. Particularmente interesantes son los congresos de [ISMIR - The International Society for Music Information Retrieval](http://www.ismir.net/) (<http://www.ismir.net/>)

2.2.6 Notaciones

El procesamiento automatizado de cualquier tipo de información requiere definir formatos que permitan el intercambio de datos. Un formato viene a ser sinónimo de lenguaje o de protocolo, ya que implica una serie de convenciones acordadas entre varias partes que quieren comunicarse.

2.2.6.1 **Niveles de representación la música**

Al hablar de música, podemos agrupar los formatos o notaciones en función del nivel de especialización del sonido, distinguiendo principalmente los siguientes niveles:

En el nivel más bajo están los formatos para representar audio digital, es decir, cualquier sonido. Representan la onda del sonido por medio de muestras tomadas a intervalos regulares.

En el nivel más alto están las notaciones para representar el Lenguaje Musical occidental, o Música Tonal; es decir, las partituras, y podemos denominarlos formatos musicales simbólicos.

En un nivel intermedio estarían los formatos especializados en ciertas cualidades de la música, y particularmente interesante es el del formato de la frecuencia o altura (pitch), ya que es la clave para facilitar el procesamiento de la música mediante ordenador.

A continuación se estudian algo más en profundidad los formatos para estos niveles.

2.2.6.2 **Formatos de audio digital**

Los formatos para representar sonidos analógicos de forma digital se suelen llamar también *formatos de audio digital* y se caracterizan porque son notaciones de bajo nivel, capaces de representar cualquier sonido en el rango perceptible por un humano, y pensadas para ser manejadas directamente por el ordenador.

Transformar la señal de audio analógica a información digital implica *muestrear* la señal a intervalos regulares. Cuanto mayor sea la *tasa de muestreo*, mayor será la *fidelidad* de la señal digital respecto a la original analógica.

Los formatos de audio incluyen generalmente también una compresión de la información obtenida tras el muestreo, con el objetivo de ahorrar recursos de almacenamiento. A los módulos que realizan esta compresión se les denomina *codecs*, y usan algoritmos distintos según el rendimiento y la fiabilidad deseados. Es importante señalar que a veces estos algoritmos están patentados, por lo que sus respectivos codecs solo pueden usarse bajo licencias restringidas.

En general, la compresión supone una pérdida de información respecto a la señal original, y esta pérdida suele ser mayor cuanto mayor es la *tasa de compresión* (relación entre la cantidad de bits antes y después de la compresión).

También es posible una compresión sin ninguna pérdida de información, pero ello supone una menor *tasa de compresión*.

Comparando la representación digital del audio con otras de mayor nivel de abstracción, se observa que esta presenta ciertas cualidades:

- máxima capacidad expresiva
- mínima estructuración
- multipropósito: orientado a la generación de sonido en general

Hay multitud de formatos de audio digital (MP3, MP4, AAC, FLAC, etc.) . A la hora de elegir uno se deben tener en cuenta aspectos como el tipo de licencia, la tasa de compresión, la fidelidad de la compresión, y la disponibilidad de reproductores y traductores para ese formato.

2.2.6.3 Formatos intermedios: MIDI

Entre el nivel de representación de audio digital, absolutamente genérico, y el de representación específico del Lenguaje Musical, se encuentra un formato a medio camino entre ambos, el MIDI.

Debido a su amplia difusión y consolidación, existe una extensa y variada gama de librerías y utilidades disponibles, por lo que su uso en una aplicación musical puede representar un importante ahorro de esfuerzo, además de mejorar la interoperatividad de la aplicación.

Es importante dejar claro que MIDI no es un formato de audio: no representa una onda sonora; simplemente contiene *órdenes*, es decir, *mensajes* con *instrucciones* dirigidas a un *sintetizador* para que éste genere música.

Los mensajes se pueden guardar en un fichero con formato específico: fichero MIDI.

No es un formato de alta calidad de sonido. Pero sí puede resultar muy práctico en entornos de aprendizaje, ya que es relativamente sencillo de usar.

El estándar también incluye el nivel físico: cables y conectores MIDI.

Tiene 16 canales, es decir, que puede manejar hasta 16 sonidos simultáneos, y para cada uno se puede elegir entre 128 instrumentos.

Existen dispositivos capaces de capturar sonidos y pasar a formato MIDI "secuenciadores MIDI".

Para conocer en profundidad este protocolo, se puede consultar la abundante información disponible en la Web, como, por ejemplo, el porta de la asociación de fabricantes MIDI (*MIDI Manufacturers Association*), donde se ofrecen todo tipo de tutoriales:

<http://www.midi.org/aboutmidi/tutorials.php>

2.2.6.4 Formatos de simbólicos para el Lenguaje Musical

Las notaciones de la música son simbólicas, ya que están destinadas a representar conceptos concebidos por la mente humana. Suelen codificarse en texto ASCII para poder ser manejables por humanos sin necesidad de herramientas especiales, es decir, con editores de texto convencionales.

Estas notaciones deben ser suficientemente expresivas como para poder representar todas las cadenas válidas, es decir, las frases posibles del lenguaje. Aplicado a la Música Occidental, esto se traduce en que la notación *debería* poder representar cualquier partitura posible. Pero, como ya se ha comentado en anteriores capítulos, el Lenguaje Musical occidental es dependiente del contexto y admite ambigüedades, por lo que al crear un lenguaje formal para representarlo, normalmente se pierde algo de capacidad expresiva a cambio de una mayor simplicidad. Para una gran mayoría de las musicales, y en particular para las de aprendizaje, es perfectamente asumible alguna limitación expresiva, porque se suele valorar más la sencillez, pues facilita el aprendizaje del lenguaje y su procesamiento.

Hay quien propone incluso una reforma del propio *Lenguaje Musical Occidental*, para simplificarlo y eliminar ambigüedades, como el caso de Nydana (<http://home.swipnet.se/nydana/>). Otros investigadores destacan también la dificultades innecesarias de la notación clásica, pero proponen cambios más modestos, como utilizar un *pentagrama cromático* en lugar de *diatónico*, como en el portal de "The Music Notation Project" (<http://musicnotation.org/>), donde se proponen:

"exploring alternative music notation systems — to make reading, writing, and playing music more enjoyable and easier to learn "

Sin entrar en debates sobre la conveniencia de reformar la notación clásica, sí que se puede afirmar que para el caso concreto del *análisis de armonía tonal*, sería muy práctico la utilización del pentagrama *cromático*, que tiene la significativa ventaja de mostrar de forma *directa* el valor de los *intervalos* con solo ver la posición de las notas en el pentagrama, mientras que en el pentagrama *diatónico* se necesitan, además de la posición, la clave, la tonalidad y las alteraciones.

Sin duda, si se va a procesar una partitura para analizar su armonía, y la notación es la tradicional *diatónica*, es aconsejable disponer de un *traductor* de la escala diatónica a otra con pasos homogéneos, ya que esto facilita enormemente el cálculo de los intervalos, que es la base de la armonía. Afortunadamente, existe una escala homogénea cuyo traductor a la diatónica es extremadamente simple, se trata de la notación en base 40 de Hewlett [HEW92], que sin duda alguna es la más recomendable si se va a procesar la armonía de música representada con la notación clásica diatónica.

Como se argumenta en "Chromatic Staves: A Better Approach to Music Notation" (<http://musicnotation.org/#intervals>):

"All of these features of traditional music notation combine to make reading music much more difficult than it might be with a better notation system. For an analogy, imagine trying to do arithmetic with Roman numerals. It can be done, but the notation system makes a big difference. Of course it is important to view traditional notation in its broader historical context and to keep in mind the innovations and reforms that it has undergone over time."

"Alternative music notation systems with chromatic staves avoid each of these pitch-related difficulties, and offer significant advantages over traditional music notation."

Otro aspecto relevante de las notaciones es que a menudo una aplicación musical crea su propio lenguaje o notación, para así poder integrar en él características propias de la aplicación. En

estos casos es muy recomendable implementar también un traductor a alguno de los lenguajes musicales más estandarizados, y en especial a MusicXML.

Respecto a las notaciones, conviene hacer una precisión: MIDI no es una notación del Lenguaje Musical Occidental; conceptualmente son muy distintos y, funcionalmente, MIDI es un subconjunto, es decir, que al traducir de MIDI a Lenguaje Musical, se pierde información.

Un tema aparte es la cuestión de intentar representar la música de culturas diversas. Para ello se están también realizando esfuerzos como el lenguaje MML (<http://www.musicmarkup.info/book/index.html>). Si bien es un tema interesante, no entraremos en él, porque queda fuera del alcance de este trabajo.

También es interesante, pero fuera del ámbito de este trabajo, la cuestión de estudiar el Lenguaje Musical desde la perspectiva de la *lingüística*. En este sentido, se recomienda consultar los trabajos de Lehrling & Jackendoff :

- "Toward a Formal Theory of Tonal Music", Journal of music theory, vol. 21, 1977
- "A deep parallel between music and language", Indiana, 1980
- "A Generative Theory of Tonal Music". The MIT Press, 1983

Un estudio desde el punto de vista de los lenguajes formales lo realiza Bryan Jurish en "*Music as a formal language*" [JUR04].

Un aspecto muy importante sobre las notaciones es que algunas de las principales cuentan con su propia librería de utilidades, lo cual es una importante ventaja, porque facilita enormemente su uso. Como ejemplo, MusicXML y GUIDO, aparte de estar entre las notaciones más extensamente utilizadas, cuentan también con su librería de apoyo.

Resumiendo el tema de la notación musical, se puede decir que hay una enorme variedad y que ninguna predomina. Generalmente se elige una en función del uso que se vaya a hacer con ella. Pero no hay que olvidar que, si se requiere interoperatividad, entonces se necesita poder traducir a una notación de uso extendido, como MusicXML.

Gerd Castan en su web sobre notaciones (<http://www.music-notation.info/>), da el siguiente consejo para elegir una notación:

"The best export formats at the moment are NIFF, MusicXML, abc and SCORE. Others are possible, but not MIDI, that loses almost all visual information. The internet is built on open standards."

Roger B. Dannenberg concluye lo siguiente sobre el tema:

- La Música se resiste a admitir una única representación.
- La notación tradicional resulta sencilla de leer a los músicos, pero muy poco adecuada para ser procesada.
- A menudo se trabaja con notaciones simplificadas o creadas a medida de las necesidades.

Para consultar sobre el tema de las notaciones musicales, se puede acudir a las siguientes fuentes de información:

El libro de referencia sobre las notaciones musicales es "Beyond MIDI: The Handbook of Musical Codes" de Eleanor Selfridge-Field [SEL97].

Gordon J. Callon mantiene una página web con enlaces a notaciones y al software que las soportan: "*Music Notation Software*" (<http://www.acadiau.ca/~gcallon/www/others.htm>).

En el portal de Gerd Castan (<http://www.music-notation.info/en/compmus/notationformats.html>) se puede encontrar un exhaustivo inventario de notaciones musicales y de las utilidades de software que las soportan. A continuación se resumen las notaciones más relevantes en una tabla.

XML	ASCII
MidiXML	PDF used as music notation format
MPEG4-SMR	DARMS
MNML	GUIDO Music Notation Language
MusicML	abc
MHTML	MusiXTeX, MusicTeX, MuTeX...
ScoreML	Musedata format (.md)
JScoreML	LilyPond
eXtensible Score Language (XScore)	TexTab
MusicXML (.mxl, .xml)	Mup music publication program
MusiqueXML	NoteEdit
GUIDO XML	Liszt: The SharpEye OMR engine output
WEDELMUSIC	file format
ChordML	Drum Tab
ChordQL	Enigma Transportable Format (ETF)
NeumesXML	CMN: Common Music Notation
JMSL Score	OMNL: Open Music Notation Language
Extensible Music Notation Markup Language	Scot: A Score Translator for Music 11
XMusic SongWrite	corpus mensurabilis musice electronicum
Music Description Language (MDL)	Nightingale Notelist
Music Catalog (MusiCat)	MusicKit ScoreFile
CapXML (.capx)	Common Hierarchical Abstract Representation of Music)
SMIL	Leadsheet Notation

Tabla 2: Algunos destacables formatos de representación musical

2.2.7 Editores de partituras y ayuda a la composición

Dentro del software musical, los programas más conocidos y usados son los que ayudan crear partituras. Algunos de ellos simplemente son *editores* de partituras y otros ,además, implementan algún tipo de *ayuda inteligente*, como, por ejemplo, analizando lo introducido y realizando sugerencias.

En el mundo profesional, *Sibelius* y *Finale* son las dos referencias en este campo, con algún otro también a su estela, como *Encore*. Todos son muy completos, ya que van dirigidos a los compositores y orquestadores *profesionales*, pero por ese motivo resultan demasiado sofisticados para su uso en el aprendizaje de la música. Otro factor negativo es su elevado precio, si bien tienen versiones simplificadas (incluso *Sibelius* tiene un *Sibelius Student*) y precios especiales para estudiantes y profesores.

Los programas edición musical se suelen denominar "music notation", ya que permiten crear partituras en la notación musical clásica.

Generalmente los editores son de tipo WYSIWYG, es decir, que al introducir datos nuevos, se ve directamente cómo queda, pero no todos son así, ya que tienen algunos inconvenientes

- Están algo limitados debido a la complejidad,
- Cuesta usarlos solo con teclado, por lo que pueden resultar lentos ,

En el aprendizaje musical se busca rapidez y agilidad para que el alumno no se desmotive y para que pueda aprovechar mejor su tiempo, realizando más ejercicios. Por ello, en este entorno conviene usar editores manejables con teclado, sin necesidad de recurrir al ratón.

Las ayudas inteligentes para la composición de los programas profesionales no resultan adecuadas para la enseñanza, pues están más orientadas a componer que a corregir y, por tanto, la información que muestran no es la adecuada para el aprendizaje.

En cuanto a programas profesionales de ayuda a la composición, destaca *Band-in-a-Box* (<http://www.pgmusic.com/>), capaz de generar arreglos completos a partir de unos pocos acordes.

A continuación se describen algunos de los editores de partituras más destacados.

Lilypond (<http://lilypond.org/>):

Es software abierto.

Notación propia en formato texto ASCII, conceptualmente similar a Latex. La ventaja es su mayor capacidad expresiva, que consigue muy buena calidad y versatilidad, incluso superior a Sibelius y Finale.

Presume de que se controla exactamente cómo va representarse la partitura. También dice que genera partituras sencillamente *más bonitas*.

El principal problema es que hay que aprender esta notación, que además es propia del programa, aunque dispone de traductores a otros formatos, como MusicXML. Otro punto negativo es la mala representación de acordes.

Otra ventaja es que se integra directamente en documentos Latex, HTML y OpenOffice.

MuseScore (<http://musescore.org/>):

Edito libre (GNU GPL), multiplataforma y WYSWYG. Además ,es bastante completo: permite una cantidad ilimitada de pentagramas y hasta 4 voces por pentagrama.

También permite introducción rápida de notas desde teclado.

Tiene *Secuenciador* (reproduce notación musical) y *sintetizador* (genera música en general) MIDI integrados.

En cuanto a interactividad, además de MusicXML, permite el formato Lilypond (*Finale* y *Sibelius* no lo soportan), lo que supone el acceso a repositorios de partituras como *Mutopia* (<http://www.mutopiaproject.org/>). También exporta a formatos de audio digital como FLAC o WAV.

Por todo ello, MuseScore parece ser el editor libre más completo y popular.

Music NotePad , ¡no confundir con *Finale Notepad!* ():

Editor libre que tiene funcionalidad bastante limitada ,ya que se orienta más a la generación de MIDI. Su principal atractivo es que es basa en la interesante librería JFugue, que es un API

sencillo, potente para generar sonidos en MIDI usando una notación propia. Otro aspecto interesante es que permite añadir extensiones (plug-in) a medida.

Roger B Dannenberg en su presentación sobre notaciones y editores "*Music as Data*" (<http://www.cs.cmu.edu/~music/cmsip/slides/week10-6-per-page.pdf>) concluye los siguiente:

Current Music Notation Systems:

- *Nearly all published music is now done with computers*
- *At least two powerful commercial systems: Finale, Sibelius*
- *Some open source editors*
- *Combine semantic-based layout with open-ended manual layout*
- *Lacking:*
- *Constraint systems to maintain manual placement when automatic spacing changes*
- *Tracking change from score to parts and vice-versa*
- *Robust interchange formats*
- *APIs giving programmable access to notation (but this exists in at least limited forms for more than one notation system)*

Para conocer más en profundidad los diversos editores, lo mejor es dirigirse a las comparativas:

- Wikipedia: List of Score Writers (http://en.wikipedia.org/wiki/List_of_scorewriters): lista con tabla comparativa.
- Music Notation Software (<http://www.acadiau.ca/~gcallon/www/others.htm>): descripción básica y enlaces interesantes, tanto de notaciones, como del software que soporta cada notación.
- Notation Software (http://www.music-software-reviews.com/notation_software.html) comparativa algo limitada, pero interesante.
- Music Notation Software Review (<http://music-notationsoftware-review.toptenreviews.com/>) exhaustiva comparativa de productos comerciales.

2.2.8 Ontologías

2.2.8.1 Introducción al concepto de Web Ontology Language

Las *ontologías* en el mundo de la computación se refieren a gramáticas específicas para modelar un dominio concreto. Proporcionan un vocabulario y unas reglas para describir los conceptos de ese dominio. Cuando el vocabulario se estructura jerárquicamente, forma una *taxonomía*.

Las ontologías se manejan habitualmente en el contexto de la Web Semántica, que es un esfuerzo por estructurar y anotar los contenidos planos de la Web para poder gestionarlos en función de su significado. En este contexto, las ontologías se denominan *Web Ontology Language*, OWL (http://en.wikipedia.org/wiki/Web_Ontology_Language).

Además de en el contexto de la Web Semántica, las ontologías pueden resultar útiles en otros contextos. En particular pueden servir para facilitar el modelado de ciertos procesos. En el contexto del Diseño Orientado a Objetos, pueden dar una idea de cómo debe ser la taxonomía de clases. Por ejemplo, si se encuentra un OWL adecuado para modelar los acordes, puede servir de utilidad para diseñar las clases que gestionen los acordes, e incluso el proceso mismo de creación de acordes.

Para clarificar los conceptos de ontología, taxonomía, vocabulario y meta-modelo se puede consultar la página web "*What are the differences between a vocabulary, a taxonomy, a thesaurus, an ontology, and a meta-model?*" (<http://infogrid.org/wiki/Reference/PidcockArticle>)

Para entender lo que representa un OWL en el contexto de la música: "A Music Ontology primer" (<http://ismir2009.dbtune.org/slides/mo.html>)

"*A Web ontology is...*

- *a set of web identifiers for concepts and relationships in a domain*
- *a set of web identifiers different datasets can refer to when they deal with the same kind of thing*
- *a set of axioms characterising those concepts and relationships*
- *specified in RDF, using one of the flavors of OWL*
- *has its roots in Description Logics "*

2.2.8.2 Ontologías de Música Tonal

Existen diversas ontologías musicales, con diversos objetivos. Principalmente se pueden agrupar en dos tipos:

Las orientadas a *describir* las **producciones** musicales

Las orientadas a *representar* la **música** de la propia pieza.

A continuación se describen algunas, muchas de ellas desarrolladas en el marco de OMRAS2, en el *Centre for Digital Music, Queen Mary, University of London* (<http://www.omras2.com/>).

"*Omras2 is a framework for annotating and searching collections of both recorded music and digital score representations such as MIDI. The project is funded by the EPSRC*".

2.2.8.2.1 Music Ontology Specification

Esta ontología es del tipo orientada a describir la producción musical.

Sitio web:

<http://musicontology.com/>

Especificación:

<http://purl.org/ontology/mo/> (RDF/XML, Turtle)

Autores:

Yves Raimond - BBC, Frédéric Giasson - Structured Dynamics

2.2.8.2.2 The Tonality Ontology

Orientada a describir la *Música Tonal*

Sitio web:

<http://motools.sourceforge.net/doc/tonality.html>

Especificación:

<http://purl.org/ontology/tonality/>

Autores:

David Pastor Escuredo

2.2.8.2.3 La ontología de acorde "OMRAS2 Chord"

Orientada a describir acordes de la música tonal y basada en la notación de Christopher Harte *Symbolic Representation of Musical Chords: A Proposed Syntax for Text Annotations* [HAR02].

Esta ontología está creada desde la perspectiva del procesamiento automatizado de la música, y por tanto puede resultar muy adecuada para describir los conceptos de armonía tonal en un formato interoperable entre humanos y ordenadores.

Varios estudios y trabajos de análisis de armonía se han basado en esta ontología de C. Harte, por lo que es recomendable estudiarla y evaluar su uso si se pretende realizar un analizador de armonía.

En el presente trabajo no se llegó a usar esta ontología porque inicialmente se desconocía. Sin embargo, la taxonomía de clases es similar, e incluso en algunos aspectos la de este trabajo es más potente y desarrollada, pues, por ejemplo, permite integrar perfectamente el concepto de voces en las clases de los acordes, lo que no está contemplado en la ontología de *OMRAS2 Chord*.

Sitio web:

http://motools.sourceforge.net/chord_draft_1/chord.html

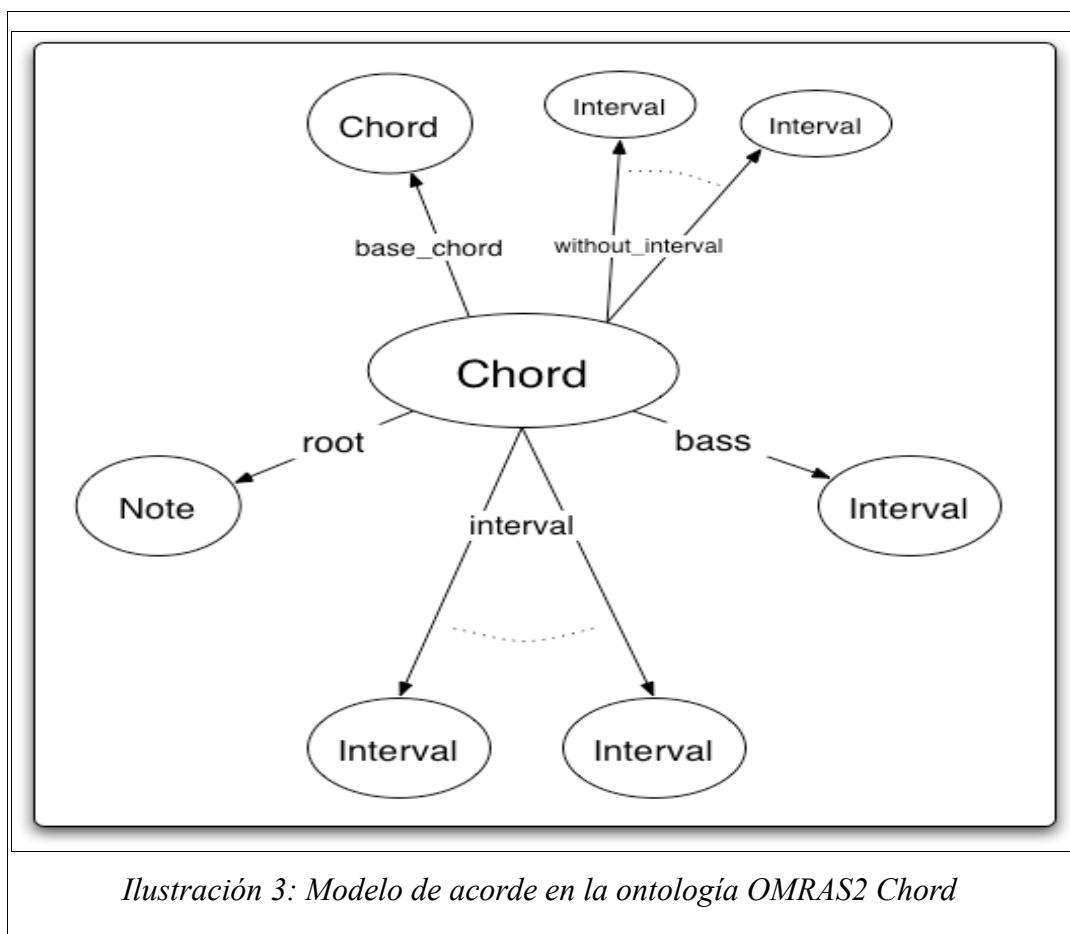
Especificación:

<http://purl.org/ontology/chord/>

Autores:

Christopher Sutton, Yves Raimond, Matthias Mauch

Detalles:



- La clase *chord* representa un acorde.
- Las propiedades de la clase son todas opcionales, por lo que una instancia pueda estar vacía de información, representando un acorde *aún sin definir*.
- Un acorde se forma por la nota fundamental (*root*) y sus intervalos (*interval*).
- Las inversiones se representan indicando qué intervalo contiene el bajo (*bass*).
- No se permite, por el momento, incluir información sobre voces.
- Si se conocen los nombres de las notas, se usa ScaleInterval, en caso contrario debe usarse SemitoneInterval
- El tipo de acorde se representa mediante un "acorde base" (*base_chord*), el cual contiene los acordes nominales del tipo. El acorde real compartirá los acordes nominales, salvo los especificados mediante "sin-intervalo" (*chord_without_interval*).

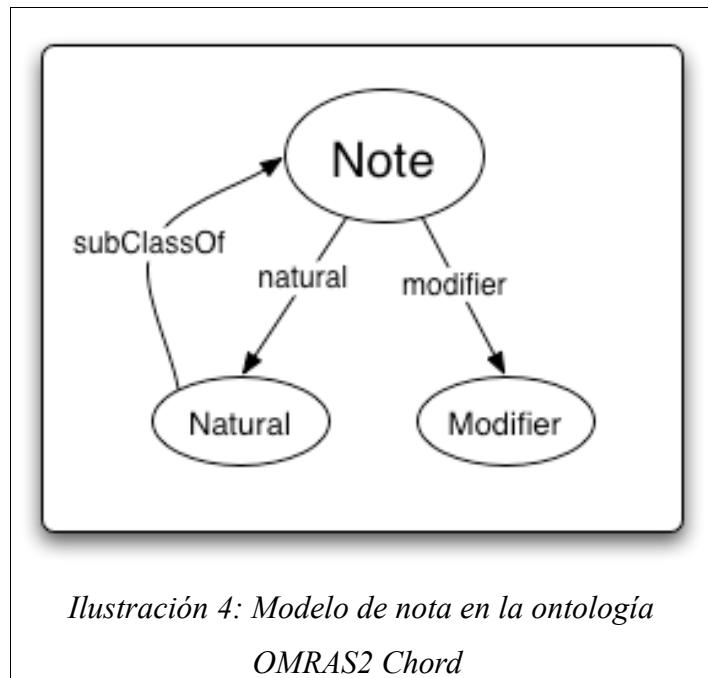


Ilustración 4: Modelo de nota en la ontología
OMRAS2 Chord

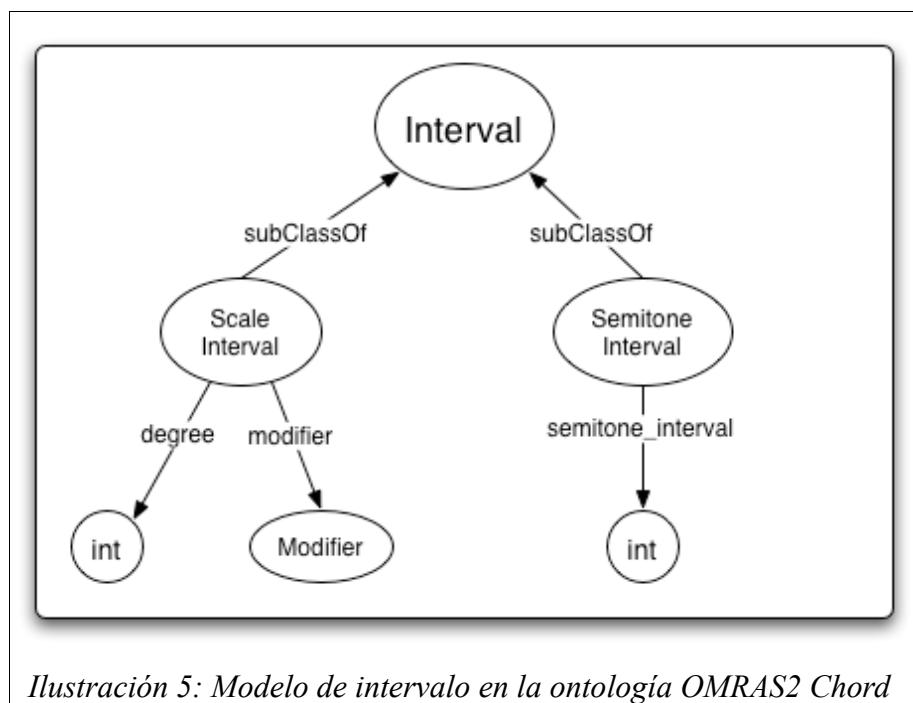


Ilustración 5: Modelo de intervalo en la ontología OMRAS2 Chord

A continuación se muestra D# menor con novena añadida y tercera bemol omitida, sobre quinta.

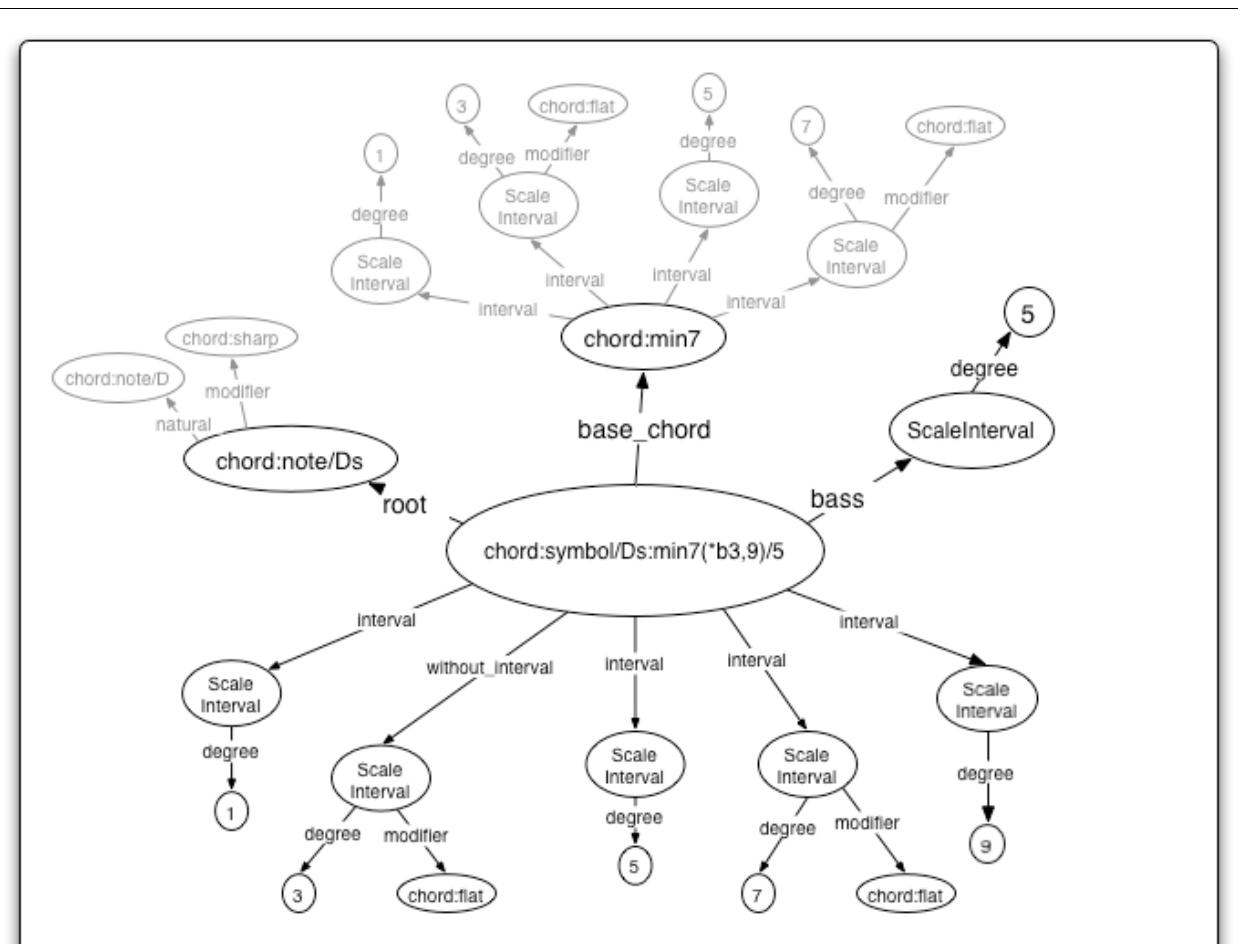


Ilustración 6: Ejemplo de descripción de acorde en la ontología OMRAS2 Chord

El diagrama anterior se corresponde con la siguiente descripción en notación RDF/N3:

```

<http://purl.org/ontology/chord/symbol/Ds:min7(*b3,9)/5>
  a chord:Chord;
  chord:root [a chord:Note; chord:modifier chord:sharp;
               chord:natural <http://purl.org/ontology/chord/note/D> ];
  chord:bass [a chord:ScaleInterval; chord:degree 5 ];
  chord:base_chord chord:min7;
  chord:without_interval [a chord:ScaleInterval; chord:degree 3 chord:modifier
  chord:flat ];
  chord:interval [a chord:ScaleInterval; chord:degree 1 ],
                  [a chord:ScaleInterval; chord:degree 5 ],
                  [a chord:ScaleInterval; chord:degree 7;chord:modifier chord:flat ],
                  [a chord:ScaleInterval; chord:degree 9 ].

```

2.2.8.2.4 Otras ontologías relacionadas con la música

Otras ontologías relacionadas con las anteriores y complementarias a ellas son las siguientes:

Prefix	XML Namespace	Description
timeline	http://purl.org/NET/c4dm/timeline.owl#	The TimeLine ontology
event	http://purl.org/NET/c4dm/event.owl#	The Event ontology
tuning	http://purl.org/ontology/tuning/	Tuning systems description

2.2.8.2.5 Utilidades para procesar ontologías musicales

Lo lenguajes de ontologías web (OWL) están pensados para facilitar el procesamiento de la información relativa a ciertos dominios. Por tanto, la ontología en sí misma resultará de poca utilidad si no se acompaña de herramientas que facilite su uso.

En la web de Music Ontology (MO) hay una página con herramientas varias (<http://www.omras2.org/motools>, <http://sourceforge.net/projects/motools/>). MoPy es una especie de traductor de MO a clases Python; sería interesante disponer de algo similar para Java y C++. Entre las herramientas hay también un visualizador de armonía ([Harmonic Visualizer](#)), pero sus funcionalidades son muy limitadas y trabaja a muy bajo nivel: la entrada que procesa es la onda del audio.

2.2.9 Lo que hay en software musical para la aprendizaje

A continuación, se intenta buscar antecedentes y referencias que permitan facilitar el diseño del analizador tonal para la enseñanza. Se repasan algunos de los casos más relevantes del software musical en el contexto del aprendizaje de la música.

2.2.9.1 Visión general del software de educación musical

Aunque hay una gran variedad de productos de software de educación musical, se observan muchas similitudes entre ellos, pues la mayoría comparten unos objetivos y estrategias comunes.

Estos objetivos comprenden siempre aprender los fundamentos de la teoría musical, pero además se busca trabajar las cuatro *áreas de la percepción musical*:

- ✓ Reconocimiento visual de elementos del lenguaje (lectura).

- ✓ Notación escrita.
- ✓ Entrenamiento auditivo.
- ✓ Destreza instrumental.

Otras características adicionales destacables son las relacionadas con las ayudas al tutor :

- ✓ Seguimiento del progreso.
- ✓ Adaptación de los cursos a los objetivos pedagógicos.

En función de la estrategia instructiva o de la perspectiva pedagógica, se orientará más hacia el seguimiento de un plan prefijado, o bien hacia promover que el alumno aprenda descubriendo por sí mismo.

En general todo software educativo hace uso en alguna medida de técnicas de *Inteligencia Artificial*, pero en algunos casos se busca expresamente utilizar al máximo IA para conseguir mejorar el resultado adaptando continuamente el proceso instructivo al alumno en función de la respuesta de este y del plan o modelo pedagógico.

Una importante ventaja del uso del ordenador es que el alumno siente menos presión al realizar su ejercicio sin audiencia humana.

Tradicionalmente, la mayoría de los programas dedicados a la educación musical concentran sus esfuerzos en enseñar la notación musical (el Lenguaje Musical) y el entrenamiento auditivo. Mediante test auditivos o dictados, la aplicación evalúa destrezas de reconocimiento básicas, como intervalos, acordes, cadencias, patrones de ritmos y melodías, etc. Los ejercicios son del tipo pregunta/respuesta. La generación de audio se realiza mediante sintetizadores, como los secuenciadores MIDI.

GUIDO (Hofstetter 1975, University of Delaware) es la aplicación de referencia en este campo. Incluía ciertas técnicas de IA de, ya que, en función de las respuestas del alumno, seleccionaba las siguientes preguntas y ajustaba la velocidad del dictado, intentando así guiar al alumno.

Otras aplicaciones comerciales posteriores que han seguido el mismo modelo son: Practica Musica, Music Ace, MIBAC Music Lessons.

Este tipo de programas ha ido evolucionando incorporando la teoría musical correspondiente a los ejercicios, convirtiéndose paulatinamente en entornos que integran multimedia e hipertexto.

Analizador tonal en software libre

Music Logo se sitúa en una perspectiva puramente constructivista. Pretende que el alumno aprenda a base de construir y probar modelos. Se basa en el concepto del lenguaje LOGO , un lenguaje de programación que encarna la filosofía de "aprender sin ser enseñado"

De entre los muchos programas desarrollados para facilitar la enseñanza de la música mediante ordenador, se destacan a continuación algunos de los más relevantes, haciendo especial énfasis en aquellos dedicados a la enseñanza de la *armonía*.

Para conocer más en profundidad detalles de estos programas y su contexto se pueden consultar a Simon Holland. "*AI Education and Music*" [HOL89] y Marcio Brandao. "*Computers in Music Education*" [BRA99].

Aplicación	Objetivos	Estrategia instructiva	Detalles
GUIDO (Hofstetter, 1988)	Teoría. Lectura. Escritura. Entrenamiento auditivo.	Aprendizaje programado.	Pregunta respuesta.
MiBAC Lessons	Teoría. Lectura. Escritura. Entrenamiento auditivo. Destreza instrumental.	Aprendizaje programado.	MIBAC I: Fundamentos. MIBAC II: armonía. Tan completo como Practica Musica, pero más sencillo.
Music Ace	Teoría. Lectura. Escritura. Entrenamiento auditivo.	Aprendizaje programado y exploratorio.	24 lecciones. Juegos variados., Ayuda a la composición.
Music Ace Maestro	Teoría. Lectura. Escritura. Entrenamiento auditivo.	Aprendizaje programado	Orientado al profesor. Lecciones adaptables.
Practica Musica	Teoría. Lectura. Escritura. Entrenamiento auditivo.	Aprendizaje programado y exploratorio.	Ayudas al profesor. Completo. Seguimiento del progreso.
Piano Tutor (Dannenberg, 1990)	Destreza instrumental.	Ejercicios y práctica.	Sistema experto para enseñar piano. Experimental.
Tune Master (Kirshbaum, 1986)	Destreza instrumental.	Exploratorio.	Tocar de oído mediante tableta táctil.
PianoFORTE (Smoliar et al., 1995)	Destreza instrumental.	Ejercicios y práctica. Guiado y seguimiento.	Realimentación por visualización de errores.
Music Logo (Bamberger, 1974)	Composición.	Exploratorio.	Micromundo LOGO.
LOCO (Desain and Honing, 1986)	Composición	Exploratorio	Micromundo LOGO.
Object LOGO (Greenberg,	Composición	Exploratorio	Micromundo LOGO.

1988)			
VIVACE (M. Thomas, 1985)	Armonía.		Sistema de IA basado en reglas. Modela el conocimiento del dominio
THE MUSES (Sorisio, 1987)	Armonía.		Sistema de IA. Modela el experto en armonía y el tutor.
Harmony ITS (Tobias, 1988)	Armonía.		Sistema IA enfocado a representar el conocimiento del dominio de la armonía mediante lógica con restricciones.
Harmony Space (Simon Holland, 1989)	Armonía.	Exploratorio	Herramienta interactiva. Basada en la teoría de Balzano
MOTIVE (Smith and Holland, 1994)	Composición melódica	Exploratorio	Basada en la teoría de Narmour.
MC (Holland and Elsom-Cook, 1990)	Composición.	Exploratorio.	Infraestructura de apoyo cognitivo.
Harmony Practice 3 (Fausto Torre, 2007)	Armonía a 4 voces clásica	Aprendizaje exploratorio: corrección de creaciones libres.	Reconocimiento de acordes y aprendizaje de reglas de armonía.
Solfeo.org (Ricchi Adams)	Lectura musical y entrenamiento auditivo	Aprendizaje exploratorio: corrección de creaciones libres. Ejercicios para completar.	<i>On-line:</i> http://solfeo.org/
Harmony Coach (John W. Schaffer 1988)	Acompañamiento armónico de melodías.	Ejercicios para completar.	Sistema IA: ITS (tutorial inteligente). Adaptativo en función de la respuesta del alumno.

Tabla 3: Programas para educación musical destacables y estrategias instructivas

2.2.9.2 Librerías de procesamiento de música

Existen diversas librerías con utilidades para procesar sonidos musicales. En general están muy orientadas a la generación (sintetización) más que al análisis, por lo que no resultan muy útiles de cara a implementar un analizador de armonía.

Otro aspecto negativo es que no suelen separar los niveles de representación de la música : audio, MIDI, Lenguaje Musical. Tienden a mezclar los tres niveles, por lo que no resultan útiles si solo se quiere utilizar por ejemplo el nivel superior.

Recordamos de nuevo la conclusión de François Pachet en su estudio "*Musical Harmonization with Constraints: A Survey*" de 2001 [PACH01], de que la armonización a 4 voces se ha conseguido ya resolver gracias a superar las aproximaciones "*de fuerza bruta*" (que intentaban analizar la armonía directamente desde el audio), gracias a que se ha estructurado la representación a un nivel superior de abstracción:

"The technical problem of four-voice harmonization may now be considered as solved,, and an adequate structuring of the problem to handle chord variables properly. This result comes after several years of trials and errors, starting from brute force approaches (e.g. Schottsdtaedt), to proprietary constraint languages (Ebcioğlu), to arc-consistency techniques (Ovans), augmented with adequate problem structuration (Pachet & Roy)."

Conclusión: si se quieren manejar los acordes como elementos de la Música Tonal, se debe realizar su procesamiento partiendo de los propios elementos de la Música Tonal (notas, tonalidades, alteraciones, etc.). Porque partir de niveles inferiores (MIDI o audio digital) complica enormemente la tarea de obtener todas las propiedades de los acordes.

2.2.9.2.1 CSound

Entorno de sintetización de sonidos muy potente y fácil de usar. Las órdenes se especifican en un fichero de texto en un formato propio.

Tiene una parte para análisis de armonía ,pero se basa en conceptos matemáticos muy particulares, que no se adaptan a las necesidades de la armonía tonal. Se adjunta un extracto de su manual de referencia como ejemplo.

csound::Voicelead Class Reference

```
#include <Voicelead.hpp>
```

Detailed Description

This class contains facilities for voiceleading, harmonic progression, and identifying chord types.

See: http://ruccas.org/pub/Gogins/music_atoms.pdf

```
static double csound::Voicelead::euclideanDistance (const std::vector<  
double > & chord1, const std::vector< double > & chord2) [static]
```

Return the Euclidean distance between two chords, which must have the same number of voices.

```
static std::vector< std::vector<double> > csound::Voicelead::inversions  
(const std::vector< double > & chord) [static]
```

Sitio web: <http://www.csounds.com/>

2.2.9.2.2 jMusic

Librería java para composición de sonidos musicales.

Aporta utilidades interesantes:

- Estructura de datos interna que representa un partitura.
- Renderización de partituras.
- Conversión de notaciones.
- Parte de análisis de armonía, pero muy limitada, como se muestra a continuación,

Chord Analysis

```
static int[] getSecondPassChords(Phrase phrase, double beatLength, int  
tonic, int[] scale)
```

Parameters:

phrase: is the phrase to be analysed

beatLength: the length of a beat, generally 1.0 which represents crotchets

tonic: a int representing the MIDI value pitch of the tonic, for example C can be represented by 0 or 60 or indeed any multiply of 12, C# by 1 or 61, G by 7 or 67.

scale: one of PhraseAnalysis.MAJOR_SCALE or PhraseAnalysis.MINOR_SCALE. If necessary, other scales can be implemented using the same format as these examples.

Return value: array of ints.

The length of the array represents the number of beats in the phrase.

Element [0] represents the first beat of bar 1, [1] the second beat of bar 1, [2] the third beat of bar one, and so.

The value of the element will always be in the range of 0 - 7. These values correspond to the following chords: I to VII

7: no particular chord, or same as previous. This generally occurs when you have things like minims or semibreves. The down beat will have a chord specified, but the second and consequent beats of the note will have 7s.

La generación de sonidos musicales es relativamente sencilla. Se muestra un ejemplo a continuación.

```
// this class that also renders the score as an audio file

import jm.JMC;
import jm.music.data.*;
import jm.util.*;
import jm.audio.*;

public final class SonOfBing implements JMC{

    public static void main(String[] args){

        Score score = new Score(new Part(new Phrase(new Note(C4, MINIM)) ));
        Write.midi(score);
        Instrument inst = new SawtoothInst(44100);
        Write.au(score, inst);

    }
}
```

Sitio web de jMusic: <http://jmusic.ci.qut.edu.au/index.html>

2.2.9.2.3 JFuge

JFugue es otra librería Java para sintetizar sonidos musicales a través de MIDI. Pretende simplificar al máximo el uso de MIDI, y en gran medida lo consigue.

Por desgracia, son muy escasas sus funcionalidades orientadas al análisis.

Notación musical propia MusicString en la que se incluyen también directrices para MIDI.

Ejemplos:

```
//se basa en patrones de MusicString
Pattern pattern1 = new Pattern("C5q D5q E5q C5q");
Pattern song = new Pattern();
song.add(pattern1, 2); // Adds 'pattern1' to 'song' twice
// Combina notación musical con parámetros midi (canal, instrumento...)
```

Analizador tonal en software libre

```
player.play("T[Adagio] V0 I[Piano] C5q F#5q CmajQ V1 I[Flute] C3q+E3q E3q+G3q Ri  
C2majI");  
  
// Permite manejar intervalos  
public static void main(String[] args) {  
    IntervalNotation riff =  
        new IntervalNotation("<1>q <5>q <8>q <1>majH");  
    Player player = new Player();  
    player.play(riff.getPatternForRootNote("C5"));  
    player.play(riff.getPatternForRootBote("Ab6"));  
  
    // Tiene ritmos  
    // This is a complete program for a 16-Beat Rock Rhythm  
    public static void main(String[] args) {  
        Rhytm rhythm = new Rhytm();  
        rhythm.setLayer(1, "O..oO...O..oO...");  
        rhythm.setLayer(2, "...*....*....*....*..");  
        rhythm.setLayer(3, "^^^^^^^^^^^^^^^^^");  
        rhythm.setLayer(4, "...!....!....!");  
        rhythm.addSubstitution('O', "[BASS_DRUM]i");  
        rhythm.addSubstitution('o', "Rs [BASS_DRUM]s");  
        rhythm.addSubstitution('*', "[ACOUSTIC_SNARE]i");  
        rhythm.addSubstitution('^', "[PEDAL_HI_HAT]s Rs");  
        rhythm.addSubstitution('!', "[CRASH_CYMBAL_1]s Rs");  
        rhythm.addSubstitution('.', "Ri");  
  
        Pattern pattern = rhythm.getPattern();  
        pattern.repeat(4);  
        Player player = new Player();  
        player.play(pattern);  
    }  
  
    // sencilla entrada/salida MIDI  
    DeviceThatWillReceiveMidi device =  
    new DeviceThatWillReceiveMidi(MidiDevice.Info);  
    sequence = player.getSequence(pattern);  
    device.sendSequence(sequence);  
    // Also: sequence = MidiSystem.getSequence(File);  
    DeviceThatWillTransmitMidi device =  
    new DeviceThatWillTransmitMidi(MidiDevice.Info);  
    device.listenForMillis(5000);  
    Pattern pattern = device.getPatternFromListening();  
  
    // Interoperabilidad  
    // exportar/importar en ficheros MusicString  
    pattern.savePattern(File)  
    Pattern pattern = Pattern.loadPattern(File)  
    // o formato MIDI  
    player.saveMidi(Pattern, File)  
    Pattern pattern = Pattern.loadMidi(File)  
    // o MusicXML
```

Se pueden encontrar aplicaciones variadas creadas a partir de JMusic en: <http://blogs.sun.com/geertjan/>. Entre otras, JFrets para aprender guitarra ,y "Music NotePad" para escribir y reproducir música.

2.2.9.2.4 CLAM

CLAM proclama ser una *infraestructura* completa para desarrollar aplicaciones de audio y música. Dice incluir un modelo de representación de las partituras (*conceptual model*).

Realmente CLAM aporta conceptos novedosos e innovadores. No es una librería típica que se integra en la aplicación programáticamente a través de un API, sino que sigue la tendencia moderna de integración visual, a base de conectar componentes, a través de un interfaz de usuario gráfico.

De esta forma, construir una aplicación consiste en algo similar a crear un diagrama de flujo, seleccionando componentes y definiendo su interconexión y configuración. Esta entorno de programación visual es similar al Simulink (<http://es.wikipedia.org/wiki/Simulink>).

El concepto de los módulos de procesamiento que se combinan a voluntad es habitual en el campo del procesamiento digital de señales y está también presente, por ejemplo, en la librería CSound , con sus *opcodes* o *unit generators*.

CLAM consigue resultados brillantes, incluso espectaculares, al procesar los acordes de una pieza musical, ya que los muestra visualmente. La aplicación Chordata, basada en CLAM, es la que realiza este análisis.

En este interesantísimo tutorial se explica el proceso de reconocimiento de acordes, que es del tipo '*fuerza bruta*', ya que se basa en aplicar sucesivos *filtros matemáticos* a la onda del sonido original. Una de las consecuencias es que los acordes no se reconocen con exactitud, sino como una distribución de probabilidad.

Una de las '*redes*' de filtros es la llamada *tonalAnalysis*, que reconoce los acordes y sus cualidades.

http://clam-project.org/wiki/Network_Editor_tutorial#Tonal_Analysis

Los procesamientos sucesivos que componen la red *TonalAnalysis* se muestra en la siguiente figura.

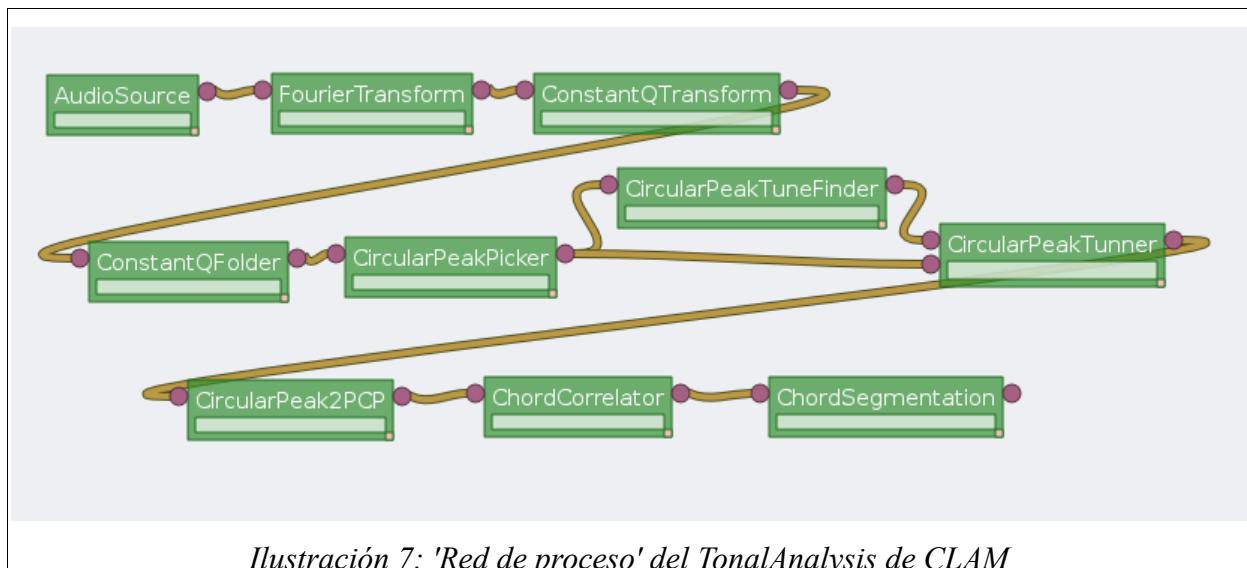


Ilustración 7: 'Red de proceso' del TonalAnalysis de CLAM

Sobre el resultado anterior se aplican otros pasos de procesamiento, como el 'chord correlation':

*"Chord correlation is built by adding the pitch by pitch product of the resulting pitch profile with a collection of ideal profiles for the different chords we want to try. This gives us, for each chord the **probability** of being the one sounding in a given instant."*

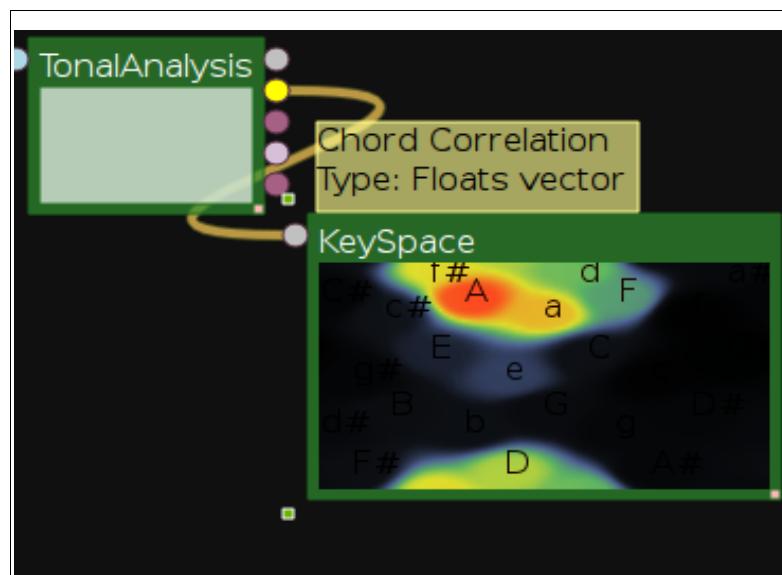


Ilustración 8: Vista 'Key Space' de CLAM, que muestra la **probabilidad** de los acordes

Otra aplicación de CLAM, el programa **ChordData**, muestra los acordes de una canción usando variadas y sorprendentes representaciones. Eso se describe en la wiki del proyecto:

http://clam-project.org/wiki/Chordata_tutorial

"Chordata is an easy to use application, built using the CLAM framework, to analyze the tonal features (harmony, chords...) of your digital music and explore them by navigating along the song using different useful views. It is addressed to amateur musicians who want to know the chords of their favorite songs but it is also very useful to get insight of the music itself."

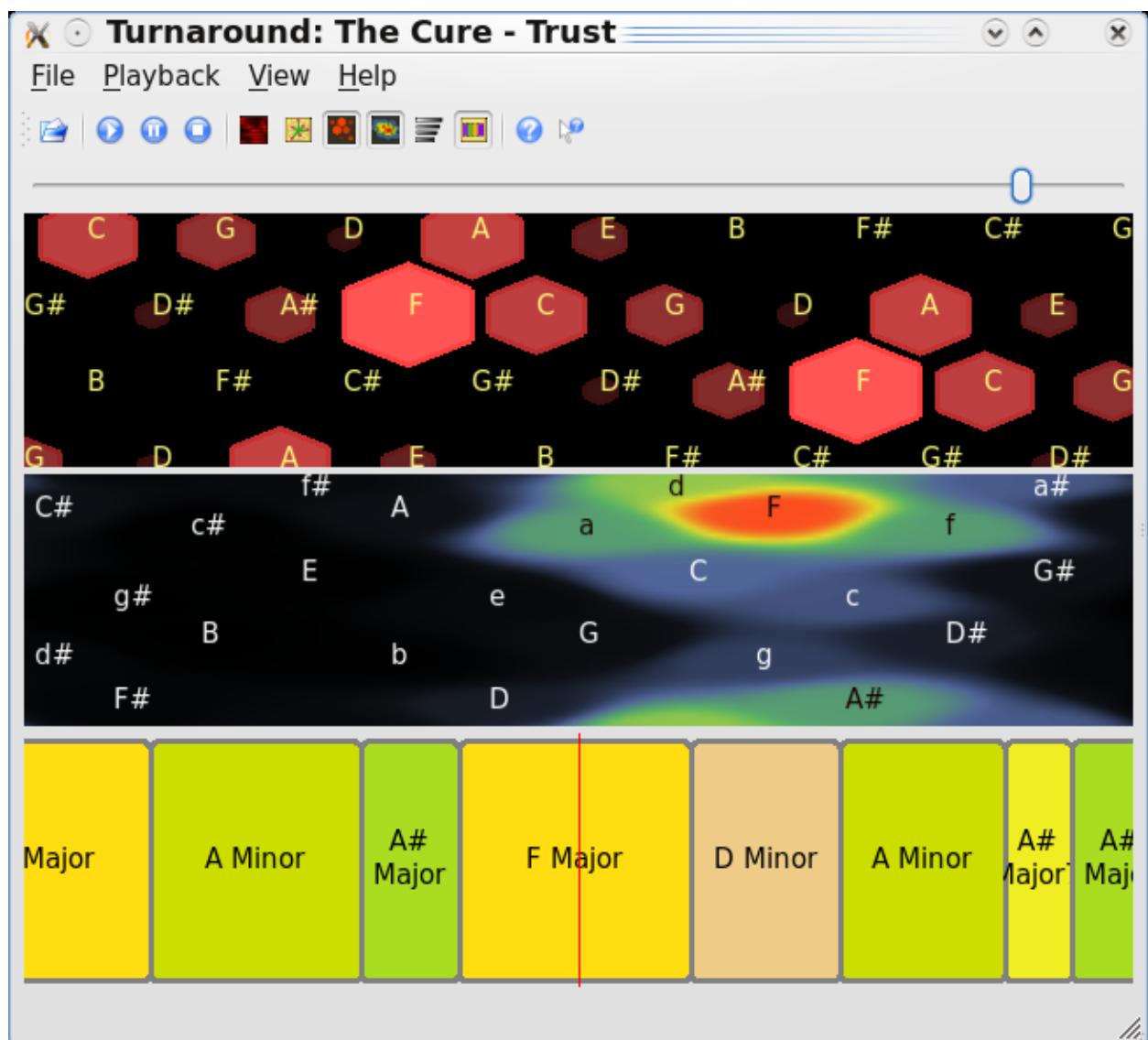


Ilustración 9: El ChordData de CLAM visualiza distintas 'dimensiones' de la armonía

Music Annotator (http://clam-project.org/wiki/Music_Annotator) es otra interesante aplicación de CLAM, en la misma línea de las anteriores de extraer propiedades musicales a partir

del audio y visualizarlas. En este caso, los atributos que se desean extraer se pueden configurar en un fichero XML.

El procesamiento de acordes se basa en los trabajos de Christopher **Harte**, "Automatic Chord Identification using a Quantised Chromagram" y "Characterizing harmony from audio" (http://www.nyu.edu/classes/bello/MIR_files/8_harmony.pdf). Harte es, además, el creador de la ontología para acordes OMRAS2 (www.omras2.org/ChordOntology).

CLAM es un proyecto dirigido por el investigador español Naviero Matriarca (<http://xavier.amatriain.net/>).

Comentario sobre CLAM desde la perspectiva de su uso para la enseña de la música:

A pesar de la versatilidad, incluso de la espectacularidad de CLAM, adolece de ser un entorno muy orientado a trabajar directamente y exclusivamente con el *audio* digital, por lo que resulta de poca utilidad a las aplicaciones que pretenden trabajar con *notación musical*, para lo que se precisa separar por niveles la representación de la música, y realizar el procesamiento de la armonía sobre el nivel más alto, el de la notación musical.

Con el mecanismo de CLAM, los acordes no se reconocen con precisión. Mucho menos se recompone la partitura entera, que es algo necesario para el aprendizaje de la teoría musical.

Sin embargo, los resultados son espectaculares, en especial teniendo en cuenta que se basan en procesamiento de 'fuerza bruta' y pueden ser muy útiles en la enseñanza de la música; quizás no para la teoría, pero sí para la práctica, por ejemplo, para comparar la 'visualización' de la pieza interpretada por el alumno con la original.

CLAM tiene enormes posibilidades para la enseñanza de la música, como un complemento visualmente muy atractivo para los alumnos. Vale la pena investigar más a fondo sus posibilidades.

2.2.9.3 Servidores web interactivos para aprendizaje on-line

Algunos cursos de música están alojados en Internet y son accesibles por medio de un navegador web. En general no tienen propósito comercial y adolecen de bastantes carencias,

comparando con los cursos descargables y ejecutados en el cliente. Sin embargo, tienen otros aspectos positivos que se destacan a continuación:

Ventajas:

- Permite seguimiento remoto de tutor humano.
- Actualizable de inmediato.
- Extensible fácilmente.
- Acceso a recursos de otros servidores.
- Posibilidad de beneficiarse de estándares y utilidades de eLearning.
- Permite desarrollo *colaborativo*.
- Posibilidad de comunicarse y compartir información mediante chats o P2P.
- Seguimiento de grupos; por lo tanto, posibilidad de analizar la eficacia y mejorar.

Inconvenientes:

- Limitaciones funcionales del navegador web en cuanto a visualización y reproducción de sonidos.
- Acceso a Internet necesario.

Para profundizar en el tema del aprendizaje de música *on-line*, se pueden consultar los siguientes documentos accesibles en Internet.

David Hebert, "Five Challenges and Solutions in Online Music Teacher Education"

<http://www.stthomas.edu/rimeonline/vol5/hebert.htm>

Bond, Austin. "Learning music online An accessible learning program for isolated students"

<http://www.ncver.edu.au/research/proj/nr1013.pdf>

Un ejemplo de lo sencillo que resulta construir un repositorio de lecciones musicales en la web:

GHT Music video lessons, <http://www.hgtmusic.com/links/?id=25>

A continuación se describen algunos portales web dedicados a la enseñanza de la música.

2.2.9.3.1 Curso de teoría de la música

URL: <http://www.teoria.com/indice.htm>

Autor: José Rodríguez Alvira.

Idioma: Español.

Aprendizaje de teoría musical de forma interactiva. Está organizado como un repositorio de material de teoría, ejercicios y diversos materiales, que crece continuamente.

Consta de varios apartados:

- Ejercicios de destrezas teóricas y auditivas.
- Referencia sobre teoría de la música.
- Análisis musicales y artículos sobre música.

Aunque no es muy profundo y riguroso en la teoría, la posibilidad de combinar ejercicios auditivos con teoría y consulta de otros materiales hacen que resulte interesante para principiantes.

2.2.9.3.2 Music Awareness

URL: <http://www.musicawareness.com/>

Autor: Music Awareness

Idioma: inglés

Curso de iniciación a la armonía que pretende conseguir un aprendizaje rápido basado en ejemplos sencillos y ejercicios básicos de armonización.

La teoría resulta bastante limitada y los ejercicios, aunque simples, pretenden resultar prácticos y entretenidos, como, por ejemplo, el interesante y curioso *Harmony-toy*, que permite jugar a armonizar melodías.

<http://www.musicawareness.com/HarmonyToy1.swf>

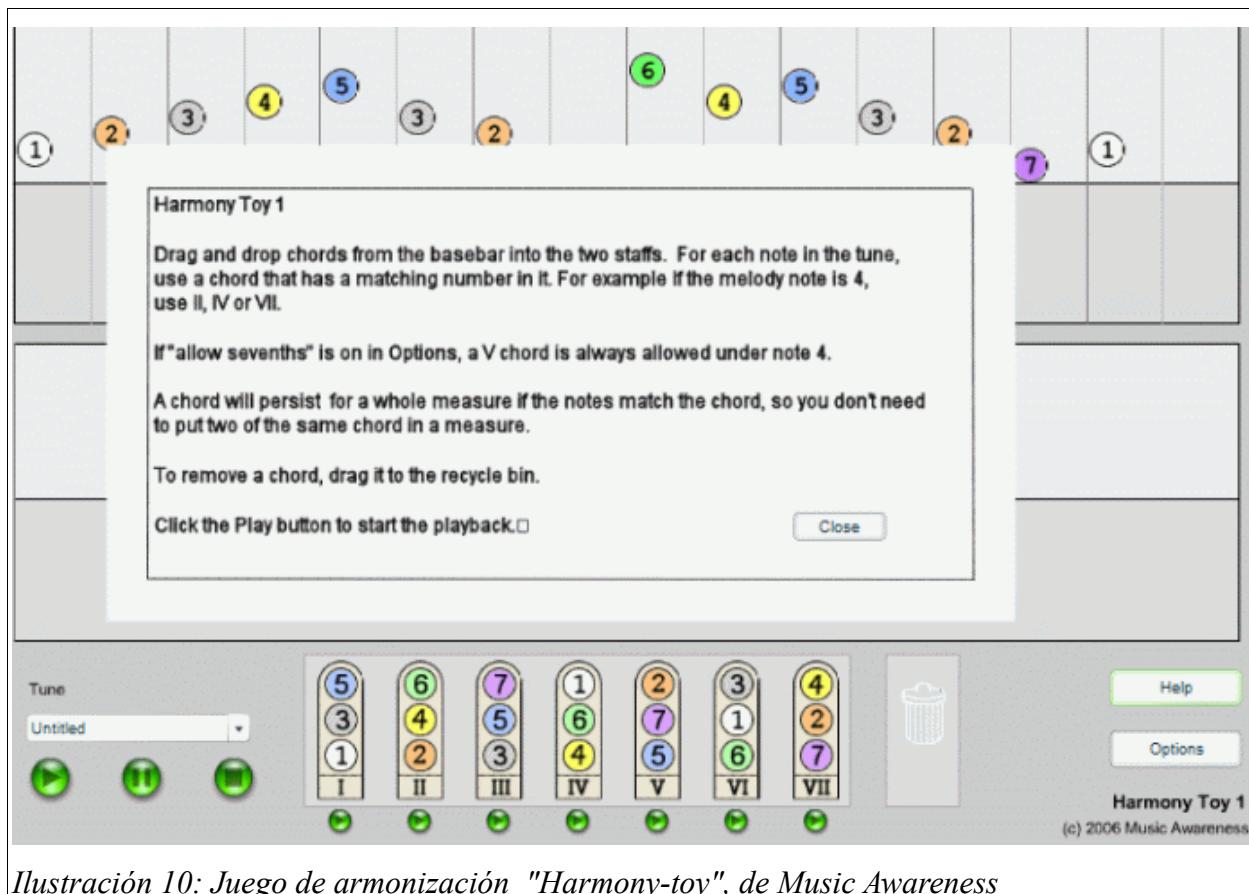


Ilustración 10: Juego de armonización "Harmony-toy", de Music Awareness

2.2.9.3.3 Curso de Michel Baron

URL: <http://michelbaron.phpnet.us/armonia.htm>

Autor: Michel Baron.

Idioma: Español y otros.

Completo y riguroso curso de armonía realizado por un profesional de gran prestigio y experiencia, Michel Baron, quien también ha colaborado en el excelente programa gratuito de aprendizaje de armonía *Harmony Practice 3*.

El curso es puramente teórico, pero muy completo y riguroso. Está destinado más bien a estudiantes algo avanzados. No se enseñan los fundamentos de la Música Tonal.

Combinado con el mencionado *Harmony Practice*, puede resultar un método muy adecuado y accesible para consolidar y mejorar los conocimientos de Armonía Tonal.

2.2.9.4 Aplicaciones de aprendizaje de armonía: Harmony Practice 3

En programas profesionales de composición musical como *Sibelius* o *Finale* existen módulos y componentes de ayuda al análisis armónico y a la armonización. También existen aplicaciones especializadas en ayudar a la composición de armonías musicales, como [Harmony Assistant](http://www.myriad-online.com/en/products/harmony.htm) (<http://www.myriad-online.com/en/products/harmony.htm>).

Existen aplicaciones desarrolladas para la enseñanza de la armonía, pero muchas de ellas son desarrollos experimentales realizados en entornos académicos y no accesible para el público en general. Es el caso de *Harmony Space*, de Simon Holland.

Pero programas pensados específicamente para la enseñanza de la Armonía Tonal, hay pocos, y entre ellos destaca claramente el *Harmony Practice 3* de Fausto Torre, realizado en colaboración con el profesor Michel Baron.

URL: <http://harmonypartice.altervista.org/HPWebSiteEn/index.html>

Programa gratuito pensado para *corregir* ejercicios de Armonía Tonal *a cuatro voces* según las reglas escolásticas clásicas. Fausto lo describe así:

"...is designed to allow music-students to learn and practice (write, listen to) four-parts harmony , according to classical treatises. Its main purpose is checking and explaining every possible error. HP3 is enough "intelligent" a software to be a rather "patient teacher".

La aplicación no orienta al alumno con teoría ni ejercicios predefinidos, sino que está enfocado a analizar y corregir los errores de ejercicio libres. No se trata de una utilidad pedagógica para *iniciar* en el Armonía Tonal, sino más bien una herramienta semiprofesional para ejercitarse la armonía en profundidad por parte de estudiantes avanzados.

A pesar de las limitaciones pedagógicas, esta utilidad destaca por su excelente y completa capacidad de configuración, por su rigor al analizar los acordes y por su elegante y eficaz interfaz visual.

Algunas características concretas de esta aplicación son las siguientes:

- Se detiene al encontrar un error.
- Tiene una utilidad denominada "*diccionario de acordes*" que permite elegir y escuchar cualquiera de los acordes posibles.
- Puede exportar solo a MIDI y a un formato propio binario y cerrado.

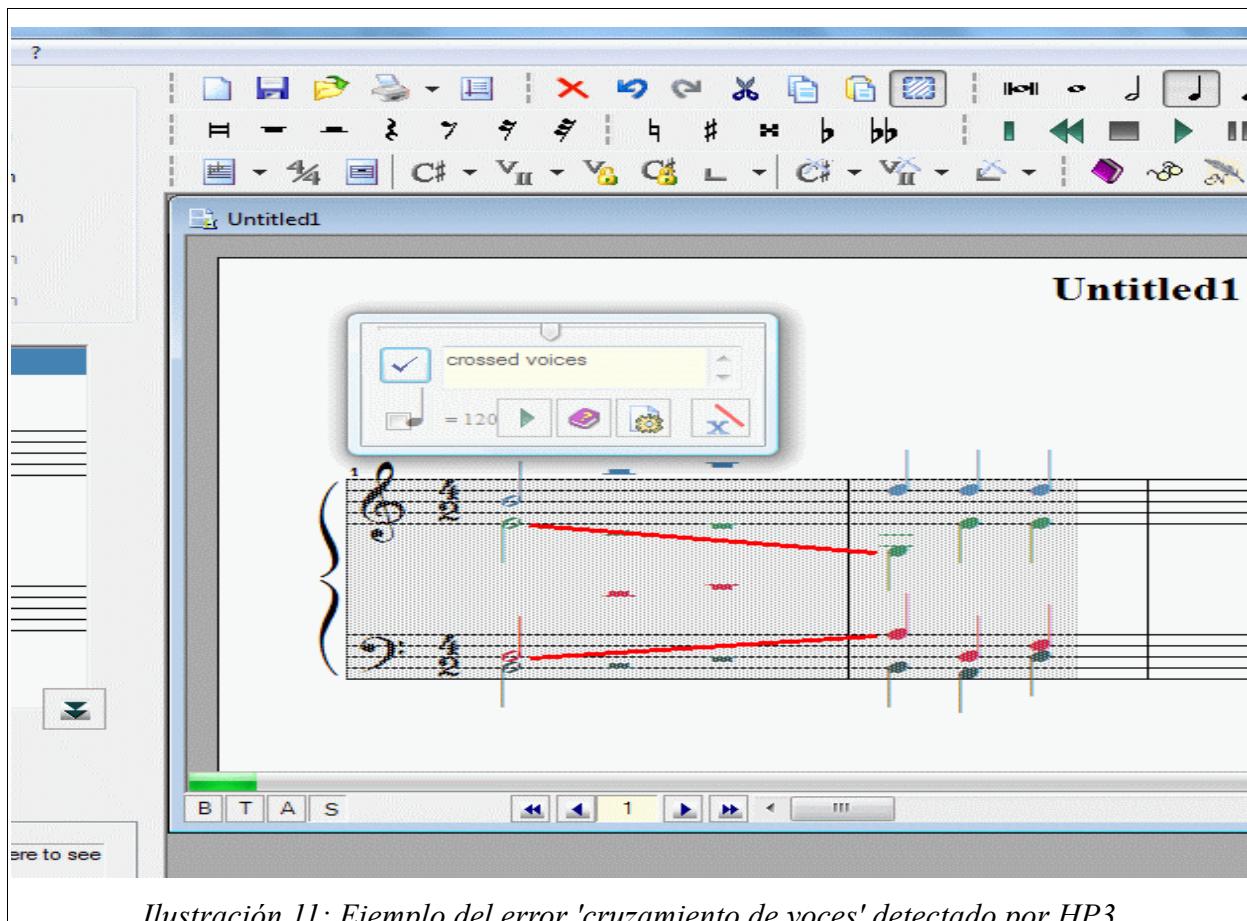
- En el ejercicio se elige: tonalidad, compás, número de voces (3 o 4) y tipo (realización de bajo o armonización de soprano).
- Dispone de un editor visual de partituras.
- Se puede elegir el tramo de la partitura que se quiere analizar.
- Se pueden elegir y configurar las reglas.

HP3 es un proyecto de mucha más envergadura que el Analizador Tonal desarrollado en este proyecto e integrado en *LenMus*. Sin embargo, ambos presentan un estilo y una funcionalidad similar: ambos son capaces de analizar la armonía de una partitura cualquiera, creada por el usuario directamente, y mostrar los resultados visualmente. Además, son capaces de reproducir la música representada por la partitura.

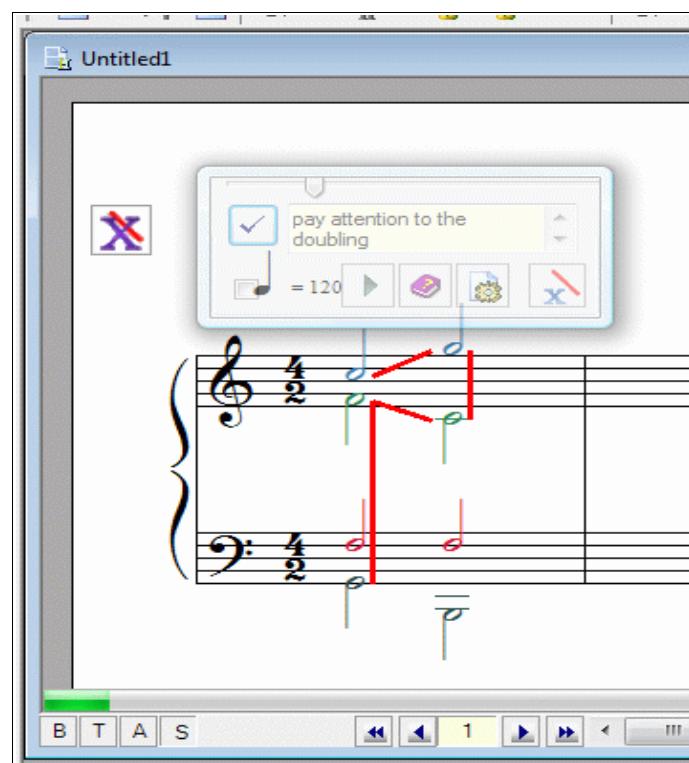
En cuanto a las diferencias, se pueden resumir en que HP3 es mucho más riguroso y exhaustivo, mientras que el Analizador de este trabajo se beneficia de la plataforma *LenMus* y aporta mucha más variedad funcional, como se describe a continuación.

Ventajas del Analizador Tonal de LenMus sobre HP3:

- | |
|--|
| <ul style="list-style-type: none"> • Se integra con teoría y otros tipos de ejercicios de armonía y de música en general, por lo que puede usarse por alumnos principiantes. • Crea ejercicios concretos y los corrige (puede <i>generar</i> acordes correctos). • Tiene Interoperatividad con otras notaciones musicales, como <i>MusicXML</i>. • Permite configurar el MIDI; por ejemplo, el <i>timbre</i> del sonido (instrumento). • Es capaz de mostrar a la vez, y dentro de la partitura, todos los mensajes de error . • Se puede guardar y recuperar la partitura, incluyendo los mensajes de error. • El editor de <i>LenMus</i> es mucho más completo. |
|--|



En estos ejemplos se observa el resultado de aplicar el analizador HP3 a una partitura. Los mensajes de error se van mostrando de uno en uno en una ventana, aunque las indicaciones sobre las notas, en forma de líneas rojas, se van superponiendo.



This screenshot is identical to the one above, showing the same musical score and harmonic analysis results. The message box now displays the text "inadequate preparation of the 7th" instead of "pay attention to the doubling". The red annotations highlighting harmonic errors remain the same.

Ilustración 12: Más ejemplos de errores de progresión armónica detectados por HP3

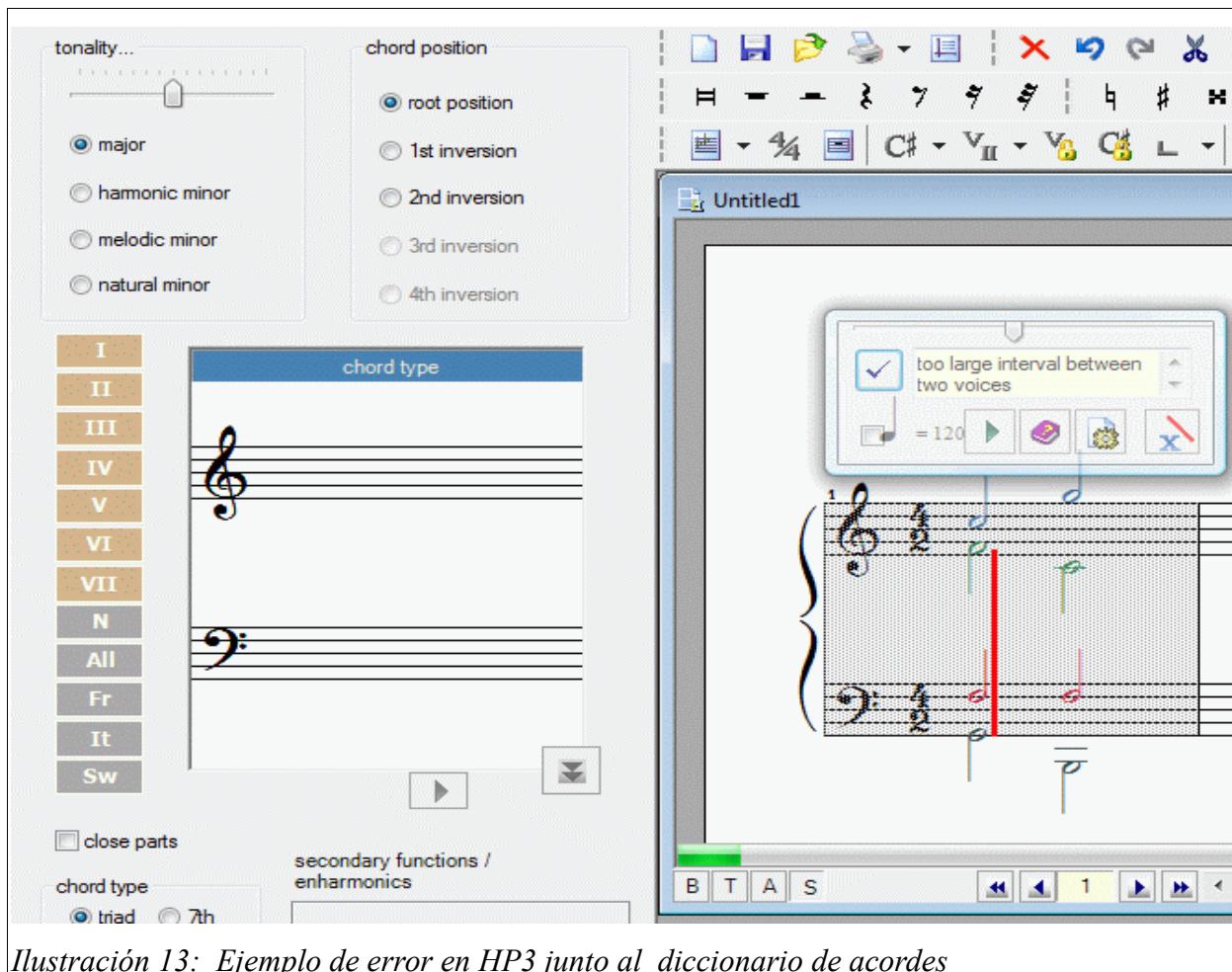


Ilustración 13: Ejemplo de error en HP3 junto al diccionario de acordes

En la imagen se observa una de las funcionalidades más destacadas del HP3: su diccionario de acordes, que permite ver y probar cualquier acorde.

2.2.9.5 Entornos heterogéneos integrados: i-Maestro

[i-Maestro](http://www.i-maestro.org/) (<http://www.i-maestro.org/>), acrónimo de "Interactive Multimedia Environment for Technology Enhanced Music Education and Creative Collaborative Composition and Performance", es un proyecto patrocinado por la Comisión Europea, que forma parte del 6º Programa Marco, dentro de la temática IST (Information Society Technologies).

Es una mezcla heterogénea de utilidades y prototipos prácticos con estudios teóricos, que incluyen el desarrollo de estándares. Todo ello con el objetivo final de promover la enseñanza de la música mediante la tecnología.

A continuación se describen algunos aspectos destacados de este interesante proyecto.

2.2.9.5.1 Objetivos

Los objetivos que se plantea son los siguientes:

- Explorar entornos multimedia interactivos para la educación musical con apoyo tecnológico.
- Integrar enseñanza musical y tecnología.
- Favorecer la accesibilidad.
- Promover el uso de estándares que favorezcan la educación musical, particularmente en un contexto eLearning (por ejemplo SO MPEG-SMR , MPEG4 y modelos de generación de tutoriales).
- Facilitar herramientas y utilidades para la creación de cursos eLearning flexibles y personalizables.
- Promover paradigmas pedagógicos de entornos de aprendizaje interactivo y cooperativo.
- Conseguir cursos que consigan despertar el interés y la motivación de los alumnos.

"The main technical objectives of the projects include: basic research and development on new solutions and enabling technologies to support traditional pedagogical paradigms for music training; novel pedagogical paradigms, such as cooperative-working, self-learning and class-studying, with particular focus on Symbolic Training paradigms and Practice Training paradigms for string instruments exploring interactive, gesture-based, and creative tools; and a framework for technology-enhanced music educational models and tools to support the creation of flexible and personalisable e-learning courses to improve accessibility to the musical knowledge. "

2.2.9.5.2 Actividades

Organiza anualmente "*i-Maestro Workshop on Technology Enhanced Music Education*":

(<http://www.i-maestro.org/workshop/>), donde se intercambian experiencias y trabajos teórico-prácticos en la línea de las nuevas tecnologías, accesibilidad y tendencias pedagógicas innovadoras aplicadas a la enseñanza de la música.

2.2.9.5.3 Herramientas y utilidades

i-maestro proporciona un conjunto variado de herramientas y utilidades :

(http://www.i-maestro.org/contenuti/contenuto.php?contenuto_id=52)

- *School server*: repositorio de contenidos pedagógicos.
- *The 3D Augmented Mirror* (AMIR) : utilidad multimedia interactiva de tipo *Realidad Aumentada* que captura las posturas al interpretar y ayuda a analizarlas.
- *Sound and Gesture Practice Training Tools*: conjunto de herramientas interactivas que permiten crear música a partir de gestos.
- *IMSym*: aplicación de apoyo al aprendizaje de la teoría musical, a base de preguntas respuestas.
- *Music Exercise Authoring Tool*: utilidad para crear cursos a medida, empaquetando teoría y ejercicios en un formato interoperable: el LPT. Los ejercicios son personalizables.
- *Cooperative Environment*: *middleware* para permitir realizar ejercicios de interpretación musical colaborativos.
- *SMR & MPEG*: notación multimedia especialmente pensada para dar soporte completo a los contenidos de educación musical.

2.2.9.6 Otros proyectos fin de carrera: "analizador de armonía musical"

Se da la curiosa circunstancia de que tras concluir el desarrollo del Analizador Tonal del presente trabajo, y al estar terminando la memoria, se descubrió un proyecto fin de carrera de características similares, en otra universidad.

Se trata del "*Analizador de armonía musical*" [AAM06], realizado por tres alumnos de Facultad de Informática de la universidad Complutense, bajo la dirección del profesor Jaime Sánchez. Disponible en <http://eprints.ucm.es/9035/> y <http://eprints.ucm.es/9035/1/TC2006-41.pdf>.

Aunque ambos proyectos parten de unos objetivos similares, toman rumbos radicalmente divergentes y consiguen logros bien distintos.

El hecho de haber descubierto ese proyecto una vez concluido el presente, da más interés a una comparativa de entre ambos.

El "*Analizador de armonía musical*" (AAM) pretende también detectar los acordes y analizarlos según las reglas de la armonía, mostrando los resultados en un editor. Aunque el punto de partida parece ser el mismo, ya desde el comienzo se diferencia en cuanto a utilidades, entorno y metodología.

El equipo del AAM se aventura a usar una entrada de datos en MIDI y a implementar la lógica de análisis en Prolog. MIDI es un lenguaje diseñado para representar los comandos para reproducir música, pero no para representar la música en sí misma. Prolog es un lenguaje de tipo declarativo pensado para la lógica, pero limitado fuera de ese contexto.

Pero el principal problema del AAM resultó ser la librería auxiliar para música: jMusic, que resultó adolecer de importantes defectos de implementación e incluso de diseño.

La mayor parte de la memoria el AAM está dedicada a relatar las continuos problemas que encuentran en su entorno y a las ingeniosas soluciones que consiguen idear para afrontarlos.

Finalmente logran el objetivo de reconocer perfectamente acordes individuales de cualquier tipo y en cualquier entorno de tonalidad, pero no se plantean analizar la evolución temporal de los acordes, o *progresión armónica*, a pesar de que es una parte esencial en el aprendizaje de la armonía.

Por contra, el Analizador Tonal del presente proyecto no se detiene en el análisis individual de acordes sino que implementa también el análisis de la *progresión armónica*, y además se adentra en

el terreno de la composición de armonías correctas, usándolas para plantear de varios tipos y también corregirlos .

A continuación detallan algunos aspectos concretos del AAM que le distinguen del presente Analizador Tonal:

- Encontraron enormes dificultades técnicas en todo lo relacionado con el procesamiento de la música y la visualización de la partitura.
- Gran parte de los problemas se debieron a los defectos de la librería jMusic
- Introducción de datos a partir de ficheros MIDI resultó problemática y se acabó recurriendo a la notación "ABC", una notación musical en texto ASCII.
- Encontraron problemas importantes hasta dar con un algoritmo para reconocer las cualidades básicas de los acordes (*Análisis Nominal*), en concreto hasta dar con el algoritmo de Larry Solomon.
- Problemas muy importantes por la dependencia de la tonalidad

En lo positivamente destacable de su analizador AAM:

- Consiguen detectar *modulaciones* tonales; es decir, cambios de tonalidad en medio de la partitura sin indicación expresa de los mismos. Esto no es sencillo y requiere de heurísticas, por lo que se adentra en el terreno de la Inteligencia Artificial
- Sin conocer la *armadura* consiguen deducir la *tonalidad* . De nuevo esto se encuentra ya en el campo de la IA, y por su complejidad, resulta bastante meritorio.

En lo *no-tan-positivo*, podemos decir lo siguiente:

- Su trabajo no es fácilmente adaptable ni reusable. Realizan un desarrollo muy específico para un contexto, y no facilitan la portabilidad
- Se complica innecesariamente el análisis, en gran parte debido a la *evitable* dependencia de la *tonalidad* y de la escala *diatónica*
- Se complica la entrada de datos (MIDI) por no tener una notación adecuada
- No se da a los *intervalos* la importancia que tienen en la armonía. De hecho no se menciona siquiera la palabra intervalo; solo se habla de "distancias", que es un concepto matemático,

pero no musical. Esto provoca que deban recurrir a algoritmos matemáticos rebuscados para detectar el tipo de los acordes, cuando existen algoritmos 'musicales', basados en los *intervalos*, más sencillos.

- No se llega a realizar ningún tipo de análisis de progresión armónica. Un analizador de armonía que no considera los enlaces entre los acordes solo sirve para los muy principiantes.
- No se crean estructuras de datos adecuadas para los acordes. En parte por las propias limitaciones expresivas de Prolog, pero también porque al representar los acordes no diferencian las cualidades *esenciales* (bajo, intervalos, inversiones) de las *accesorias* y derivadas o calculables (tipo, grado del acorde, nota fundamental, grado de las notas, cifrado, elisiones, etc.).

Para finalizar, tras observar los problemas principales encontrados en este trabajo y en el AAM, se puede hacer la siguientes reflexiones:

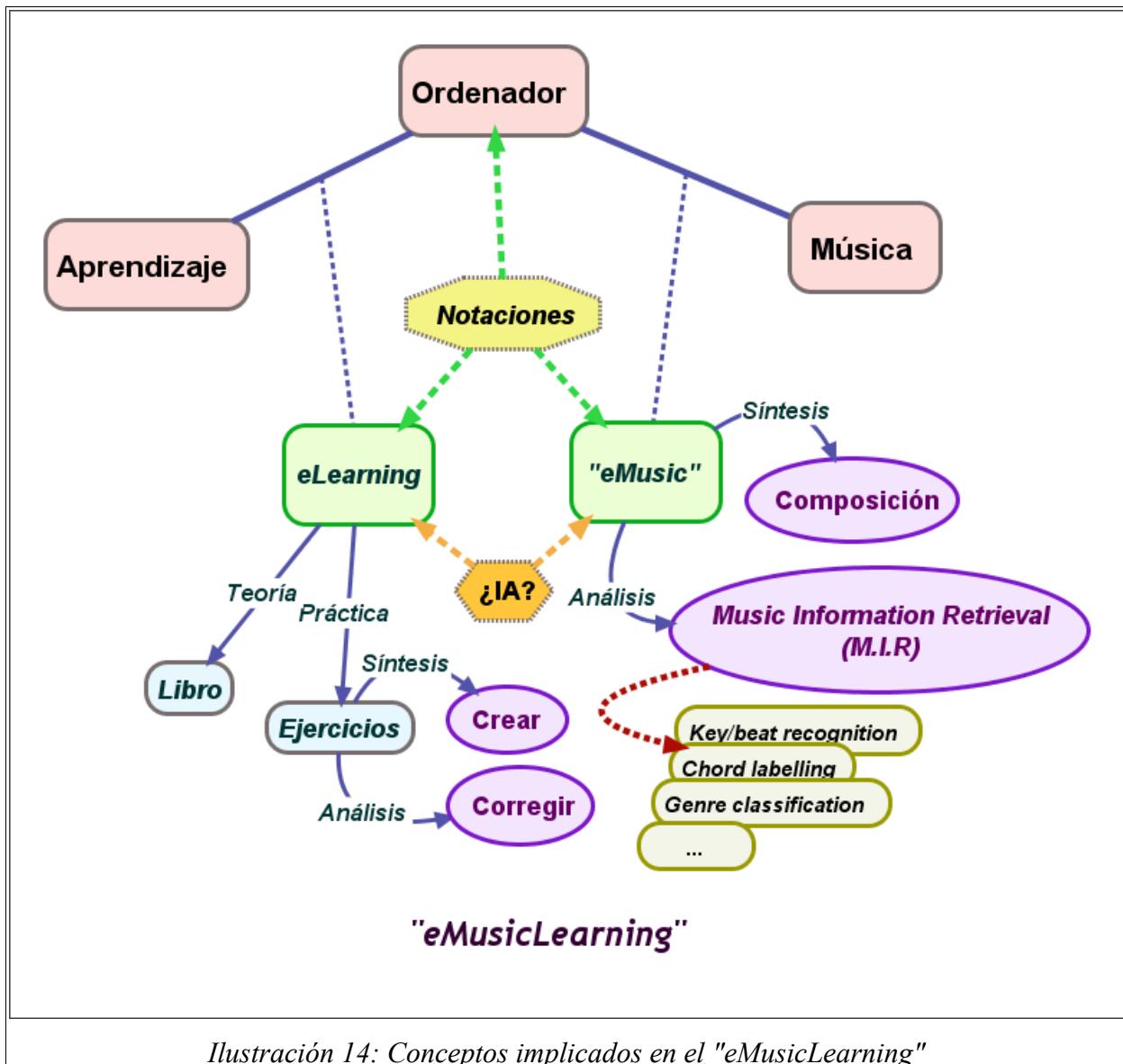
Desarrollar un analizador de armonía resulta más complicado de lo que inicialmente parece, principalmente por el hecho de que un acorde, que para los músicos es algo absolutamente preciso (un grupo de notas), es en realidad un ente que puede verse de maneras distintas, según su evolución o contexto, y por tanto, para poder representarlos y procesarlos mediante un lenguaje de programación, se precisa que este lenguaje tenga la suficiente *potencia expresiva* como para soportar este *polimorfismo* de los acordes. Lógicamente, si se necesita dar soporte al *polimorfismo*, conviene utilizar un lenguaje orientado a objetos.

Más a delante se explican los cinco niveles de abstracción, o *contextos*, de los acordes, y las propiedades que muestra en cada uno de ellos. No ser consciente de estos contextos, o no adecuarse a ellos hace que el procesamiento de los acordes resulte complejo y forzado.

2.3 Software educativo

2.3.1 Características deseables en los programas eMusicLearning

Diseñar un programa de aprendizaje musical exitoso representa un gran desafío, ya que, para resultar práctico y atractivo, deberá atender y responder adecuadamente a un variado conjunto de demandas y necesidades. Y no basta con destacar parcialmente en alguno de estos aspectos, sino que debe conseguir un equilibrio en todos ellos.



En cierto modo, muchos de estos rasgos buscados se podrían resumir diciendo que consisten en integrar correctamente instruccionalismo / constructivismo. Pero como este debate está demasiado

impregnado de subjetividad, intentaremos precisar las características concretas deseables en un programa eMusicLearning para que resulte eficaz y eficiente, es decir, exitoso.

2.3.1.1 *En cuanto a la funcionalidad*

Entre la funcionalidad deseable en un programa para educación musical, se pueden resaltar los siguientes aspectos:

2.3.1.1.1 *Amplio espectro de usuarios*

Debe ser útil tanto para profesores como para alumnos autodidactas. Y entre los alumnos, debe servir, tanto a alumnos avanzados, como para principiantes.

2.3.1.1.2 *Equilibrio teoría / práctica*

Es necesario combinar teoría y práctica en dosis *compatibles*, de forma que se consiga mantener la motivación, pero al la vez asegurar los fundamentos teóricos básicos.

2.3.1.1.3 *Equilibrio estático/dinámico*

Se debe permitir generar y también modificar con facilidad los contenidos teóricos y prácticos, con objeto de poder crear cursos 'a medida' y de adaptarse a la trayectoria de cada alumno.

2.3.1.1.4 *Creatividad*

Es muy importante que se complemente el curso con utilidades que permitan ejercitarse y desarrollar la creatividad del alumno. Particularmente importante es disponer de un editor de partituras suficientemente completo, y que, por supuesto, incluya la capacidad de reproducir sonoramente las partituras, de forma que el alumno pueda experimentar libremente en la composición de melodías, armonías, ritmos y demás elementos de la música.

2.3.1.1.5 *Seguimiento*

Es preciso recoger y, en la medida de lo posible, analizar información sobre el proceso de aprendizaje del alumno, para poder corregirlo o reorientarlo.

2.3.1.1.6 Orden, dosificación y guiado

Se debería conseguir que el proceso de aprendizaje siga unas etapas preestablecidas, al menos en la medida suficiente como para evitar que el alumno llegue a conclusiones teóricas incorrectas. Además, es deseable que el esfuerzo que requieren las etapas esté destruido de forma equilibrada, para que no termine desmotivando al alumno. También conviene guiarle convenientemente para evitar que se sienta desorientado, confundido o desmotivado.

2.3.1.2 En cuanto a la arquitectura y a la tecnología

Además de responder a los retos funcionales antes mencionados, un buen *software* educativo musical debe también cumplir con otros requisitos relacionados con la tecnología que utiliza. En este sentido, se pueden destacar los siguientes aspectos:

2.3.1.2.1 Transcripción automatizada

Imaginemos, por ejemplo, que para introducir los acordes que nuestro Analizador Tonal va a procesar, se permite no solo el teclado, sino cualquier instrumento. Esto abriría enormes posibilidades que sin duda volverían más atractiva y más eficaz nuestra herramienta educativa.

La tecnología actual ya permite transcribir automáticamente una entrada por MIDI, como lo demuestra Haruto Takeda en su ponencia "*Hidden Markov Model for Automatic Transcription of MIDI Signals*" [TAK03] presentada en el [ISMIR2003](#).

La *transcripción automatizada* permite enormes posibilidades de interacción entre aplicaciones musicales, ya que, una vez una vez se ha conseguido trasladar el sonido a una notación musical cualquiera, resulta muy simple transformarlo a otra notación.

Dentro de la intensa actividad investigadora en *análisis musical* (o *Music Information Retrieval*), este campo de la transcripción es particularmente activo, porque se están consiguiendo notables avances, y todo parece indicar que muy pronto será una realidad la transcripción automatizada en tiempo real, incluso con polifonía.

Si este sueño se hace realidad, puede suponer una inmensa revolución en el campo del *eMusicLearnig*, pues haría posible la corrección automatizada e instantánea de la interpretación que realiza el alumno con cualquier instrumento. La disponibilidad de este corrector infalible e incansable supondría que el tutor digital podría ya reemplazar al tutor humano en muchos aspectos, al menos para alumnos principiantes.

La tarea no es fácil en absoluto, incluso con una sola voz. Por ejemplo, es difícil distinguir dónde comienza exactamente una nota o cuándo termina, o diferenciar entre una nota larga y varias cortas seguidas. Sin embargo, ya se están obteniendo éxitos notables y parece que el reto más grande actualmente es el de conseguir un nivel de exactitud aceptable.

Algunos ejemplos de trabajos y productos en este campo:

- Matti Ryynänen "*Transcription of the Singing Melody in Polyphonic Music*" [Ryy06], presentado en el ISMIR2006 y "*Automatic Transcription of Melody, Bass Line, and Chords in Polyphonic Music*".
- Enlaces de software de transcripción automatizada:
<http://www.seventhstring.com/resources/transcription.html>
- [TS-AudioToMIDI](#) "is a program developed for automated music recognition. TS-AudioToMIDI is able to both recognize pre-recorded audio files and perform on-fly recognition"
- [IntelliScore](#) "is the only product in the world that helps you create music notation by converting multiple-instrument CD audio, WAV, MP3, WMA, AAC, and AIFF files to multi-track MIDI (.mid) files containing the notes played, broken down by instrument"

2.3.1.2.2 Interoperatividad

El término *interoperatividad* es el equivalente en nuestro idioma al inglés *interoperability* -se usa habitualmente el término *interoperable*, pero aún no figura en el diccionario de la R.A.E- y se refiere a la capacidad que poseen algunos sistemas *heterogéneos* de interactuar intercambiando información.

El concepto se puede resumir en tres palabras: "capacidad para entenderse", y en este sentido viene a ser un sinónimo de *comunicación*, pero dejando claro que se trata de comunicación eficaz, es decir, aquella que consigue transmitir información. Para ello es preciso un entendimiento en todos los niveles implicados en esa comunicación.

Aplicado al campo de las Tecnologías de la Información, la interoperatividad requiere disponer de unas interfaces estandarizadas y accesibles y, por supuesto, que estén implementadas correctamente a todos los niveles.

Se puede clasificar la interoperatividad en dos tipos, según la *latencia*, es decir, el tiempo que tarda el sistema en responder a un cambio en la entrada:

- *En tiempo real*: a la velocidad normal del sistema, sin necesidad de ralentizarlo o detenerlo. Es exigente, pues requiere una respuesta muy rápida a cada cambio en las entradas; normalmente se pide un tiempo de respuesta menor de 1/20 segundos. Esta interoperatividad es muy valiosa, ya que, por su rapidez, resulta *transparente* al usuario.
- *Desconectada* (en inglés: "off-line"): sin límite de tiempo en la respuesta. Por ejemplo, la capacidad de importar y exportar.

Cuando se trata de la informática aplicada al procesamiento de la música, la interoperatividad se traduce en entendimiento a varios niveles, y en cada uno de ellos, el entendimiento se basa principalmente en el uso de un *formato de archivo*.

2.3.1.2.2.1 Formatos de eMusic

Por un lado, está el nivel de formatos de *audio*, tales como MP3 y WAV, donde la capacidad de traducir un formato a otro es ya un tema *resuelto*.

A un nivel superior, a mitad de camino entre el audio y el Lenguaje Musical, está el formato MIDI, un protocolo creado para intercomunicar instrumentos musicales electrónicos con ordenadores. Este protocolo está ya enormemente consolidado y aceptado, lo cual representa una gran ventaja en términos de interoperatividad. Por ello, para cualquier aplicación musical es muy conveniente disponer de la capacidad de entenderse en formato MIDI.

A nivel de Lenguaje Musical, están los formatos o notaciones digitales para representar la Música Occidental. Se trata de formatos de ficheros de texto que están pensados para ser a la vez procesables por ordenador y entendibles por humanos. A este nivel no hay todavía ninguna notación claramente consolidada, pero algunas van cobrando fuerza, como el formato MusicXML, que parece el más universal en la actualidad.

Existen intentos de crear *metaformatos* que integren todos los niveles implicados en el procesamiento de la música. Estos formatos están aún poco difundidos y las aplicaciones que los soportan parecen estar aún en fase experimental. Es posible que su uso se extienda y que esto facilite la interoperatividad entre aplicaciones musicales, pero también es cierto que los metaformatos suelen dejar cierta ambigüedad y permitir implementaciones parciales, lo que conlleva riesgo de incompatibilidad.

En general, los MPEG se pueden considerar todos metaformatos para multimedia. Entre ellos podemos destacar el MPEG-4 SMR (Symbolic Music Representation), ya que está específicamente diseñado para integrar multimedia y notación musical.

El MPEG-7 es un ambicioso proyecto de crear un formato de descripción de contenidos multimedia. Sin embargo, no concreta ningún formato de codificación de los contenidos mismos, ni tampoco especifica una notación simbólica para la música.

Más detalles sobre los formatos digitales y las notaciones para la música se puede encontrar, por ejemplo, en la introducción al MPEG-4 SMR por parte de sus creadores: P. Bellini y P. Nesi en "MPEG SMR reference page. MPEG Symbolic Music Representation" (<http://www.interactivemusicnetwork.org/mpeg-ahg/>)

2.3.1.2.2.2 Formatos de eLearning

El concepto de "eLearning" se refiere a la creación y gestión de recursos educativos en formato digital. Existen muchos formatos y a muchos niveles. Para los recursos individuales se pueden utilizar formatos clásicos de textos y multimedia, como pdf, txt, hml, odf, mp3, avi, etc.

Pero el gran reto del eLearning tiene que ver con la reutilización de los recursos en distintos entornos. La interoperatividad en el mundo del eLearning es un concepto fundamental, y en torno a ella se han desarrollado numerosas iniciativas y esfuerzos, por lo que es un campo muy activo a nivel académico, comercial e institucional. Estos trabajos se dirigen principalmente a la estandarización de los metadatos de los recursos educativos. Para comprender mejor esta idea, conviene repasar algunos conceptos básicos del eLearning.

Conceptos básicos del eLearning

eLearning

Gestión de contenidos instructivos por medios electrónicos. En español se usa a veces el concepto *e-Aprendizaje* y el término "Aprendizaje Electrónico", pero por no estar muy consolidados aún, se suele utilizar el término en inglés, con o sin guión antes de la "L".

LO (Learning Objects)

Se refiere a recursos educativos *autocontenidos*, en el sentido de que pueden utilizarse en distintos contextos educativos, lo que permite reutilizarlos y combinarlos de diversas maneras.

Las propiedades de los LO se definen mediante *metadatos*. Para conseguir *interoperatividad*, estos metadatos deben estar *estandarizados*.

LOM (Learning Objects Metadata)

Los metadatos de los LO se organizan en grupos de estándares o "perfiles de aplicaciones" (en inglés: *application profiles*). Uno de los perfiles más conocidos es SCORM.

LMS (Learning management system)

Aplicación informática que gestiona recursos de aprendizaje. Los LO son gestionados por los LMS gracias a que entienden los LOM que los describen.

SCORM (Sharable Content Object Reference Model)

Es un conjunto de estándares para gestionar eLearning sobre la Web. Define tanto las comunicaciones entre el servidor LMS y el cliente del alumno, como el empaquetamiento de los contenidos. En SCORM, los LO se denominan SCO (*Sharable Content Object*).

Problemática de la interoperatividad en el eLearning

Para que el eLearning resulte *eficiente*, es decir, conveniente en términos de coste/resultados, se necesita poder reutilizar contenidos. La clave del éxito del eLearning es, por tanto, la interoperatividad, y la clave de ésta es la estandarización.

Pero la creación de estándares se enfrenta a intereses contrapuestos por parte de los distintos agentes que intervienen: empresas comerciales, países, instituciones de estandarización, instituciones educativas. Incluso también existe un conflicto motivado por los distintos modelos pedagógicos, principalmente por el permanente debate entre *Instruccionalismo* y *Constructivismo*.

Estos conflictos de intereses dificultan enormemente la convergencia de estándares, sin la cual, el avance en el uso y consolidación del eLearning no es posible.

Robby Robson, del *IEEE Learning Technology Standards Committee*, señala además otros problemas particulares para la estandarización del *Aprendizaje Electrónico*:

- Dificultad para distinguir la tecnología del aprendizaje de otras tecnologías dentro de la infraestructura empresarial.
- Estandarizar los metadatos de los recursos educativos es complicado por la enorme variedad de estos recursos.
- El reto no es 'gestionar' el aprendizaje, sino propiciar que el aprendizaje surja, creando un entorno adecuado.

Para profundizar más sobre este tema, se pueden consultar los estudios y trabajos disponibles en algunas de las instituciones dedicadas al eLearning, como las siguientes:

- LETSI: *The International Federation for Learning, Education, and Training Systems Interoperability* (<https://letsi.org/>)
- *Advanced Distributed Learning* (<http://www.adlnet.gov/>)
- *Interdisciplinary Journal of E-Learning and Learning Objects* (<http://ijklo.org/>)

2.3.1.2.3 Estructuración por niveles

Al diseñar una aplicación dedicada al procesamiento de la música, y en particular a la enseñanza de la música, conviene facilitar su adaptación a futuros cambios, y para ello es conveniente estructurar la arquitectura en forma de componentes modulares, organizados en niveles poco acoplados.

En el campo del que hablamos, todavía hay mucha variedad de estándares, tecnologías, librerías auxiliares, protocolos, etc. En el caso del aprendizaje, además, hay tener en cuenta que éste siempre tiene lugar en contextos muy variados, en función de las necesidades del alumno y profesor, o de otras muchas circunstancias que afectan al proceso de aprendizaje.

Conviene, por tanto, plantear un diseño abierto y adaptable a contextos variados. Esto es un tema muy estudiado y trabajado dentro del campo de la Ingeniería del Software, por lo que no se va

a entrar en detalles sobre cómo conseguir este objetivo, aunque no está de más destacar algunos consejos:

- Envolver las dependencias en interfaces bien delimitados.
- Estructurar la funcionalidad por niveles.
- Buscar en cada nivel una representación simbólica adecuada para las estructuras de datos manejadas.
- Facilitar la posibilidad de incorporar dinámicamente complementos o añadidos (*plug-in*).
- En general, evitar dependencias de protocolos, notaciones, librerías o formatos concretos, y especialmente si pueden quedarse obsoletos.

2.3.1.2.4 Extensibilidad

Ya se ha mencionado la conveniencia de que el software musical y educativo pueda adaptarse al entorno, lo que en terminología informática podría denominarse polimorfismo. Una variante muy interesante en los programas de amplio espectro, que pretendan aplicarse en entornos variados, es la capacidad de incorporar nueva funcionalidad sin necesidad de regenerar la aplicación de nuevo.

Permitir la extensibilidad requiere un esfuerzo adicional en el diseño, por lo que no siempre se tiene en cuenta. Pero es uno de los factores que se deben considerar a la hora de definir los requisitos de la aplicación, ya que puede ser un elemento clave para su éxito a medio y largo plazo.

Algunas formas de conseguir la extensibilidad es diseñar interfaces que permitan la incorporación dinámica de:

- Ejercicios y contenidos teóricos nuevos.
- Herramientas nuevas.
- Protocolos, formatos y estándares nuevos.

2.3.2 Las teorías de la educación ante las nuevas tecnologías

En las últimas décadas las teorías pedagógicas han experimentado una significativa evolución, debido, en buena medida, a la irrupción de las *Tecnologías de la Información*, que han abierto una

inmensidad de nuevas y variadas posibilidades de gestión de los contenidos educativos, lo que ha provocado efervescente debate sobre cómo debe entenderse la pedagogía en la *Era Digital*.

Se han cuestionado los modelos tradicionales de enseñanza de una forma tan radical que el debate ha transcendido el terreno de la *pedagogía* ("¿cómo aprender?") para llegar al de la *psicología cognitiva* ("¿cómo se crea el conocimiento?"), e incluso al de la *filosofía* ("¿en qué consiste el conocimiento?").

La trascendencia de los cambios tecnológicos en la sociedad parece justificar un replanteamiento serio de los modelos pedagógicos, sin embargo, el hecho de que los resultados no mejoran necesariamente con el incremento en el uso de la tecnología, ni tampoco con la aplicación de las modernas técnicas de enseñanza, hace pensar que la solución deberá basarse en una visión integradora.

Al diseñar un programa dedicado a la enseñanza por ordenador, conviene tener presente este debate y, por ello, a continuación se entrará en el tema con algo más de profundidad.

El uso de la tecnología informática como apoyo a la enseñanza de la música se ha planteado desde enfoques muy diversos. En general, en todas estas aproximaciones se pretende utilizar herramientas concretas para tareas específicas, pero apenas se encuentran intentos de crear entornos pedagógicos integrados y completos, que aborden todas las disciplinas involucradas en la pedagogía musical.

A continuación, se estudian los aspectos más relevantes del uso de tecnologías informáticas para el apoyo de la enseñanza de la música. Una importante fuente de información para este estudio ha sido el trabajo de Marcio Brandao *Computers in Music Education* [BRA99], donde se puede encontrar abundante información sobre este tema y una bibliografía muy completa.

Para conocer más en profundidad los sistemas de enseñanza en relación al uso de la Inteligencia Artificial, consultar el libro de Jesús G. Boticario y Elena Gaudioso "Sistemas Interactivos de Enseñanza/Aprendizaje" [BOT03].

2.3.2.1 Terminología pedagógica básica

Algunos términos habituales en la pedagogía interactiva.

CAI: "*Computer Assisted Instruction*", *Enseñanza Asistida por Ordenador*.

Cognitiva. Metáfora: libro. Material *estático*, independiente del alumno (*Enseñanza Programada*).

No hay lugar de técnicas de *Inteligencia Artificial*, ya que, según las cualidades descritas por J. Mira en [MIRA03].2, pag. 12:

- El conocimiento del dominio es suficientemente completo y representable simbólicamente.
- No se precisa aprendizaje en función de los resultados.
- El conocimiento es formalizable. No se usa conocimiento abstracto, de tipo "razonamiento de sentido común".

ITS: "*Intelligent Tutoring Systems*", Tutores Inteligentes.

Conexionista. Material dinámico, que se adapta según la respuesta del alumno.

El comportamiento no es determinista, *sino que se adapta en función de los resultados*. Para ello, mediante el uso de técnicas de *Inteligencia Artificial*, se implementan dos modelos: el *Modelo del Estudiante*, que estima lo que el alumno ha aprendido, y *Modelo del Pedagógico*, que decide cómo gestionar el proceso de aprendizaje (cap. 1.3 de J . G. Boticario en [BOT03]).

ILE : "*Interactive Learning Environments*"

El conocimiento no se adquiere por *transferencia* de un tutor, sino por *exploración* e investigación, gracias a las herramientas interactivas que permiten un alto grado de libertad al alumno.

AIEd : "*Artificial Intelligence and Education*"

Similar al ITS, con mayor énfasis en el uso de técnicas de *Inteligencia Artificial*

ICAI: "*Intelligent Computer Assisted Instruction*", *Enseñanza Asistida por Ordenador Inteligente*.

Similar al CAI, pero intentando añadir la *adaptación a la respuesta* del alumno.

2.3.2.2 Teorías del aprendizaje interactivo

La aplicación de las tecnologías de la información a la enseñanza ha ido evolucionando y diversificándose progresivamente, siguiendo modelos y aproximaciones pedagógicas diferentes y, hasta cierto punto, enfrentadas.

Por un lado, la *psicología cognitiva* estudia cómo adquirimos el conocimiento los humanos, y se concreta en los llamados *modelos cognitivos*. Por otro, la *pedagogía* estudia las teoría y técnicas de enseñanza y aprendizaje denominadas *estrategias instructivas*.

Sin entrar en detalles sobre los orígenes y las variantes de las diversas teorías y modelos cognitivos y pedagógicos, resulta interesante resumir las tendencias más relevantes, para poder comparar el software educativo desde el punto de vista de su modelo pedagógico.

Principalmente se distinguen dos tendencias contrapuestas. Una es la tradicional, basada en el concepto de que el tutor debe intentar transmitir al alumno unos conocimiento concretos, al estilo de los típicos libros de teoría y ejercicios. Es el *instrucionismo*.

La otra visión surge de forma paulatina y en su desarrollo tiene mucho que ver el uso de la tecnología, pues ésta, además de realizar la función de herramienta de apoyo en la transmisión del conocimiento, permite al alumno ejercitar y desarrollar su creatividad.

El *constructivismo* defiende que el conocimiento no se transmite, sino que se construye mediante la creación de interrelaciones multidimensionales. Por ello propone una forma de aprender basada en emular el comportamiento natural de la mente humana.

La aparición de la tecnología como un importante elemento de apoyo a la educación, ha representado un hecho clave que ha favorecido la visión **constructivista** de la pedagogía. Los modernos paradigmas educativos promueven tendencias como las siguientes:

- Enriquecimiento de contenidos multimedia y web.
- Adaptativos, *incluso se permite al alumno construirse el curso a medida*.
- Altamente interactivos; buscan captar el interés del alumno y lo motivarlo.
- Colaborativos: entornos virtuales que integran alumnos y profesores dispersos.
- Soporte tecnológico para autocorrección de ejercicios: realidad aumentada y similares
- Trazabilidad del historial del alumno para que el profesor controle y adapte.
- Integrados: que el alumno encuentre cerca todos los materiales que precisa.

Para comprender mejor este trascendental debate sobre las principales estrategias o perspectivas pedagógicas, resumimos algunos atributos básicos que las definen.

	Instrucionismo	Construcionismo
Modelo cognitivo	conexionismo	constructivismo
Metodología	estímulo/repuesta	exploración
Paradigma	pregunta/respuesta	aprendizaje <i>natural</i> , por descubrimiento
Metáfora	libro	juego
Instrucción	programada	por indagación
Conocimiento	objetivo y estático	ambiguo y dinámico
Objetivo	enseñar	aprender
Orientado a	el conocimiento	el proceso de aprendizaje
Actitud docente	conducir y guiar	orientar y apoyar
Ejemplos	CAI, ITS	ILE
Estructuración	jerárquica	colaborativa
Uso de la tecnología	como medio de transmisión	integrado y natural
Enfoque	preciso y cerrado	vago y holístico
Flujo de conocimiento	transmitir	adquirir por uno mismo

Tabla 4: Teorías y modelos del aprendizaje

Para conocer más sobre la visión construccionalista de la pedagogía y su relación con las nuevas tecnologías, se puede consultar el estudio de Seymour Papert "La máquina de los niños. Replantearse la educación en la era de los ordenadores" [SAY95] http://www.papert.org/articles/const_inst/

2.3.2.3 Estrategias instructivas y teorías del aprendizaje

Se pueden observar diferentes estrategias pedagógicas e instructivas en función de las teorías del aprendizaje en las que se basan.

Resulta útil resaltar algunas de ellas para poder caracterizar luego los programas de enseñanza musical.

Marcio Brandao realiza la siguiente clasificación en [BRA99]:

Instruccionalismo

Aprendizaje programado:

- Basado en las ideas de *Skinner*.
- Base del CAI: presentar material *estático*.
- Gestión de las respuestas simple: *comparación* con las esperadas.

Ejercicios y práctica:

- Repetición de secuencias de actividades hasta su asimilación.

Construccionalismo

Diálogo Socrático:

- Aprender por *descubrimiento*.
- El tutor *provoca* que el alumno descubra sus propias carencias y defectos.

Guiado y seguimiento:

- Se intenta que el alumno se *implique* en la tarea.
- Se sigue de muy de cerca su evolución con objeto de dirigirle hacia el óptimo rendimiento.

Exploratorio:

- Se promueve desarrollar la *inquietud indagatoria*.
- No se requiere un tutor cercano.

2.3.2.4 Debate instruccionalismo/constructivismo

Sin ánimo de entrar en polémicas pedagógicas que quedan fuera del alcance de este modesto proyecto, sí que nos podemos aventurar a decir que, al observar las características de los programas de aprendizaje musical, los más completos se caracterizan porque incorporan rasgos, tanto de la aproximación *instruccionalista*, como la *constructivista*.

En el ámbito pedagógico, como en tantos otros ámbitos, se puede decir que *la virtud está en el punto medio*. Dependiendo de cada necesidad u objetivo, se debe buscar el punto de equilibrio adecuado entre la aproximación *instruccionalista* y la *constructivista*. Ambas perspectivas tienen ventajas e inconvenientes, y ambas son necesarias. La primera establece el "*centro de gravedad*" del proceso de aprendizaje en el *profesor*, mientras que la segunda lo desplaza hacia el alumno. Si los contenidos teóricos son fuertes, se corre el riesgo de que el alumno se desmotive; si son demasiado flojos, quizás el alumno, por muy motivado que esté, no consiga distinguir lo importante.

El software de aprendizaje que solo sigue uno de esos caminos suele reducirse a trabajos experimentales y académicos. Para tener éxito comercialmente, debe resultar interesante y práctico para el alumno, y también debe responder a las exigencias del profesor, el cual espera que la formación sea completa y eficaz, y además desea poder realizar un seguimiento de los progresos del alumno para poder corregir y adaptar el plan didáctico en función de los resultados.

Por tanto, la mayoría de los programas de enseñanza musical, que son los analizados, intentan guiar al alumno para que adquiera un conocimiento concreto, en un progreso sistemático, pero también deja un cierto margen de libertad al alumno para que desarrolle su creatividad, explorando caminos variados, proporcionándole utilidades que le permitan crear, pero sintiéndose libre como si estuviese jugando.

2.3.2.5 LenMus en el contexto pedagógico

LenMus es un programa que integra ambas perspectivas, ya que, por una parte se plantea como un libro que contiene un plan ordenado para aprender contenidos teóricos y prácticos concretos, pero, por otro lado, facilita al alumno utilidades para explorar libremente su creatividad musical.

Dentro de LenMus, el *Analizador Tonal* desarrollado en este proyecto se situaría precisamente como *puente* entre ambos enfoques pedagógicos, pues funciona tanto en modo libre, corrigiendo cualquier composición en el editor, como en modo dirigido, planteando una serie concreta de ejercicios para habilidades específicas.

2.4 Perspectiva integradora

Antes de adentrarse en el análisis del problema concreto del analizador, conviene situar el problema en su contexto. Mejor habría que empezar por decir "contextos", ya que de entrada aparecen tres conceptos o *dominios* muy amplios, y a la vez muy relacionados entre sí actualmente: *aprendizaje, música y ordenador*.

Los tres han sido intensamente estudiados individualmente desde muchas perspectivas, pero también se ha discutido mucho sobre sus relaciones. Podemos destacar algunos temas clave de esas relaciones.

La relación *música-ordenador* es particularmente exitosa en muchos aspectos y, en especial, en temas relacionados con el *sonido* en general, como su almacenamiento, copia, corrección y reproducción digital.

Sobre esa especialización del sonido que representa la música propiamente dicha, con sus aspectos de ritmo, melodía y armonía, se encuentra también mucho interés por utilizar la potencia de procesamiento del ordenador para ayudar en todo tipo de tareas.

La música en nuestra sociedad se representa y se maneja mediante un lenguaje específico, con sus signos y sus reglas, creado a lo largo de siglos. Este lenguaje denominado *música occidental*, requiere un gran esfuerzo de aprendizaje, independientemente del esfuerzo necesario para dominar un instrumento musical. Por ello, el Lenguaje Musical queda apartado y lejano para una gran parte de la población que no dispone de tiempo y motivación suficientes para familiarizarse con él.

La digitalización ya ha acercado la música a la gente, pero principalmente en el sentido de hacerla más accesible, gracias a los formatos de audio digitales y a los medios de comunicación, intercambio y reproducción de contenidos musicales. Pero en este contexto, la informática solo se usa para tareas de bajo nivel, no relacionadas con el Lenguaje Musical.

Al adentrarse en el camino de aplicar la informática para procesar *música occidental* (en adelante, y para abreviar, usaremos el término **música** como sinónimo de *música en el Lenguaje Musical occidental*) surgen varias dificultades destacables, que mencionaremos a continuación.

Aprender música mediante la tecnología es un concepto que cada vez despierta más interés como objeto de investigación en el ámbito académico por los atractivos y variados desafíos técnicos. También despierta gran interés en el ámbito puramente comercial, ya que el proceso de aprendizaje musical requiere un largo esfuerzo en trabajo y en recursos (tiempo y dinero) por parte del alumno y, en consecuencia, si se consigue un programa que verdaderamente acorte el coste del

aprendizaje, por ejemplo, reemplazando en parte al tutor humano, sin duda, podría obtener un gran éxito comercial.

Aparte de las anteriores motivaciones, también existen justificaciones *sociológicas* para el interés por el *eMusicLearning*, en el sentido de hacer accesible a todo el mundo el *Lenguaje Musical* y, como consecuencia, también la música en general, y la llamada *Música Culta* en particular.

En este capítulo se analizan y consideran aspectos variados que conviene considerar al tratar con software musical en general, pero incidiendo en aquellos más relacionados con la tarea de diseñar e implementar un analizador de armonía tonal.

Al intentar aplicar el ordenador a campos como el *aprendizaje* y la *música*, lo que se pretende es modelar, es decir, hacer *computables* su conocimiento y sus mecanismos y reglas para conseguir imitar un rol o actividad concreto de estos dominios mediante el uso del ordenador. Al hacer esto, transformamos el aprendizaje en "aprendizaje computable" o *eLearning*, y la música en *eMusic*.

En el caso de este trabajo, lo que se pretende de alguna forma es modelar y emular mediante técnicas informáticas el rol del profesor o tutor en el contexto de la pedagogía musical. Se puede decir que se pretende trabajar con *eMusicLearning* desde la perspectiva del profesor.

Como el aprendizaje representa una actividad intrínsecamente humana, su modelado nunca es perfecto y solo se puede abordar como una aproximación. Por ello, es lógico que surjan muchas y variadas estrategias diversas para enfrentarse a esa tarea.

En este trabajo solo se intenta hacer un módulo muy concreto y definido de ayuda al aprendizaje y, por tanto, no es preciso ser conscientes de los distintos enfoques o paradigmas del aprendizaje, pero conviene citar por su trascendencia el debate permanente entre el *Instruccionalismo* y el *Construcionismo*.

Un interesante enfoque integrador entre instruccionalismo y construcción, usando la tecnología, lo da Singh Mukherjee en "*Collaborative learning and technology: Moving from Instructionism to Constructionism*" [MUK05]

3 Desarrollo

A continuación se describen los aspectos más relevantes del proceso de desarrollo de este trabajo.

3.1 LenMus

LenMus es una aplicación *eLearning* para el aprendizaje musical, versátil, flexible y variada. Se puede utilizar para crear y visualizar contenidos educativos tanto teóricos como prácticos en un formato de *libros* denominado *eBook*, basado en HTML, que permite integrar, y también personalizar, teoría y ejercicios.

LenMus dispone también de un completo editor gráfico de notación musical, capaz de representar casi cualquier partitura, y que permite, además, interpretarlas automáticamente mediante un secuenciador MIDI integrado, y salvarlas en un formato interoperable como MusicXML, o en una notación propia: LDP.

Pero LenMus puede usarse también como una plataforma o infraestructura para desarrollar aplicaciones de música, gracias a la librería de variadas utilidades que contiene.

Es importante destacar que LenMus es un proyecto completamente libre y abierto, y que se ha mantenido activo y en continua progresión y ampliación durante bastantes años, gracias al entusiasmo y a la dedicación de Cecilio Salmerón, su creador.

LenMus es un producto en constante evolución y mejora, que incorpora con frecuencia nuevas características, y que también experimenta a veces cambios en su estructura interna. Por ello la mejor forma de obtener documentación actualizada es acudir a la página de *documentación* del portal de LenMus: <http://www.lenmus.org/>.

Por los motivos arriba expuestos, se decidió evaluar la posibilidad de desarrollar el analizador tonal dentro de la *infraestructura* LenMus, con objeto de poder concentrar esfuerzos en lo verdaderamente importante, y de esta forma llegar lo más lejos posible en la consecución de los objetivos marcados.

Ante la entusiasta acogida de Cecilio al proyecto, y a su decidido compromiso a apoyarlo, se optó finalmente por desarrollar el analizador en el marco de LenMus.

3.1.1 LenMus como entorno de aprendizaje musical

LenMus es una plataforma que permite crear colecciones personalizadas de textos teóricos integrados con ejercicios interactivos de fundamentos musicales y entrenamiento auditivo, formando libros hipermedia empaquetados en un formato propio, basado en XML (*eMusicBook*).

Algunas de las características más relevantes de LenMus como herramienta de aprendizaje musical:

- Ejercicios interactivos
 - Pregunta/respuesta, con control de aciertos.
 - Entrenamiento auditivo (*aural training*) : reconocimiento de...
 - Claves, intervalos, acordes, escalas, tonalidades y cadencias
 - Adaptables mediante opciones y parámetros
 - Integrables en cualquier lugar dentro de los textos
 - Se pueden crear ejercicios nuevos, programándolos en C++
- Teoría musical
 - Estructurada en libros
 - Contenidos interactivos
- Lenguaje propio de XML para diseñar libros 'a medida' y generar contenido HTML
- Editor musical avanzado
 - Crea partituras en modo WYSIWYG
 - Interpreta las notas mediante MIDI
 - Completo: incorpora los elementos más importantes del Lenguaje Musical:
 - claves, tonalidades, notas, alteraciones, compases, ligaduras, signos, etc.
 - Voces de armonía *en cuatro partes*
 - Permite varios instrumentos (pentagramas)
 - Decisiones inteligentes:

- Dirección de plicas, asignación de voces, inserción de barras de compás
- Analizador de armonía
- Verificabilidad
 - exporta a fichero en formato propio (LDP) o estandarizado MusicXML
- Autónomo: no requiere conexión de Internet
- Multiplataforma (Linux, Win32)
- Internacionalización: soporte para múltiples idiomas
- Abierto: licencia GPL

Para conocer más detalles sobre LenMus: <http://www.lenmus.org/sw/page.php?pid=paginas&name=features>

3.1.2 Introducción a las funcionalidades de LenMus

Phonascus, en latín '*el profesor de música*', es un software para el aprendizaje del lenguaje musical, que se puede utilizar para mejorar las habilidades para leer partituras, el oído musical o, simplemente, para aprender los principios fundamentales del lenguaje y la teoría de la música.

- ***LenMus para la educación del oído***

La entonación y el dictado musical resultan difíciles. La identificación de intervalos, escalas y acordes es casi imposible de practicar sin un profesor que los ejecute al piano. LenMus Phonascus está siempre disponible para ello, evitando la necesidad del profesor y la del piano. LenMus incluye los siguientes ejercicios:

- Comparación de intervalos.
- Identificación de intervalos.
- Identificación de acordes.
- Identificación de escalas.
- Identificación de cadencias.

- ***LenMus para practicar conceptos de teoría***

LenMus permite concentrarse en aquellos aspectos que interese practicar. Los distintos ejercicios pueden personalizarse para amoldarse a las necesidades específicas. Se puede adaptar el ritmo de trabajo a lo deseado, recibiendo del programa evaluaciones, hasta alcanzar el grado de maestría deseado. Phonascus incluye ejercicios sobre:

- Identificación y construcción de armaduras.
- Identificación y construcción de intervalos.
- Identificación de escalas.
- Identificación de acordes.
- Identificación de cadencias.
- Asimilación y práctica de elementos rítmicos.
- Prácticas de solfeo: partituras compuestas por el programa; nunca se repiten.
- Aprendizaje de claves.

- ***LenMus incluye un editor de partituras***

El editor está en desarrollo y regularmente se mejora con nueva funcionalidad, pero la que ya ofrece puede resultar de gran utilidad para generar partituras con rapidez y facilidad.

El editor permite guardar y recuperar las partituras en varias notaciones musicales.

- ***LenMus soporta varios idiomas***

LenMus se ofrece con la posibilidad de elegir el idioma entre una variada gama de ellos.

Además ofrece la posibilidad de realizar traducciones a otros idiomas sin requerir conocimientos de programación.

- ***LenMus permite libros interactivos de música***

LenMus no es sólo una colección de ejercicios. La arquitectura permite integrar las partituras, los textos y los ejercicios. La idea es que los profesores puedan escribir libros interactivos de música, de forma que los ejercicios estén imbricados en el texto y las partituras no sean imágenes

pegadas, sino objetos con los que el estudiante pueda interactuar y escucharlas, en su totalidad o los compases elegidos.

- Cualquiera de los ejercicios disponibles puede incluirse en un libro (eBook) y sus parámetros pueden predefinirse.
- Los eBooks se escriben en XML, con la sintaxis de DocBook.
- ***LenMus es gratuito***

LenMus es un proyecto abierto, comprometido con los principios del *software libre*, la educación gratuita y el libre acceso a la información. No tiene objetivos comerciales. Por ello, se distribuye bajo la licencia [GNU General Public License](#), lo que significa que es gratuito y que el usuario tiene el derecho a acceder al código fuente, a copiarlo, modificarlo y dárselo a otros.

- ***LenMus es multi-plataforma***

LenMus está desarrollado pensando en que se pueda usar fácilmente en cualquier plataforma. Actualmente están disponibles las versiones Windows y Linux. El código fuente no tiene dependencias de plataformas concretas: está programado C++ con wxWidgets. Tampoco tiene requisitos especiales de hardware; cualquier equipo no demasiado viejo que tenga tarjeta de audio será suficiente.

- ***LenMus tiene otras características interesantes***

- **Para grupos en una clase**

- Opción global para realizar ejercicios en competición entre equipos. En estos casos los ejercicios muestran dos contadores, uno para cada equipo, y las preguntas van asignándose alternativamente a cada uno de los equipos.

- **Usabilidad**

- Al iniciar el programa, automáticamente comprueba si existe alguna nueva actualización.
- Las partituras pueden exportarse como imágenes bmp o jpg.

- El dibujo de las partituras se realiza con técnicas de *anti-aliasing*, lo que mejora considerablemente su aspecto visual en pantalla, facilitando la lectura y disminuyendo la fatiga visual.

3.1.3 El uso de LenMus

LenMus se utiliza principalmente de dos formas distintas: como editor de partituras con reproductor de sonido integrado y como libro educativo.

3.1.3.1 LenMus como editor de partituras

Como editor permite ejercer total libertad creativa. Es una herramienta auxiliar del aprendizaje muy útil para profesores y alumnos, pero también para aficionados a la música en general.

Las partituras se pueden salvar, recuperar y también reproducir si el equipo dispone de secuenciador MIDI, que es lo habitual hoy en día en cualquier PC.



Ilustración 15: LenMus como editor de partituras

3.1.3.2 LenMus como herramienta de eLearning

Como herramienta de aprendizaje interactivo, LenMus sigue la metáfora de libro. Las lecciones se organizan en capítulos donde se integran la teoría y los ejercicios.

- La gran ventaja de LenMus es la automatización de los ejercicios:
- Genera ejercicios sobre temas concretos, pero con infinitas variaciones al azar
- Los ejercicios son adaptables
- Corrige automáticamente los ejercicios
- Permite llevar estadísticas de los resultados

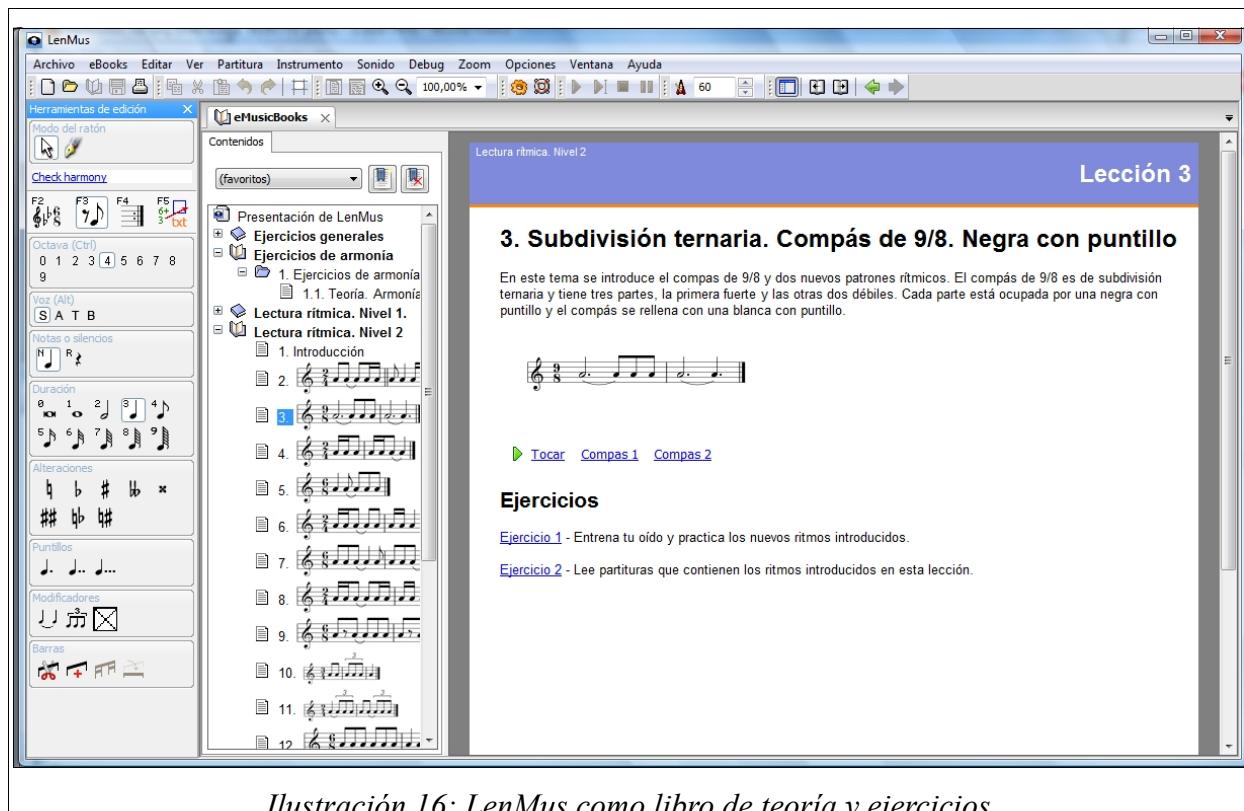


Ilustración 16: LenMus como libro de teoría y ejercicios

3.1.4 Integración del Analizador Tonal en LenMus

El Analizador de Armonía Tonal ha quedado perfectamente integrado en el entorno de aprendizaje musical LenMus, pasando a formar parte de su infraestructura como un componente más.

El Analizador puede aplicarse a cualquier partitura controlada por el editor; ya sea cargada desde fichero, creada por un generador de ejercicios o la usada directamente por el propio editor.

En la fecha de terminación de este proyecto, LenMus integra el analizador solo en una versión experimental basada en la variante de depuración (*debug*) del programa. Está previsto en los próximos meses que se incluya de forma definitiva en la versión oficial, una vez se hayan perfilado

los ejercicios de armonía finales y se hayan traducido los textos descriptivos de los mismos a los idiomas de LenMus.

El idioma por defecto de LenMus es el inglés y la traducción de los mensajes mostrados debe realizarse para cada uno de los idiomas soportados.

Se ha generado un instalable de LenMus específicamente pensado para este proyecto que incluye los ejercicios de armonía que están previstos en el proyecto y que son de tres tipos:

- Completar acordes conociendo el grado, la tonalidad y la nota del bajo
- Completar acordes conociendo el grado, la tonalidad y la nota del soprano
- Cifrar acordes mostrados

Además, en esta versión, el analizador también puede aplicarse a cualquier partitura manejada por el editor.

Para realizar los ejercicios de armonía, se abre el libro de teoría que viene con la aplicación y se elige el capítulo de ejercicios de armonía a cuatro voces. En el propio libro se detallan las instrucciones para realizar los tres tipos de ejercicios.

Cuando el editor tiene la partitura cargada, ya sea creada por el generador de ejercicios o de otra forma, para lanzar el analizador simplemente hay que elegir la opción "Check Harmony" del menú "Debug". Seguidamente aparecerán sobre la propia partitura las indicaciones de los errores detectados por el analizador.

En caso de no encontrar ningún error, se mostrará un mensaje indicando que la partitura es correcta.

3.1.4.1 Cómo usar el Analizador Tonal en LenMus

Como se ha mencionado anteriormente, hasta el momento en que aparezca la versión oficial de LenMus que incorpore el analizador y los ejercicios de armonía, se pueden utilizar versiones de desarrollo y depuración, que permiten aplicar el analizar cualquier partitura y también acceder al libro que incluye los tres tipos de ejercicios mencionados.

3.1.4.1.1 Instalación

La versión que incorpora el analizador es la rama RB-4.2.2 del repositorio de LenMus en Sourceforge. El usuario puede descargar el código fuente en la siguiente URL:

<https://lenmus.svn.sourceforge.net/svnroot/lenmus/branches/RB-4.2.2>

En el momento actual, la rama principal de LenMus (*trunk*) también incorpora la misma funcionalidad del analizador que en la rama RB-4.2.2. Si se usa la rama principal, se incorporarán las últimas mejoras generales de LenMus, entre las que son particularmente interesantes las mejoras del editor de partituras. Esta rama está disponible en :

<https://lenmus.svn.sourceforge.net/svnroot/lenmus/trunk>

Si se usa la rama principal, hay que descargarse el **libro** (*eMusicBook*, en terminología LenMus) con los ejercicios de armonía que está en: <https://lenmus.svn.sourceforge.net/svnroot/lenmus/branches/RB-4.2.2/locale/es/books/HarmonyExercises.lmb> y colocarlo en el directorio `/locale/es/books/`.

Para descargarla el código fuente se precisa un cliente Subversion como TortoiseSVN. Una vez descargado, se puede generar en los siguientes entornos:

- Windows Visual Studio C++ 2008
- Windows Visual Express C++ 2008
- Linux CodeBlocks + GCC

Las instrucciones para compilar el código fuente y generar los ejecutables están en el directorio `\docs\building`.

Está previsto también que en los próximos días se ponga a disposición pública en el repositorio LenMus un ejecutable que instale la versión con el Analizador Tonal.

Los instalables LenMus son muy fáciles de usar, pues basta con lanzarlos y responder a unas pocas preguntas sencillas.

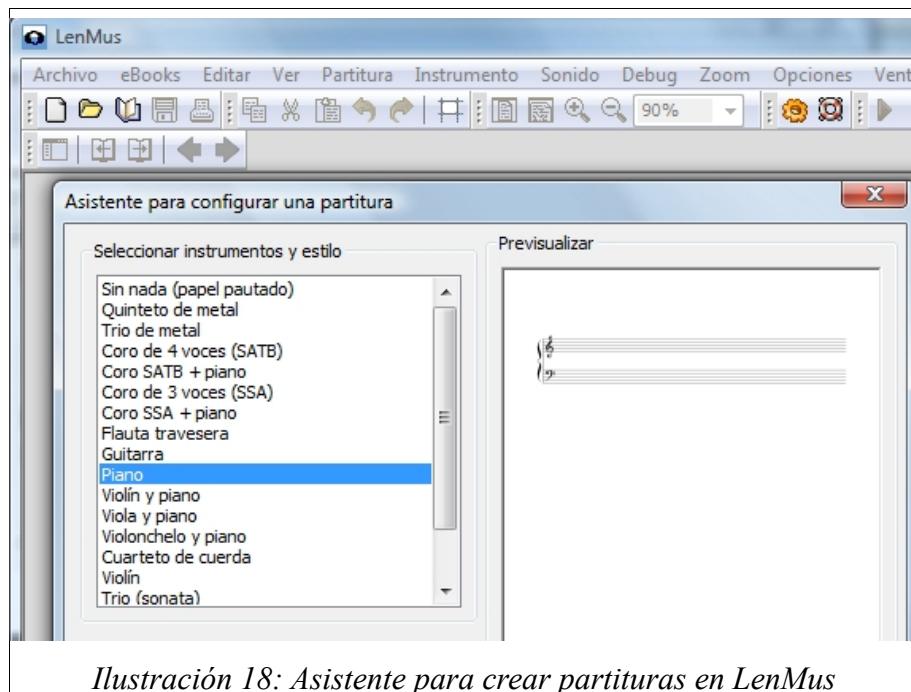
Utilizar un compilador para probar LenMus tiene la gran ventaja de que se puede utilizar la versión debug, con la cual se pueden observar paso a paso las instrucciones ejecutadas del Analizador mediante el depurador, y además muestra una traza de lo de los pasos más relevantes en la creación y el análisis de los acordes.

3.1.4.1.2 Uso del analizador tonal dentro de LenMus

Cuando se ejecuta LenMus, aparece un ventana desde la que se pueden crear partituras, *importar* partituras existentes o *abrir libros* de ejercicios. Estas operaciones permiten todas ellas utilizar el Analizador Tonal y se corresponden precisamente con los tres primeros iconos de la barra de herramientas. Veamos con más detalle cada una.

3.1.4.1.2.1 Analizando una partitura nueva en el editor

Este uso es muy simple: se pide a LenMus crear una partitura nueva, se responde a las preguntas del asistente sobre el tipo de partitura, y cuando aparece el editor se introducen las notas, usando el teclado o el ratón. Entonces se llama al analizador con la opción de menú "Debug" "Check harmony".



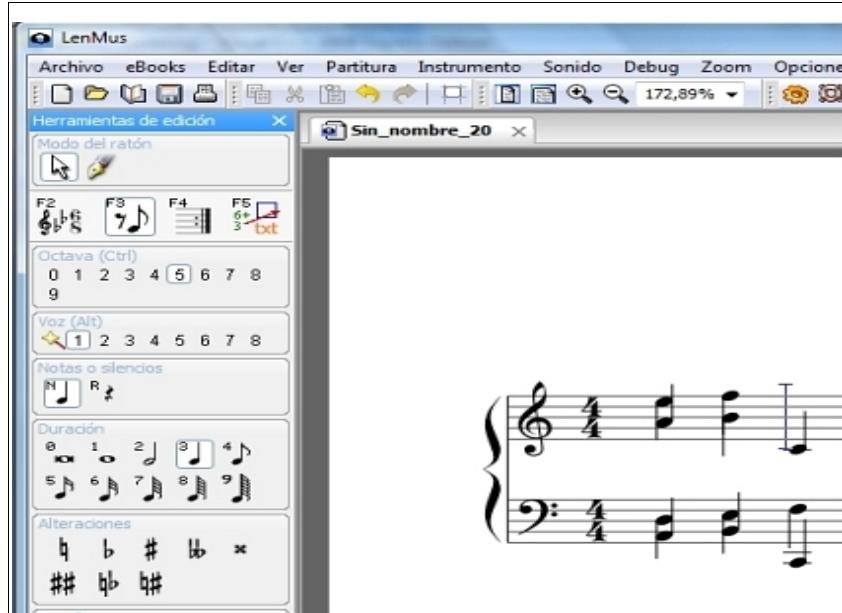


Ilustración 19: Aparece la partitura en el editor LenMus e introducimos las notas

 A screenshot of the LenMus software interface showing a musical score with analysis results. The score consists of two staves: treble and bass. Several annotations are overlaid on the music:

- A grey box at the top left says: "Chord 1: Interval p12 higher than octave between voices 2 (a4) and 3 (d3)".
- A pink box in the middle right says: "Direct movement resulting 5ª disminuida. Chords:1,2. Voices:1 e5→f5 and 2 a4→b4. Interval: d5".
- A red box at the bottom left says: "Chord 2: Interval p12 higher than octave between voices 2 (b4) and 3 (e3)".
- A red box in the bottom center says: "Bad chord 3: No reconocido; Notes: c4 f3 c2".
- A red box at the bottom right says: "Bad chord 2: No reconocido; Notes: f5 b4 e3 b2".

Ilustración 20: Resultado del analizador aplicado a partitura LenMus

3.1.4.1.2.2 Analizando partitura cargada

Este caso es similar al anterior; la diferencia es que en lugar de crear la partitura nueva se carga desde un fichero. Si este fichero tiene el formato LDP de LenMus se usa la opción de Menú "Archivo", "Abrir", mientras que si el formato es MusicXML se debe usar la opción "Importar". También se puede abrir una de las partituras recientes.

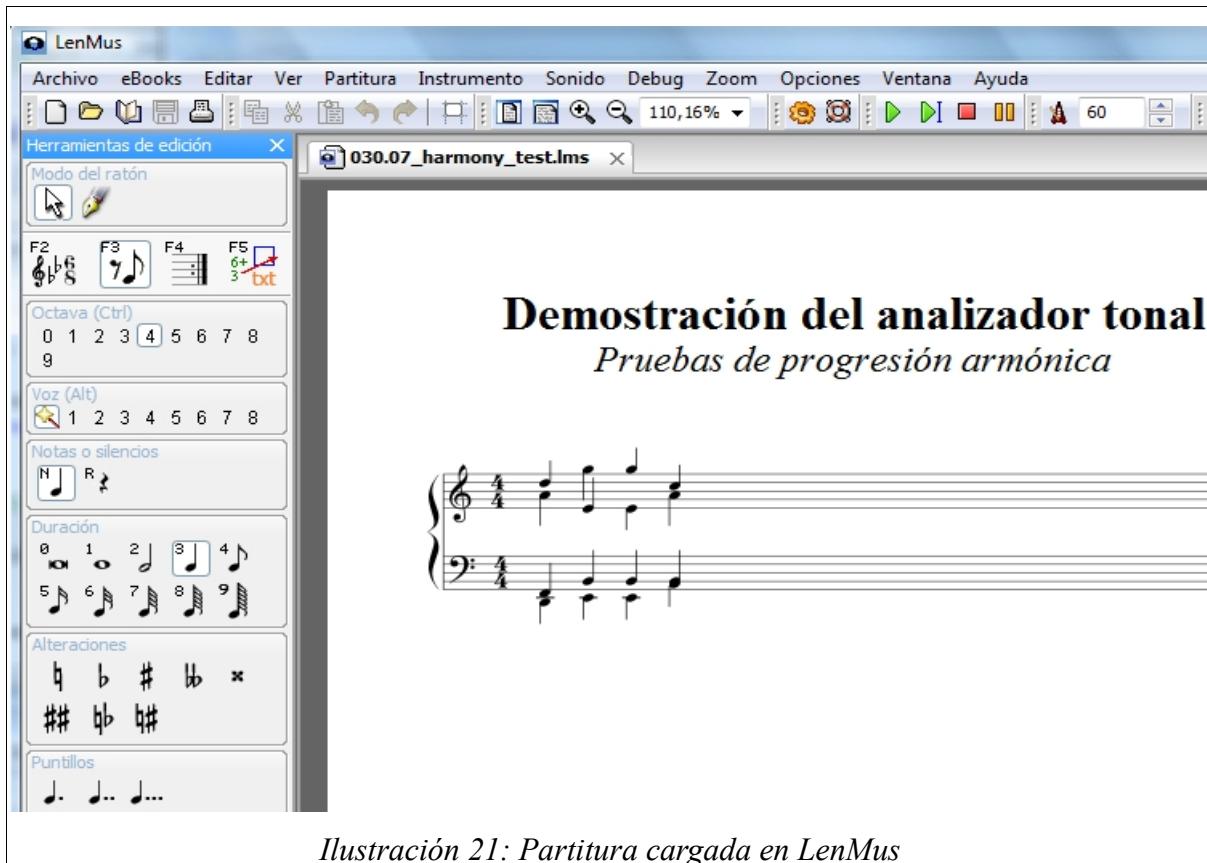


Ilustración 21: Partitura cargada en LenMus

Una vez cargada la partitura, se pueden utilizar las funciones disponibles en el editor para modificarla, guardarla o corregirla con el analizador.

Aplicamos el analizador a la partitura eligiendo en el menú "Debug" la opción "Check harmony". El analizador corrige la armonía, mostrando todos los errores detectados.

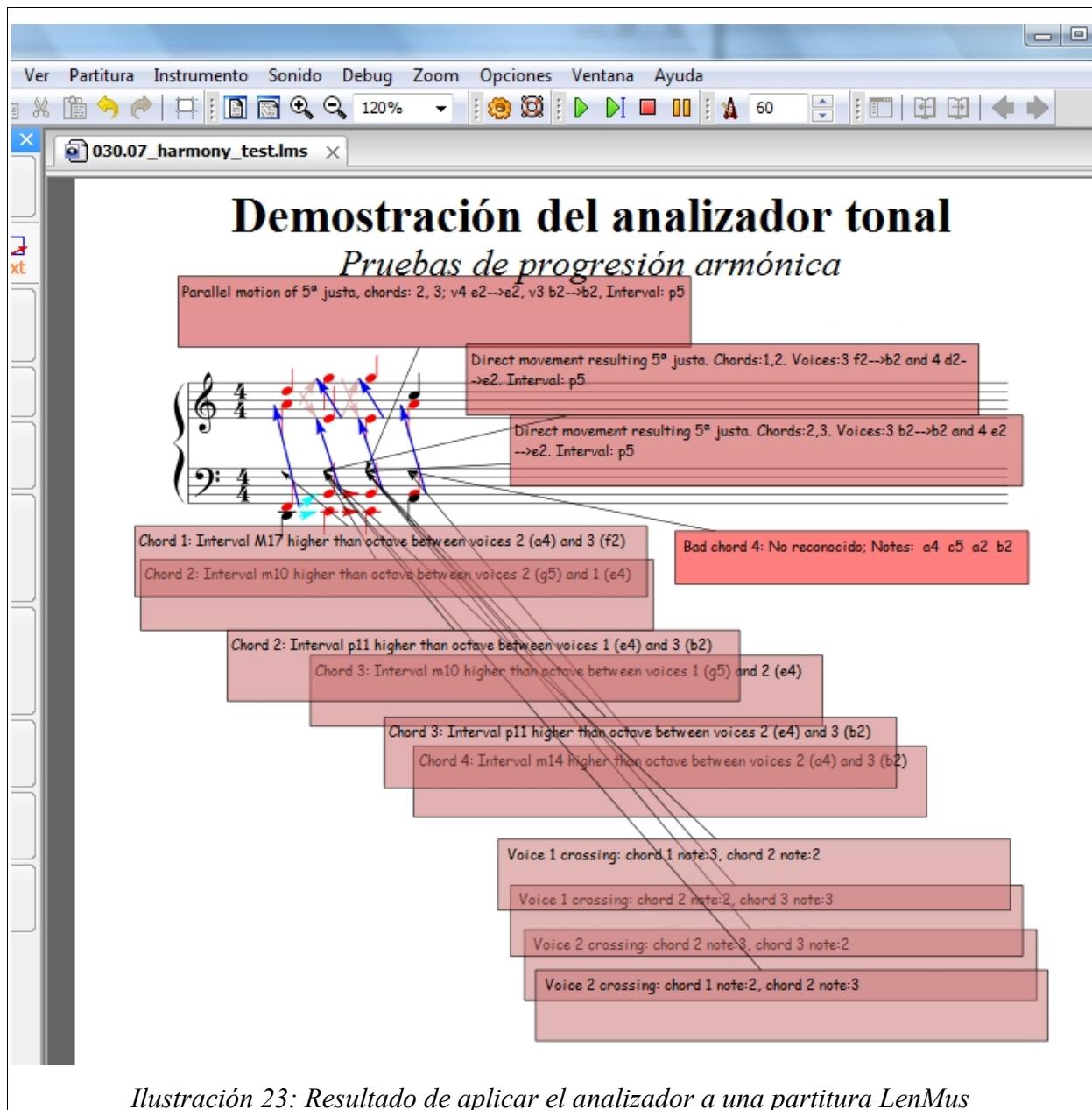


Ilustración 23: Resultado de aplicar el analizador a una partitura LenMus

En esta figura se puede observar el resultado de aplicar el analizador a una partitura de prueba, que contiene muchos y variados tipos de errores.

Los mensajes de error se muestran todos a la vez. Para facilitar su visibilidad, la aplicación intenta distribuirlos por la partitura y les aplica una transparencia. Cuando hay muchos mensajes, estos aparecen solapados, pero se pueden reubicar con el ratón.

3.1.4.1.2.3 Uso del analizador en los ejercicios de armonía

Para realizar los ejercicios de armonía se precisa abrir el libro de ejercicios de armonía, que como se ha indicado, lleva el nombre `HarmonyExercises.lmb` y debe estar situado en el directorio de los libros del idioma usado (`/locale/es/books/`).

El formato de empaquetamiento de libros en LenMus (extensión "`lmb`") es en realidad el formato "`zip`", por lo que se puede acceder al contenido del los libros con solo cambiar la extensión de `lmb` a `zip`. En el interior, los libros LenMus tienen ficheros HTML junto con un XML especial para describir el índice (extensión "`toc`").

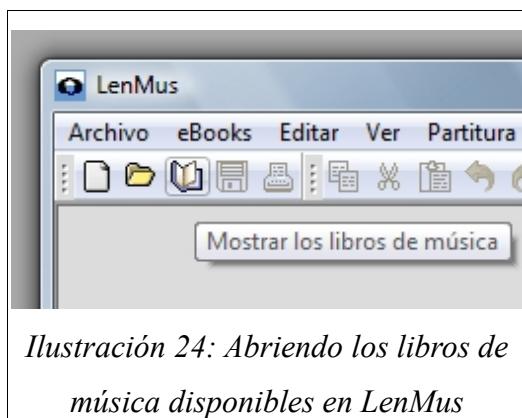


Ilustración 24: Abriendo los libros de música disponibles en LenMus

Tras pulsar el icono de "abrir libros de música" aparece el árbol de libros disponibles. Navegando por el libro de Ejercicios de armonía se accede a los ejercicios de *armonía a cuatro voces*, donde se describen los tres tipos de ejercicios y las instrucciones para realizarlos.

Aparece un enlace "Opciones", pero no está implementado aún.

Pulsando el enlace "Nuevo problema" se genera al azar un ejercicio de los tres tipos posibles.

Los ejercicios sirven para ejercitarse en habilidades básicas en el aprendizaje de la armonía a cuatro voces. En los ejercicios de tipo 1 y 2 se presenta una de las cuatro voces de cada acorde, y el alumno debe completar las tres voces que faltan, respetando el estado y el grado que se indican. En los ejercicios de tipo 3, los acordes se muestran completos y el alumno debe introducir el *bajo cifrado* asociado a cada acorde.

La forma de completar y corregir el ejercicio está pormenorizadamente detallada en el libro, aunque es muy simple, ya que consiste tan solo en introducir notas o cifrados.

Por el momento, el libro de armonía no contiene aún teoría. Está previsto que próximamente aparezca una versión que incorpore lecciones básicas de teoría de la armonía musical.

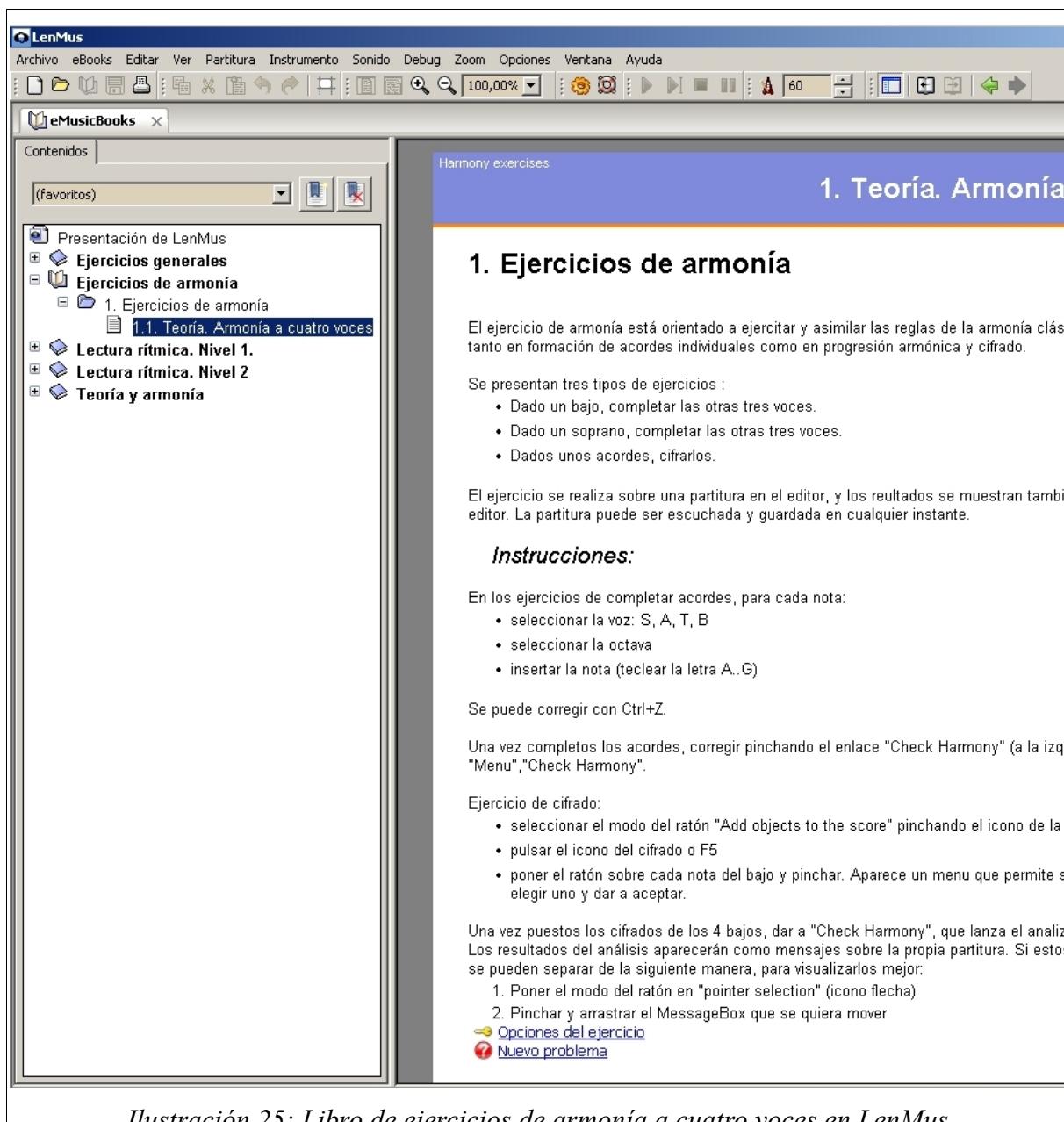


Ilustración 25: Libro de ejercicios de armonía a cuatro voces en LenMus

Cuando se crea un nuevo problema, al alumno le aparece la partitura creada por LenMus mediante el algoritmo de generación de armonías de este proyecto, sobre el que el alumno debe introducir las notas utilizando el editor.

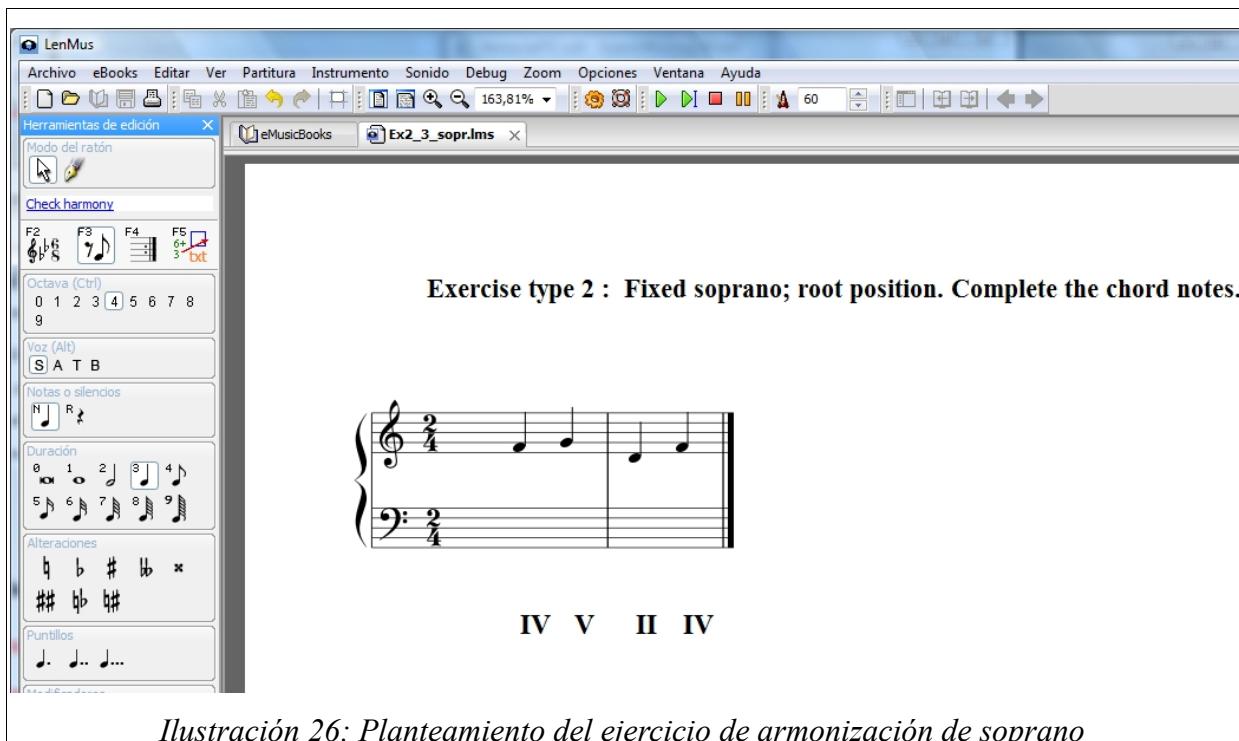
Siguiendo la filosofía de LenMus de crear siempre ejercicios distintos, el generador elige al azar el tipo de ejercicio, de entre los tres tipos posibles:

1. Completar armonización de soprano
2. Completar armonización de bajo

3. Cifrar una armonización dada

Los tipos 1 y 2 son muy similares. En ellos el generador crea una secuencia correcta de acordes, oculta tres de las cuatro voces, y el alumno debe completar las voces que faltan, respetando las restricciones de grado, soprano y estado del acorde.

La corrección de los ejercicios la realiza también el analizador tonal. Este tiene una primera parte genérica que siempre se ejecuta y que consiste en comprobar que los acordes tienen un tipo correcto y que se enlazan cumpliendo las reglas de la armonía. Pero además tiene un añadido que se activa solo para corregir los ejercicios del libro de armonía, y que se encarga de verificar que el alumno ha respetado las restricciones impuestas en el ejercicio, tales como grado, voz de referencia y estado del acorde.



En este ejemplo se muestra el planteamiento de un ejercicio de tipo 2, consistente en completar las voces, dado el soprano. El alumno completa los acordes utilizando las funcionalidades disponibles en el editor de partituras de LenMus. Se pueden introducir las notas fácilmente tanto con el teclado como con el ratón.

The screenshot shows a software window titled "eMusicBooks" with a file named "Ex2_3.lms". The menu bar includes "Editar", "Ver", "Partitura", "Instrumento", "Sonido", "Debug", "Zoom", "Opciones", "Ventana", and "Ayuda". The toolbar contains various icons for file operations and zoom. On the left, there's a vertical palette with icons for FB, 6, 7, 8, 9, and b*. The main area displays a musical score in 2/4 time. The top line (soprano) has notes G4, D4, G4, D4. The bottom line (bass) has notes B3, F3, B3, F3. Below the staff, the Roman numerals IV, V, II, IV are written. The title "Exercise type 2 : Fixed soprano; root position." is centered above the staff.

Ilustración 27: Ejercicio de armonización de soprano completado por el alumno

The screenshot shows a software window with a musical score in 2/4 time. The soprano line has notes G4, D4, G4, D4. The bass line has notes B3, F3, B3, F3. Below the staff, the Roman numerals IV, V, II, IV are written. A red box contains the text: "Direct movement resulting 5ª justa. Chords:2,3. Voices:1 g4→d4 and 3 b3→g3. Interval: p5". Another red box at the bottom right contains the text: "Bad chord 4: No reconocido; Notes: b3 g3 f4 f2". The title "Exercise type 2 : Fixed soprano; root position. Complete the chord notes." is centered above the staff.

Ilustración 28: Ejercicio de armonización de soprano corregido por el analizador

El ejemplo muestra que el analizador es capaz de reconocer acordes formados por notas de distinta duración, como en los acordes 2 y 4. También detecta el error de enlace entre los acordes 2 y 3 respecto a la regla de "no llegar a quinta por directo", así como un acorde incorrectamente formado.

Como se muestra, los mensajes aparecen con mezcla de idiomas español e inglés. Esto es debido a que la fase de traducción del inglés al resto de idiomas solo está aún parcialmente completada.

El Analizador detecta también otros errores tales como el incumplimiento del estado o del grado solicitado, o la violación de regla de "intervalos no superiores a octava", como se observa en el siguiente ejemplo.

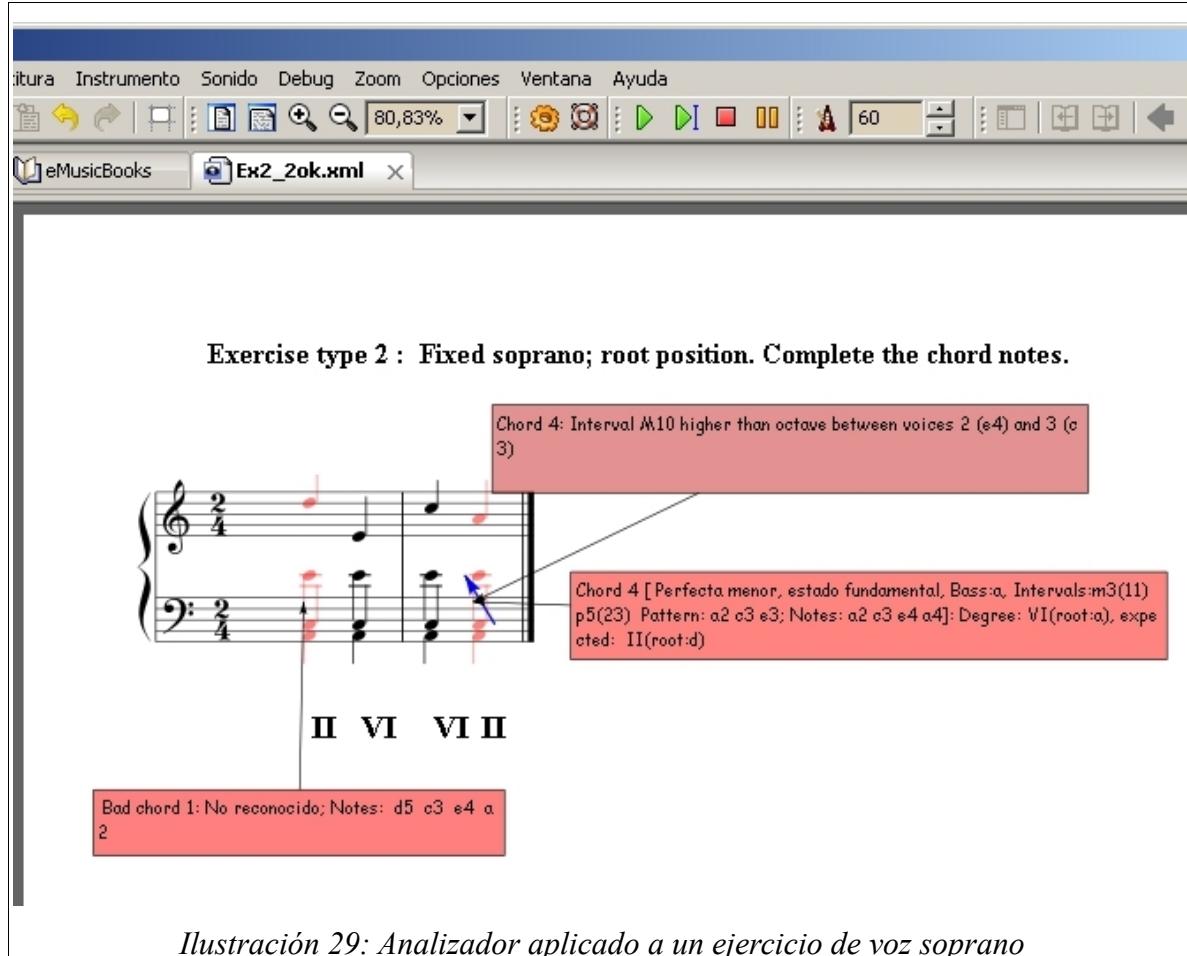


Ilustración 29: Analizador aplicado a un ejercicio de voz soprano

En la figura se aprecia cómo el corrector de ejercicios incorporado en el analizador indica al alumno que no ha respetado la restricción del grado II para el acorde 4. Además le muestra toda la información del acorde para ayudarle a comprender su error.

También se aprecia un acorde incorrectamente formado, y que por tanto no corresponde a ningún tipo reconocido.

Para el ejercicio de bajo cifrado, el alumno simplemente debe elegir la notación correspondiente a los acordes mostrados.

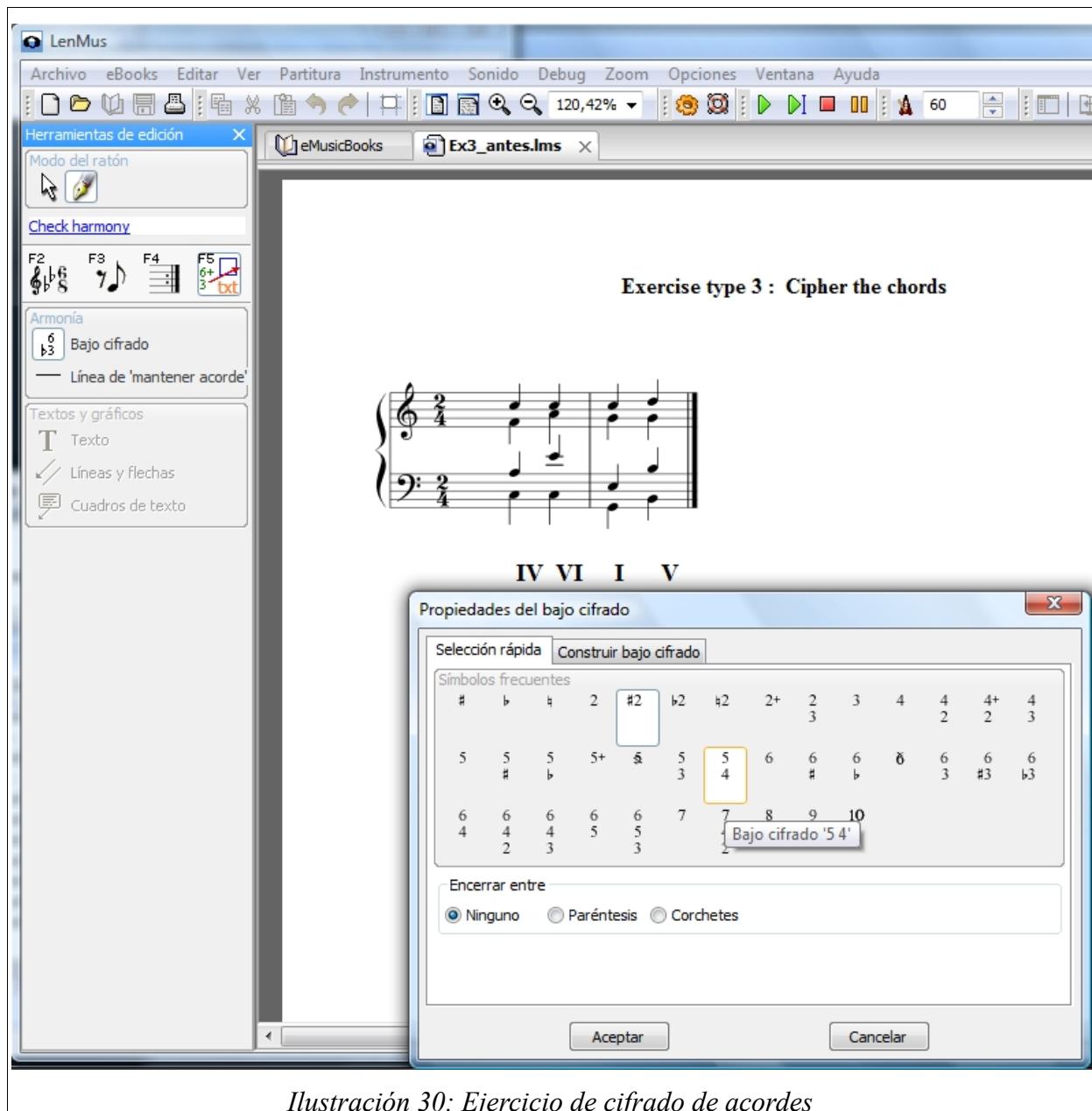


Ilustración 30: Ejercicio de cifrado de acordes

En el ejercicio de tipo 3 el alumno introduce el cifrado para cada acorde. Entonces llama al analizador, el cual compara los cifrados reales con los introducidos por el alumno e indica los errores correspondientes.

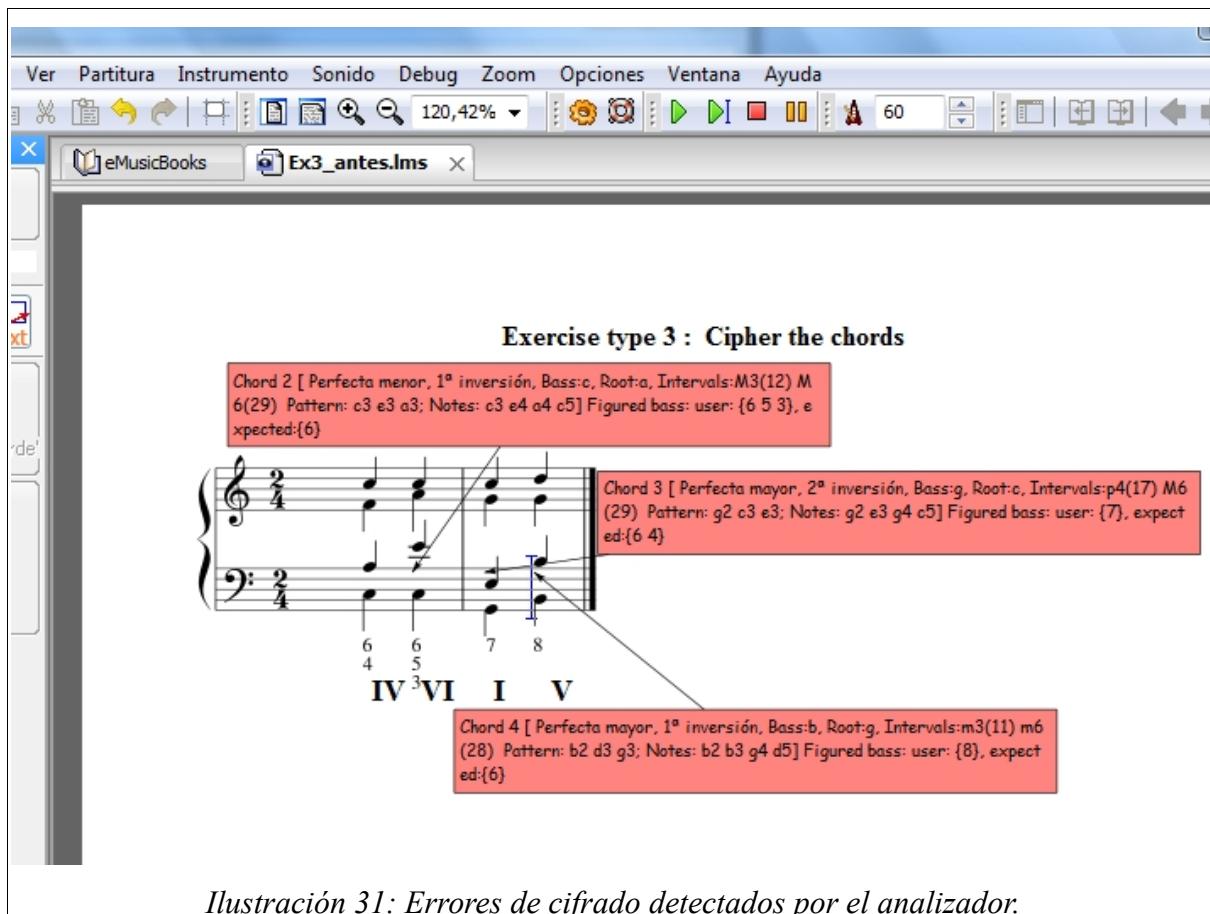


Ilustración 31: Errores de cifrado detectados por el analizador.

En este ejemplo se aprecia que el analizador ha detectado varios cifrados incorrectos. Se observa que para justificar su decisión y ayudar a comprender el error, el analizador muestra información detallada sobre cada acorde incorrecto, tal como el tipo del acorde y sus notas.

3.2 Metodología de desarrollo

A pesar del carácter de investigación experimental que tenía este proyecto, se intentó desde un principio implantar una disciplina de trabajo en forma de metodología, con objeto de gestionar de forma sistemática y rigurosa el avance.

Dadas las características tan particulares de proyecto, se buscó un tipo de metodología que permitiese el desarrollo *incremental*, por iteraciones sucesivas. También se tuvo en cuenta que, ante la imprevisibilidad de estimar con precisión el esfuerzo de cada etapa, previsiblemente se precisaría un replanteamiento frecuente de las prioridades y de los objetivos inmediatos.

La idea clave era avanzar paso a paso, replanteando los objetivos inmediatos tras cada avance, por ello resultaba evidente que se necesitaba una metodología *iterativa e incremental*.

También se precisaba la máxima *agilidad*, para poder adaptar cuanto antes el rumbo del desarrollo por el camino más adecuado, en función de los resultados y la problemática.

Se optó por intentar adaptar alguna de las metodologías *ágiles*.

3.2.1 Adaptación de las metodologías ágiles: SCRUM, XP, TDD

De entre las metodologías ágiles, se pensó en adaptar *Scrum*, por su adecuación a las características particulares del proyecto.

- El problema global es demasiado complejo como para diseñar una solución antes de codificar. Se requieren avances cortos y, tras cada iteración, un replanteamiento del rumbo, decidido entre todos los implicados. Las iteraciones deben ser cortas; de 2 o 3 semanas. Esto encaja muy bien con la gestión de los avances (*sprint*) de Scrum.
- Recursos limitados que hay que optimizar en función de las necesidades de los responsables del producto. Según *Scrum*, es el propio cliente quien tiene la responsabilidad de ir gestionando la relación *alcance / prioridad / recursos*.
- Los requisitos son muy genéricos, pero pueden concretarse en una lista de funcionalidades, de forma que en cada iteración se implemente un subconjunto de esa lista. La lista se asemeja mucho al concepto de 'pila de producto' (*product backlog*) de Scrum, la funcionalidad para cada iteración se corresponde con el *sprint backlog*. Como se ha mencionado, la asignación de elementos a implementar en cada sprint lo deciden todos los implicados, pero es el cliente quien tiene en definitiva la mayor capacidad de decidir.

- Los objetivos inmediatos de cada iteración los debía decidir el creador y responsable de LenMus, ya que solo él conocía el problema desde todos los puntos de vista, y por lo tanto solo él podía decidir el mejor camino para avanzar. Esto se asemeja mucho al rol del *cliente* en *Scrum*, quien debe implicarse continuamente y es el responsable final de marcar los objetivos para cada incremento.

Para conocer más en profundidad la metodología Scrum se puede consultar el libro de Ken Schwaber "*Agile Project Management with Scrum*" [SCH04].

Conviene advertir aquí que el término '*metodología de desarrollo*' no es el más adecuado para clasificar a *Scrum*, pues, según sus creadores, se trata más bien de una *metodología de gestión*.

A un nivel inferior al de la gestión, las metodologías ágiles proponen establecer una disciplina de trabajo mediante un conjunto de prácticas *saludables*. En concreto, la disciplina más recomendada en el ámbito ágil es la Extreme Programming, o XP, creada por Kent Beck.

XP consiste básicamente en un conjunto de 12 *prácticas* independientes entre sí, y que según Beck, resultan positivas para el desarrollo. Para conocer esas prácticas y su justificación, se puede consultar por ejemplo el libro de Beck "*XP explained*" [BECK03], pero mencionaremos al menos algunas de las más conocidas: *Integración Continua*, *Refactorización*, *Programación por Parejas*, y *Diseño Dirigido por Pruebas* (TDD).

La *práctica* del TDD ("*Test Driven Development*") se ha consolidado rápidamente y ha evolucionado por su cuenta hasta el punto de que ya se considera una metodología, o al menos una disciplina de desarrollo en sí misma.

En algunos aspectos se puede considerar a TDD como una metodología *revolucionaria*, pues supone un cambio *radical* respecto a los planteamientos tradicionales. Para entender mejor la implicación de su propuesta en el mundo del desarrollo del software, podemos recurrir a una equiparación con el cambio que supuso el *constructivismo* en la pedagogía.

El *constructivismo* rompe radicalmente con el pasado del instrucionismo, pues viene a decir que el método tradicional de enseñanza, basado en la *transmisión* del conocimiento, no sólo *no se debe* seguir usando, sino que *no se puede* seguir usando, ya que sencillamente "***no es posible transmitir el conocimiento***" y solo se puede *propiciar* que aparezca por medio de *experiencias* o *vivencias*. Es a lo que se refiere Simon Holland en "*Artificial Intelligence, Education and Music*" [HOL89] (<http://mcs.open.ac.uk/sh2/Simon%20Holland%20PhD.pdf>), cuando habla de la "*filosofía educacional de aprender sin ser enseñado*" (par. 2.1, pag. 15).

El planteamiento rupturista de TDD es de similar trascendencia que el del *constructivismo*, ya que plantea que no solo no se debe intentar *diseñar* o *modelar* la arquitectura de un sistema software, sino que en realidad **no se puede modelar**, ya que la arquitectura *emerge* a base de implementar pequeños ejemplos.

Se puede conocer más en detalle los preceptos del TDD en el libro de Carlos Blé Jurado "Diseño Ágil con TDD" [BLE10] (<http://www.dirigidoportests.com/el-libro>) , en el que aparecen citas tan llamativas como las siguientes:

"Modelar (traducir diagramas a código fuente) es peligroso y va en contra de lo que se explica en este libro"

"La única utilidad que tiene el UML es la de representar mediante un diagrama de clases, código fuente existente"

No entra en el alcance de este proyecto analizar las metodologías ágiles y mucho menos entrar en discusiones trascendentales sobre la conveniencia y la viabilidad del diseño, pero se ha mencionado todo esto sobre TDD por dos motivos principales: por un lado, TDD tiene claros aspectos positivos que se ha intentado aplicar en este proyecto, y por otro lado, el debate que plantea el TDD puede marcar la evolución de la ingeniería del software en los próximos años.

Dejando de lado todo aspecto filosófico, desde un punto de vista exclusivamente práctico, TDD plantea también un cambio muy relevante respecto a los métodos tradicionales: se debe probar antes que diseñar o implementar.

Lo que plantea TDD es que antes de comenzar en la implementación de un requisito debemos pensar la prueba que lo demostrará, e incluso debemos crear y ejecutar esa prueba que, por supuesto, inicialmente fallará. Después de fallar, comenzamos a desarrollar e implementar el código suficiente para que la prueba se pase con éxito, y entonces se termina el desarrollo, dando paso a una etapa de *refactorización* o rediseño.

La siguiente figura muestra el planteamiento del ciclo de desarrollo del software según el TDD.

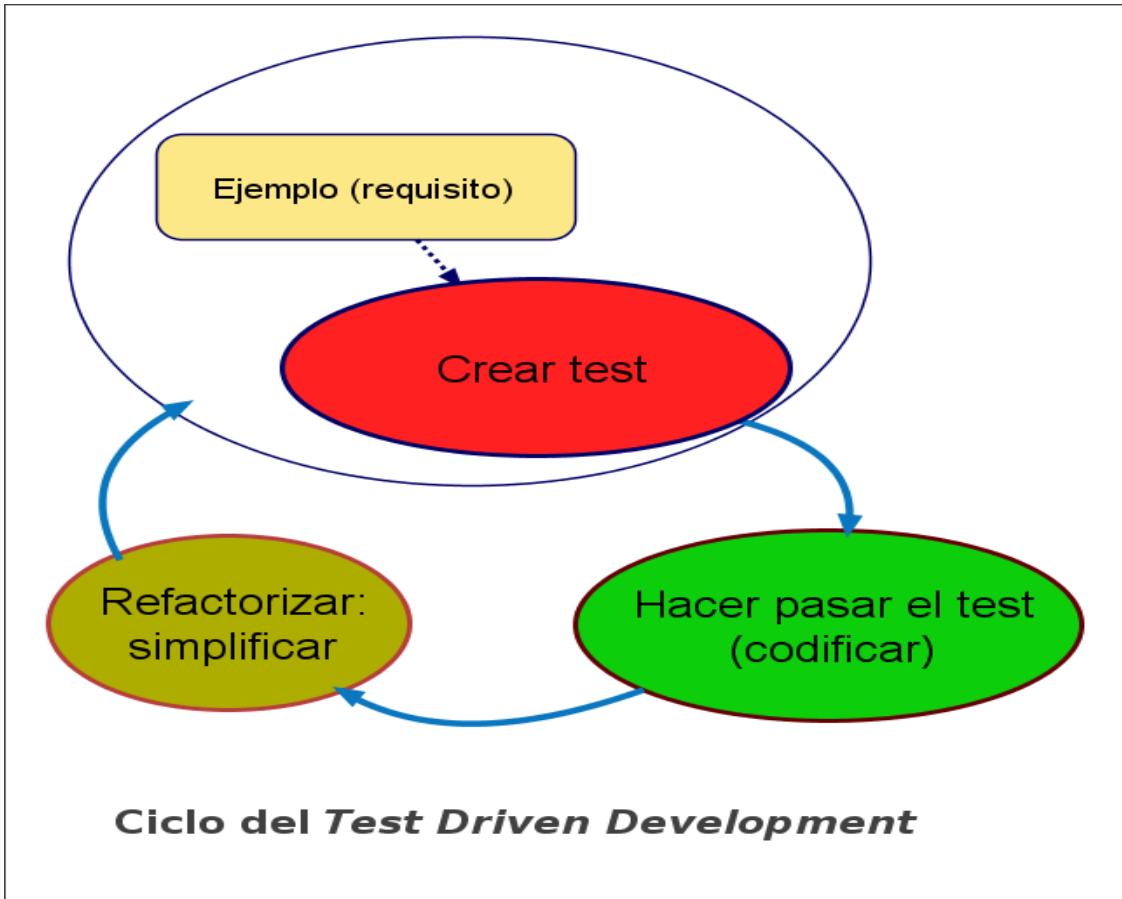


Ilustración 32: Ciclo del TDD (Test Driven Development)

3.2.2 Metodología mixta aplicada en el proyecto del Analizador Tonal

Los anteriormente explicados Scrum, XP y TDD se han intentado adaptar al contexto del presente proyecto de forma que se establezca una metodología, es decir, una forma sistemática de trabajar, que se puede resumir en los siguientes pasos.

1. Establecer un plan del proyecto definiendo las etapas del mismo, donde cada etapa se corresponde con un incremento de funcionalidad significativo.
2. Repetir el ciclo de desarrollo básico siguiente mientras se pueda (mientras no se alcancen todos los objetivos o se agoten los recursos dedicados)
 - 2.1. Acordar entre los implicados el siguiente hito funcional (objetivo) que se pretende conseguir.
 - 2.2. Acordar la forma de verificar que el objetivo se cumple (prueba)
 - 2.3. Esbozar un diseño del algoritmo para conseguir el objetivo (pseudocódigo)

2.4. Implementar el algoritmo

2.5. Probarlo

2.6. Si no se pasa la prueba, analizar el origen del problema y corregir código, y si es preciso, el algoritmo. Si se pasa la prueba exitosamente, continuar en el siguiente paso.

3. Rediseñar buscando simplificar y optimizar las estructuras y algoritmos (refactorizar).

Scrum establece unos roles y asigna unas responsabilidades a cada uno de ellos. Lógicamente, esta distribución de roles no es fácilmente extrapolable a un proyecto tan reducido como este, pero sí vale la pena destacar el "Responsable del Producto" (*Product Owner*), por su relevancia en Scrum.

El Responsable del Producto es quien tiene la visión de producto en su conjunto, quien establece los objetivos a largo plazo (*Product Backlog*) y a corto plazo (*Sprint Backlog*), y también es quien decide las prioridades y el valor de lo conseguido.

Este rol tan relevante y decisivo le correspondió, lógicamente, al responsable de LenMus, Cecilio Salmerón, quien asumió con decisión este rol, y supo guiar acertadamente este complicado proyecto hacia un final exitoso y provechoso.

Seguidamente describiremos algunos otros conceptos de Scrum que permitirán comprender mejor el desarrollo del proyecto en el contexto de esta metodología.

La *Pila de Producto* o *Product Backlog* es la lista global de necesidades (objetivos) priorizadas. A cada *necesidad* se le denomina *Historia del Usuario*.

Un Incremento o *Sprint* es un desarrollo 'atómico', en el sentido de que se buscan unos objetivos concretos y cercanos, que no se van a alterar con cambios de requisitos o de prioridades. Normalmente ocupa un tiempo de 3 a 5 semanas.

La clave del desarrollo con *Scrum* está en la gestión de los *Sprint*: decidir en función de las prioridades y de la capacidad de desarrollo del equipo, qué parte del *Product Backlog* se va a desarrollar en el siguiente *Sprint*, y concretar los objetivos específicos (*Sprint Backlog*). También se deciden al comienzo del Sprint los criterios de aceptación del producto resultante.

Tras cada *Sprint* se deben replantear las prioridades de las Historias de Usuario que faltan por implementar en el *Product Backlog*. También es importante aplicar una visión crítica a lo desarrollado hasta el momento y rediseñar o refinar ("refactorizar") lo que sea preciso para conseguir que el producto se mantenga robusto, elegante y simple.

Si se precisa realizar una refactorización a gran escala, se puede plantear como un Sprint específico para ese propósito.

Se han utilizado *Sprint* de aproximadamente 2 o 3 semanas de duración.

3.2.3 Plan de trabajo

Como ya se ha explicado, el *Product Backlog* en *Scrum* es una lista de necesidades priorizada. Es la base para decidir qué hay que hacer y en qué orden. A estas necesidades se les denomina *Historias de Usuario*, porque el usuario (normalmente el *Product Owner*) las describe informalmente, a modo de ejemplos de funcionamiento; algo similar a los *Casos de Uso*, pero muy simples.

Al comienzo del desarrollo de este Analizador Tonal, los implicados en el mismo mantuvimos una reunión para acordar los objetivos básicos y también establecer unos roles.

Como se ha comentado, el rol principal, el equivalente al *Product Owner*, recayó sobre el responsable de LenMus, quien por sus conocimientos informáticos, musicales y, sobre todo, del propio LenMus, era claramente quien debía desempeñar ese papel. Su misión inicial fue establecer un plan de trabajo que marcase las etapas del desarrollo del proyecto, junto con la funcionalidad asociada a cada una. Cada etapa se correspondía con un incremento significativo en la funcionalidad. Además, estos incrementos eran progresivos y secuenciales, pues para conseguir uno, en general se necesitaban los anteriores, por lo que la prioridad quedaba ya implícita en el orden dentro del plan.

Así pues, nuestra metodología de avance consistía en un proceso cíclico de extraer el siguiente requisito del plan de trabajo, analizarlo y desglosarlo en *incrementos* concretos y relativamente cortos de funcionalidad, y para cada incremento, pensar en cómo se probaría su corrección y lanzarnos a implementarlo.

De todo ello surgió la lista detallada de actividades a desarrollar, o *Historias de Usuario*, que se muestra a continuación:

1. Familiarización con LenMus
 - 1.1. Crear/modificar capítulos
 - 1.2. Añadir ejercicio
 - 1.3. Crear '*controles*' para los ejercicios

- 1.4. Crear '*procesadores*' para los ejercicios de partitura completa
2. Análisis de acordes individuales
 - 2.1. Detectar notas simultáneas
 - 2.2. Detectar cambios en las notas simultáneas
 - 2.3. Tras cada cambio analizar el grupo de notas anterior
 - 2.3.1. Preparar las notas: ordenarlas ascendentemente
 - 2.3.2. Calcular intervalos respecto al bajo
 - 2.3.3. Normalizar intervalos
 - 2.3.4. Ordenar ascendentemente
 - 2.3.5. Eliminar duplicados y equivalentes
 - 2.3.6. Calcular el tipo y las inversiones del acorde a partir de los intervalos
 - 2.3.7. Obtener el resto de información necesaria para los acordes
3. Crear un mecanismo para mostrar mensajes e indicaciones en la propia partitura
4. Análisis de enlaces entre acordes
 - 4.1. Decidir las reglas a aplicar y los parámetros de las reglas
 - 4.2. Crear un mecanismo de reglas que permita personalizar
5. Crear ejercicios
 - 5.1. Crear acordes correctos
 - 5.2. Crear secuencias correctas de acordes
 - 5.3. Implementar la notación del Bajo Cifrado
 - 5.4. Métodos para crear acordes desde un Bajo Cifrado
 - 5.5. Crear ejercicios de los tipos propuestos
 - 5.6. Corregir los ejercicios

3.3 Problemas encontrados

3.3.1 Estructuras adecuadas para los acordes

Durante buena parte del desarrollo no fuimos suficientemente conscientes de la importancia de crear estructuras adecuadas para representar los acordes. Inicialmente nos dejamos llevar por la definición de acorde como un simple conjunto de notas, y pensamos que para describirlo bastaba partir de una colección de notas y, simplemente, añadir otras propiedades a medida que se fuesen necesitando.

Pronto surgió el problema de que la información de intervalos debíamos guardarla en otra estructura aparte, porque necesitábamos modificarla durante el procesamiento. De esta forma aparecieron dos clases, una para el acorde y otra para los datos provisionales del acorde. Pero, en realidad, ambas guardaban datos de 'candidatos a acorde' usados para el cálculo del tipo.

Si se encontraba un tipo correcto para el acorde, entonces los datos provisionales ya se ignoraban y aparecían datos nuevos, exclusivos de un acorde correcto.

En definitiva, existían dos clases para los acordes y ambas se usaban para almacenar sus propiedades tanto antes como después de validar estos acordes calculando el tipo.

Esta mezcla de estados y de datos provocaba mucha confusión y dificultaba el control del procesamiento en sí mismo.

Entonces nos dimos cuenta de que no era correcto permitir que los objetos acordes cambiaseen de estado, sino que debía haber una clase distinta para cada estado de procesamiento.

También nos dimos cuenta de cuáles eran realmente los estados de procesamiento de los acordes, que básicamente consistían en: un simple grupo de intervalos, un acorde reconocido y un acorde con notas.

Tras este descubrimiento, nos decidimos a rehacer completamente las clases de los acordes, y el resultado fue que a partir de entonces todo resulta mucho más sencillo. Por ejemplo, añadir nueva funcionalidad como el bajo cifrado, ya no representaba problema alguno.

3.3.2 Cómo delimitar temporalmente los acordes

La primera etapa consistía en detectar los *protoacordes* (cualquier grupo de notas simultáneas) a partir de una partitura, que estaría ya en el formato simbólico interno de LenMus (estructuras C+).

+), pues la infraestructura LenMus se encarga de obtener la representación simbólica de la partitura a partir de una entrada que puede venir del propio editor o bien de un fichero de notación musical.

En esta etapa inicial nos costó dar con una forma de distinguir un acorde de otro. El problema venía de que sólo considerábamos los instantes de tiempo que coincidían con los pulsos musicales, y esto resultó ser insuficiente. Se puede resumir la cuestión diciendo que se distingue un acorde nuevo cuando en el grupo de notas activas cambia alguna. En ese instante temporal, el acorde ha dejado de sonar, y esto puede ocurrir dentro o fuera de los pulsos.

Los pulsos son divisiones del tiempo de cada compás, determinadas por la métrica, y sirve para saber cuándo acaba un compás, y para otras cosas en las que no entraremos, como la determinación de los acentos.

En la música, se considera que los acordes deben siempre coincidir con posiciones temporales de pulsos, por ello inicialmente solo buscábamos notas en esos instantes. Sin embargo, nos dimos cuenta de que resultaba mucho más fiable, sencillo y práctico considerar todos los instantes de tiempo. De esta forma nos aseguramos de que no se nos escapa ningún cambio en las notas.

3.3.3 Control del ritmo armónico

Una controversia que surgió fue la de si debíamos controlar que los acordes se ajustasen a los conceptos clásicos de Ritmo Armónico. Según la teoría musical clásica, los acordes deben ir en pulso fuertes (acentos) y además deben aparecer preferentemente al comienzo de los compases.

Estas reglas, como tantas otras en la música, admiten matizaciones y no son de obligado cumplimiento. Por ello optamos por dejar toda la parte de detección de errores para la etapa de análisis y, en la fase de reconocimiento de acordes, permitir que los acordes se situasen en cualquier instante de tiempo.

En general, a lo largo del proyecto se prefirió simplificar la fase de detección de acordes, con la intención de dejar una fase de análisis configurable y adaptable a los criterios del tutor, y a la subjetividad de las reglas musicales.

3.3.4 Obtención de tipo del acorde

Un tema tan delicado como encontrar el tipo resultó más fácil de lo previsto gracias al uso de una simple tabla que relaciona de forma unívoca los intervalos de un acorde con su tipo básico.

Esta tabla es independiente de las notas y, por tanto, su representación en la escala diatónica: de tonalidad, alteraciones, etc., solo depende de los intervalos, y esta simplicidad ha resultado clave para poder avanzar sin complicaciones.

A partir de esta tabla de tipos, se genera otra aplicando a cada tipo todas sus posibles inversiones. Cuando se quiere identificar un acorde basta con buscar su conjunto característico de intervalos en esta tabla, y nos dará el tipo correspondiente junto con sus inversiones.

Una posibilidad que evitaría la segunda tabla es probar las inversiones sobre los propios acordes. Así se hizo inicialmente y funcionaba, pero se prefirió finalmente crear la segunda tabla para poder reutilizarla en otras partes de LenMus relacionadas con la armonía, donde podría resultar útil.

La tabla básica que permite obtener los tipos de acordes a partir de los intervalos es la que se incluye a continuación.

Insistimos de nuevo en que el uso de esta sencilla tabla ha sido la clave para conseguir resolver de forma relativamente sencilla una de las tareas más complejas del analizador: la de calcular el tipo de los acordes.

```
static lmChordData tChordData[ect_Max] =
{
    //Triads:
    { 3, { lm_M3, lm_p5 } }, //MT      - MajorTriad
    { 3, { lm_m3, lm_p5 } }, //mT      - MinorTriad
    { 3, { lm_M3, lm_a5 } }, //aT      - AugTriad
    { 3, { lm_m3, lm_d5 } }, //dT      - DimTriad
    //Suspended:
    { 3, { lm_p4, lm_p5 } }, //I,IV,V - Suspended_4th
    { 3, { lm_M2, lm_p5 } }, //I,II,V - Suspended_2nd
    //Sevenths:
    { 4, { lm_M3, lm_p5, lm_M7 } }, //MT + M7 - MajorSeventh
    { 4, { lm_M3, lm_p5, lm_m7 } }, //MT + m7 - DominantSeventh
    { 4, { lm_m3, lm_p5, lm_m7 } }, //mT + m7 - MinorSeventh
    { 4, { lm_m3, lm_d5, lm_d7 } }, //dT + d7 - DimSeventh
    { 4, { lm_m3, lm_d5, lm_m7 } }, //dT + m7 - HalfDimSeventh
    { 4, { lm_M3, lm_a5, lm_M7 } }, //aT + M7 - AugMajorSeventh
    { 4, { lm_M3, lm_a5, lm_m7 } }, //aT + m7 - AugSeventh
    { 4, { lm_m3, lm_p5, lm_M7 } }, //mT + M7 - MinorMajorSeventh
    //Sixths:
    { 4, { lm_M3, lm_p5, lm_M6 } }, //MT + M6 - MajorSixth
    { 4, { lm_m3, lm_p5, lm_M6 } }, //mT + M6 - MinorSixth
    { 4, { lm_M3, lm_a4, lm_a6 } }, // - AugSixth
    //Ninths:
    { 5, { lm_M3, lm_p5, lm_m7, lm_M9 } }, // - DominantNinth = dominant-seventh + major ninth
    { 5, { lm_M3, lm_p5, lm_M7, lm_M9 } }, // - MajorNinth = major-seventh + major ninth
    { 5, { lm_m3, lm_p5, lm_m7, lm_M9 } }, // - MinorNinth = minor-seventh + major ninth
    //11ths:
    { 6, { lm_M3, lm_p5, lm_m7, lm_M9, lm_p11 } }, // - Dominant_11th = dominantNinth+perfect 11th
    { 6, { lm_M3, lm_p5, lm_M7, lm_M9, lm_p11 } }, // - Major_11th = majorNinth + perfect 11th
    { 6, { lm_m3, lm_p5, lm_m7, lm_M9, lm_p11 } }, // - Minor_11th = minorNinth + perfect 11th
    //13ths:
    { 7, { lm_M3, lm_p5, lm_m7, lm_M9, lm_p11, lm_M13 } }, // - Dominant_13th = dominant_11th + major 13th
    { 7, { lm_M3, lm_p5, lm_M7, lm_M9, lm_p11, lm_M13 } }, // - Major_13th = major_11th + major 13th
    { 7, { lm_m3, lm_p5, lm_m7, lm_M9, lm_p11, lm_M13 } }, // - Minor_13th = minor_11th + major 13th
    //Other:
}
```

```
//{ 2, { lm_p5 }}, // - PowerChord = perfect fifth, (octave)
{ 4, { lm_a2, lm_a4, lm_a6 }}, // - TristanChord = augm. fourth, augm. sixth, augmented second
};
```

3.3.5 Visualización de resultados e indicaciones

Es conocida la importancia del interfaz de usuario en cualquier aplicación. Una interfaz poco confortable o ergonómica puede arruinar cualquier desarrollo, por interesante que resulte su funcionalidad.

En el caso de este proyecto, dado lo ambicioso de sus objetivos quedaba poco tiempo para proponerse una interfaz muy elaborada. Sin embargo, al poco tiempo de progresar con el analizador, comenzó a hacerse patente la necesidad de poder mostrar información sobre el resultado del análisis. Más que para el usuario final, esto era importante para la depuración del desarrollo.

Lógicamente, aparecía mucha información en forma de trazas, mediante mensajes que se mostraban secuencialmente en una ventana. Pero aun siendo esto muy útil, resultaba insuficiente. El motivo es que la información de las trazas era siempre muy abundante, aun cuando se intentase reducir mediante filtrado por niveles de relevancia.

Buscar la información relevante dentro de la ventana resultaba demasiado lento, por lo que se decidió dedicar esfuerzos extras a implementar un mecanismo de visualización de mensajes e indicaciones dentro de la propia partitura.

De esta forma se crearon extensiones en la funcionalidad del editor de partituras de LenMus para poder ubicar cuadros de texto en cualquier lugar de la partitura gráfica. También se consiguió añadir una flecha al cuadro de texto y poder asociar la flecha a cualquier nota de la partitura.

Esto significó un gran avance en el desarrollo, pues, tras el análisis, ya se podía ver toda la información relevante dentro de la propia partitura, sin necesidad de buscar en la ventana de depuración.

Inicialmente se mostraba un cuadro de texto para cada acorde detectado, detallando el tipo, las notas y otra información del acorde. Esto resultó muy práctico en el desarrollo, pero se suprimió en la versión final, pues recargaba demasiado la partitura, y se optó por mostrar solo los errores.

Podría resultar interesante permitir al usuario decidir si desea o no que se le muestre la información exhaustiva de los acordes, pues quizás para los principiantes sí que sería práctico y útil. Se deja esto como una posible mejora.

Un aspecto complicado de la visualización de mensajes era cómo colocar la información sobre la partitura para que evitar que se colocasen unos mensajes sobre otros. Intentamos un mecanismo simple a base de hacer que la posición del mensaje fuese relativa a la posición de la nota a la que señalaba (normalmente el bajo del acorde), pero resultaba frecuente que los mensajes se saliesen de la partitura. Podrían ponerse límites para evitarlo, pero teniendo en cuenta que en una partitura pueden aparecer una gran cantidad de acordes y muy repartidos, se consideró que resultaría demasiado complicado.

Finalmente se buscó una solución relativamente sencilla, compuesta, a su vez, de varias soluciones simples. Por un lado, se permitió al usuario mover los mensajes mediante el ratón, para así poder separar mensajes solapados. También se añadió una capa de transparencia a los mensajes, de forma que, aun solapándose varios, pudieran leerse. Por último se buscó un algoritmo de colocación en la partitura sencillo pero efectivo: cada nuevo mensaje se colocaba debajo y a la derecha del anterior, hasta llegar al límite derecho de la partitura, y entonces cada nuevo mensaje se colocaba debajo, pero a la izquierda.

Así se ha conseguido mostrar todos los mensajes de error simultáneamente, lo cual parece preferible a la opción de Harmony Practice 3, de mostrar uno a uno.

En cualquier caso, el tema de la visualización de mensajes resultó mucho más complejo de lo inicialmente previsto, y su implementación final aún tiene margen para ser mejorada. Se podría planear como una extensión o mejora realizar un mecanismo más inteligente para mostrar las indicaciones, además de permitir controlarlo mediante opciones.

3.3.6 Generación de armonías a cuatro voces

Para los ejercicios de armonía se precisaba poder generar secuencias de acordes armónicamente correctas, si bien no se ha entrado en la generación automatizada de armonías a partir de una melodía; esto es un campo muy estudiado y bastante interesante, que podría desarrollarse en una futura mejora o extensión de este analizador, o bien como otro proyecto fin de carrera.

Hay que tener en cuenta que se necesitan acordes de cuatro voces, lo que significa que hay que duplicar alguna de las tres notas del acorde en una octava superior. Esta duplicación hace que generar un único acorde no resulte inmediato. Se descartó de inmediato por inviable computacionalmente y por poco elegante la solución de *fuerza bruta* de generar notas al azar hasta dar con un acorde correcto.

Se optó por un camino más lógico. Se creó un constructor nuevo para la clase de acordes que permitiese crearlo a partir de información básica: grado, octava, intervalos e inversiones. Tras generar el acorde se calculaban las notas a partir del bajo y los intervalos. Luego había que decidir qué nota se duplicaba y en qué octava. De esta forma, se conseguía un acorde correcto.

Si generar aleatoriamente acordes que correctos individualmente era ya complicado, hacer que a su vez formasen una secuencia armónicamente correcta complicaba aún más la tarea: hay muchas reglas y resulta altamente probable romper alguna. De nuevo se descartó una solución *fuerza bruta*, por el alto riesgo de *explosión combinatoria* de soluciones, que impediría encontrar una en un tiempo razonablemente bajo.

Para disminuir la probabilidad de que el acorde incumpla alguna regla de armonía, se aplicaron ciertas restricciones adicionales, en forma de reglas *heurísticas*, que acotan el campo de búsqueda de la solución.

Es preciso destacar que además de las restricciones explícitas de las reglas formales de la armonía, en estos ejercicios de armonía a cuatro voces se dan otras restricciones implícitas, en concreto se debe conseguir que las cuatro voces aparezcan entre las octavas 2 y 5, de forma que no se salgan *demasiado* de los dos pentagramas usados para estos ejercicios, y que son los típicos de las partituras de piano:

Pentagrama superior, o de *melodía*: va de *MI4* a *FA5*. Clave de SOL.

Pentagrama inferior, o de *acompañamiento*: va de *SOL2* a *LA3*. Clave de FA.

Por tanto esta tarea ya sí se puede considerar que entra en el terreno de la *Inteligencia Artificial*, puesto que no tenemos una forma a priori segura de conseguir encuadrar el acorde en los dos pentagramas.

Para superar este problema se recurrió de nuevo a aplicar reglas heurísticas, que no garantizaban dar directamente con la solución pero sí permitían acotar las soluciones a probar, y así hacer viable una búsqueda con *backtracking*.

El algoritmo que sintetiza armonías utiliza plenamente la potencialidad y robustez de la jerarquía de clases de acordes creada y se explica con detalle en el capítulo de Logros.

Para concluir, podemos decir que la tarea de generar acordes resultó más difícil de lo esperado y exigió una mezcla de técnicas, algunas de las cuales pertenecen a la Inteligencia Artificial.

4 Evaluación y pruebas

4.1 Metodología de pruebas

Como se ha mencionado anteriormente, la metodología de desarrollo que se siguió en este proyecto estaba inspirada en las llamadas metodologías *ágiles*, en las cuales el papel de las pruebas es absolutamente esencial, hasta el punto de haber dado origen a una técnica de *desarrollo dirigido por las pruebas*: TDD (*Test Driven Design*).

El concepto clave de TDD es crear la prueba lo primero. También es esencial que las pruebas se ejecuten automatizadamente, de forma que se puedan lanzar en cualquier momento y rápidamente se obtenga el resultado.

Ciertamente de poco sirven las pruebas si no están bien diseñadas. Sobre los diversos tipos de pruebas y las técnicas de diseño de las mismas, hay mucha bibliografía, pero bien puede servir el libro (gratuito, por cierto) de Carlos Blé "*Diseño Ágil con TDD*" [BLE10].

En nuestro caso, dividimos las pruebas en dos tipos bien diferenciados:

- Pruebas unitarias: automatizan mediante la herramienta UnitTest++
- Pruebas de integración: se realizan manualmente cargando partituras de prueba en notación LDP y lanzando el Analizador Tonal

4.2 Pruebas unitarias

La utilidad UnitTest++, utilizada para las pruebas unitarias, resultó bastante cómoda de usar y relativamente sencilla. Quizás la única pega es que la información sobre ella no es abundante aún.

A continuación se describen algunos detalles del uso de UnitTest++ en LenMus:

- En UnitTest++ los test se crean con la macro TEST{ }, que convierte el contenido entre las llaves en métodos ejecutados por UnitTest::RunAllTests()
- Una variante de TEST{ } permite crear "*fixtures*":

```
struct SomeFixture
{
    SomeFixture() { /* some setup */ }
    ~SomeFixture() { /* some teardown */ }
    int testData;
};
```

Analizador tonal en software libre

```
// En lugar de usar TEST(miTest) se usa TEST_FIXTURE(miFixture, miTest):
TEST_FIXTURE(SomeFixture, YourTestName)
{
    int temp = testData;
}
```

- Lanzar test: en modo consola con la opción "**t**" o al ejecutar en *debug* la opción de menú "Unit test".

En el código fuente, los test se lanzan con la siguiente llamada:

```
lmTestRunner::RunTests()
{
    // ...
    //Run the tests
    using namespace UnitTest;
    lmTestReporter reporter(outdata);
    TestRunner runner(reporter);
    runner.RunTestsIf(Test::GetTestList(), NULL, True(), 0);
    //...
}
// int RunTestsIf(TestList const& list, char const* suiteName,
//                 const Predicate& predicate, int maxTestTimeInMs) const
```

- Se pueden crear colecciones de test o "suites"

```
// Cada suite crea un namespace
//      * permite repetir nombres de tests
// Es un argumento opcional de RunAllTests
//      * se puede lanzar una suite concreta

SUITE(MiColeccionDeTest)
{
    TEST(MiTest)
    {
    }
}
```

- Se ha creado un *sucesor* de TestReporter (lmTestReporter) adaptado para dirigir la salida de los tests a un fichero mediante un *stream* de STL: std::ofstream
- El *reporter* crea un fichero UnitTests-results.txt en el directorio build

Como ejemplo, se muestran algunos de los test unitarios correspondientes al analizador armónico:

```

TEST(HarmonicDirection)
{
    // Harmonic direction of an interval:
    // descending (-1) when the interval is negative, ascending (1) when positive,
    linear when 0
    CHECK( GetHarmonicDirection(-1) == -1);
    CHECK( GetHarmonicDirection(-66000) == -1);
    CHECK( GetHarmonicDirection(1) == 1);
    CHECK( GetHarmonicDirection(67000) == 1);
    CHECK( GetHarmonicDirection(0) == 0);
}

TEST(HarmonicMovementType)
{
    // Harmonic motion of 2 voices: calculated from the harmonic direction of each
    voice
    // parallel when both voices have the same direction
    // contrary: both voices have opposite direction
    // oblique: one direction is linear and the other is not
//    GetHarmonicMovementType( lmFPitch fVoice10, lmFPitch fVoice11, lmFPitch fVoice20,
lmFPitch fVoice21)
    CHECK( GetHarmonicMovementType( 10, 20, 5, 6) == lm_eDirectMovement);
    CHECK( GetHarmonicMovementType( 15, 14, 30, 10) == lm_eDirectMovement);
    CHECK( GetHarmonicMovementType( 10, 20, 6, 5) == lm_eContraryMovement);
    CHECK( GetHarmonicMovementType( 15, 14, 10, 30) == lm_eContraryMovement);
    CHECK( GetHarmonicMovementType( 12, 12, 5, 6) == lm_eObliqueMovement);
    CHECK( GetHarmonicMovementType( 23, 23, 6, 5) == lm_eObliqueMovement);
}

TEST(FPitchInterval)
{
// Get interval in FPitch from:
// chord degree (root note step)
// key signature
// interval index (1=3rd, 2=5th, etc)
//lmFIntval FPitchInterval(int nRootStep, lmEKeySignatures nKey, int nInterval)
    CHECK(FPitchInterval(lmSTEP_C, earmDo, 1) == lm_M3);
    CHECK(FPitchInterval(lmSTEP_C, earmDo, 2) == lm_p5);
    CHECK(FPitchInterval(lmSTEP_A, earmLam, 1) == lm_m3);
    CHECK(FPitchInterval(lmSTEP_A, earmLam, 2) == lm_p5);
}

struct SortChordNotesFixture
{
bool CheckOrdered(int nNumNotes, lmFPitch fInpChordNotes[])
{
    for (int i = 1; i < nNumNotes && fInpChordNotes[i]; i++)
        for (int j = 0; j < i; j++)
            if (fInpChordNotes[j] > fInpChordNotes[i])
                return false;
    return true;
}
};

TEST_FIXTURE(SortChordNotesFixture, SortChordNotes)
{
    const int nNumNotes = 5;
    lmFPitch fInpChordNotes1[nNumNotes] = { 39, 30, 20, 11, 0};
    SortChordNotes(nNumNotes, fInpChordNotes1);
    CHECK ( CheckOrdered(nNumNotes, fInpChordNotes1) );
    lmFPitch fInpChordNotes2[nNumNotes] = { 0, 30, 1, 11, 2};
    SortChordNotes(nNumNotes, fInpChordNotes2);
    CHECK ( CheckOrdered(nNumNotes, fInpChordNotes2) );
    lmFPitch fInpChordNotes3[nNumNotes] = { 5, 1, 3, 11, 39};
    SortChordNotes(nNumNotes, fInpChordNotes3);
    CHECK ( CheckOrdered(nNumNotes, fInpChordNotes3) );
    lmFPitch fInpChordNotes4[nNumNotes] = { 0, 1, 3, 11, 39};
}

```

```

SortChordNotes(nNumNotes, fInpChordNotes4);
CHECK( CheckOrdered(nNumNotes, fInpChordNotes4) );
}

TEST(Intervals)
{
    const int nNumIntv = 5;
    const int i1 = 1;
    const int i2 = 12;
    const int i3 = 31;
    lmFIntval pFI[nNumIntv] = { lm_p8, i3+lm_p8, i3, i2+(lm_p8*2), i1 };
    lmChordIntervals ci(nNumIntv, pFI);
    // first check raw intervals
    CHECK( ci.GetNumIntervals() == nNumIntv );
    CHECK( ci.GetInterval(4) == pFI[4] );
    // then normalize: reduce octave, remove duplicated and sort
    ci.Normalize();
    int i = ci.GetNumIntervals();
    CHECK(3 == ci.GetNumIntervals());
    i = ci.GetInterval(0);
    CHECK(i1 == ci.GetInterval(0));
    i = ci.GetInterval(1);
    CHECK(i2 == ci.GetInterval(1));
    i = ci.GetInterval(2);
    CHECK(i3 == ci.GetInterval(2));
}

TEST(Chords)
{
    // Test data set. TODO: make a fixture with several data sets for different chords,
    // including inversions
    // MajorTriad chord
    wxString sRootNote = _T("c4");
    wxString psNotes[] = { _T("c4"), _T("e4"), _T("g4") };
    const int nNumNotes = 3;
    wxString sIntervals = _T("M3,p5");
    const int nNumIntervals = nNumNotes - 1;
    lmEKeySignatures nKey = earmDo;
    int nNumInversions = 0;
    lmEChordType nChordType = ect_MajorTriad;
    int nDegree = lmSTEP_C; // root C4, no inversion : degree I
    int nOctave = 4; // C4: octave 4
    // notes for lmFPitchChord
    lmFPitch fNotes[] = { FPitch(psNotes[0]), FPitch(psNotes[1]), FPitch(psNotes[2]) };
    // With lmFPitchChord the notes can follow the chord rules:
    // notes can be duplicated
    // a note can be increased any number of octaves
    // but the notes must be ordered
    lmFPitch fNotes2[] =
    // AWARE: remember: the bass note is ABSOLUTE (octave dependent);
    // the rest are RELATIVE (octave independent)
    // Therefore if two chords are equivalent, both must have the same absolute bass
    note
    // Therefore, the bass note must be in the first position, while the rest can be
    unordered
    { FPitch(psNotes[0]), FPitch(psNotes[2])+(lm_p8*4), FPitch(psNotes[1])+(lm_p8*3),
      FPitch(psNotes[2]) };

    // Check constructors
    // build a chord from root note and type
    // lmChord(wxString sRootNote, lmEChordType nChordType, int nInversion = 0,
    lmEKeySignatures nKey = earmDo);
    lmChord C1(sRootNote, nChordType, nNumInversions, nKey);
    // build a chord from a list of notes in LDP source code
    lmChord C2(nNumNotes, psNotes, nKey);
    // build a chord from a list of intervals (as strings)
    lmChord C3(sRootNote, sIntervals, nKey);
    // build a chord from "essential" information
}

```

```

lmChord C4(nDegree, nKey, nNumIntervals, nNumInversions, nOctave);

wxString sC1 = C1.ToString(); // for debug
wxString sC2 = C2.ToString();
wxString sC3 = C3.ToString();
wxString sC4 = C4.ToString();

CHECK( C1.AreEqual(&C2) );
CHECK( C2.AreEqual(&C3) );
CHECK( C3.AreEqual(&C4) );
CHECK( C4.AreEqual(&C1) );

CHECK(sC4.compare(sC3) == 0);
CHECK(sC3.compare(sC2) == 0);
CHECK(sC2.compare(sC1) == 0);
CHECK(sC1.compare(sC4) == 0);

// lmFPitchChord(int nNumNotes, lmFPitch fNotes[], lmEKeySignatures nKey = earmDo);
lmFPitchChord C5(nNumNotes, fNotes, nKey);
const int nNumfNotes2 = sizeof(fNotes2) / sizeof(fNotes2[0]);
lmFPitchChord C6(nNumfNotes2, fNotes2, nKey);

CHECK( C5.lmChord::EqualTo(&C2) );
CHECK( C6.lmChord::EqualTo(&C3) );
CHECK( C6.EqualTo(&C5) );

// aware: lmFPitchChord is a chord with absolute notes
// therefore, two lmFPitchChord can have different notes but
// the same chord type (lmChord)
// in this case, both are "equal" when compared using
// EqualTo()
// but different when comparing the ToString()
// conclusion: in this case, compare the ToString()
// of the parent class lmChord
wxString sC5= C5.lmChord::ToString();
wxString sC6= C6.lmChord::ToString();
CHECK(sC5.compare(sC1) == 0);
CHECK(sC6.compare(sC2) == 0);
CHECK(sC5.compare(sC6) == 0);
}
}

```

4.3 Pruebas de Integración

La pruebas de integración se han realizado de forma manual. Su ejecución es bastante simple, ya que tan solo se precisa cargar los ficheros con las partituras de prueba para la armonía , lanzar el analizador y verificar, por inspección visual, que los resultados son correctos.

Su automatización no se ha llegado a implementar porque no aportaba ventajas realmente significativas y el elevado esfuerzo que suponía su automatización se ha preferido dedicar a ampliar la lista de objetivos conseguidos.

En LenMus existe un directorio especial donde se guardan partituras para pruebas de integración y, dentro de este directorio, se han creado un grupo de partituras específicas para probar el Analizador Tonal, con variados tipos de acordes y de enlaces entre ellos.

Analizador tonal en software libre

Para pasar una prueba, simplemente se abre en LenMus la partitura correspondiente y se le pasa el analizador. Posteriormente se comparan los resultados mostrados por el analizador con los valores esperados.

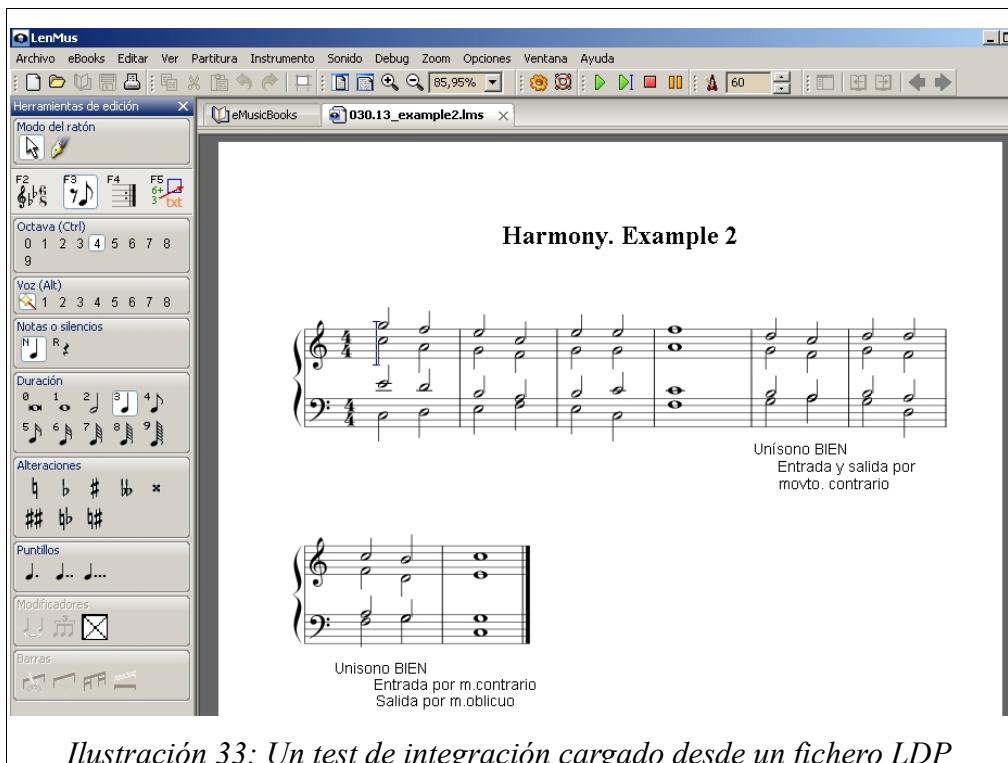


Ilustración 33: Un test de integración cargado desde un fichero LDP

En la figura se observa la partitura generada automáticamente al cargar uno de los ficheros de prueba. La prueba consiste en aplicar el analizador y comprobar manualmente los resultados.

The screenshot shows a musical score titled "Harmony. Example 2" with two staves. Various annotations are overlaid on the music, primarily in red boxes, indicating harmonic analysis results. These annotations include:

- "Unisono BIEN Entrada y salida por m. contrario"
- "Parallel motion of 5ª justa, chords: 8, 9; v3 g4-->f4, v4 d5-->c5, Interval: p5"
- "Parallel motion of 5ª justa, chords: 9, 10; v3 f4-->g4, v4 c5-->d5, Interval: p5"
- "Direct movement resulting 5ª justa. Chords: 3, 4. Voices: 4 e5-->f4 and 3 g4-->f4. Interval: p5"
- "Unisono BIEN Entrada y salida por m. contrario"
- "Direct movement resulting 5ª justa. Chords: 7, 8. Voices: 4 f5-->d5 and 3 a4-->g4. Interval: p5"
- "Direct movement resulting 5ª justa. Chords: 8, 9. Voices: 4 d5-->c5 and 3 g4-->f4. Interval: p5"
- "Direct movement resulting 5ª justa. Chords: 9, 10. Voices: 4 c5-->d5 and 3 f4-->g4. Interval: p5"
- "Direct movement resulting 5ª justa. Chords: 10, 11. Voices: 2 b3-->a3 and 1 g3-->d3. Interval: p5"
- "Direct movement resulting 5ª justa. Chords: 12, 13. Voices: 3 f4-->d4 and 1 a3-->g3. Interval: p5"

Ilustración 34: Resultados de aplicar el analizador el test de integración

Como se observa en la figura, en las partituras de prueba pueden aparecer muchos mensajes de error, lo que hace poco práctica la corrección manual. Transformar esta corrección en automática es una mejora prevista para el próximo futuro.

5 Resultados y conclusiones

5.1 Logros

Los resultados obtenidos se pueden calificar de **altamente satisfactorios**, pues se han cumplido todos los objetivos establecidos y, además, se ha conseguido hacerlo con un conjunto de algoritmos y estructuras **altamente reutilizable**.

También se pueden destacar algunos logros que van más allá de los objetivos iniciales. El analizador implementado es adaptable y extensible gracias a su diseño modular y a la parametrización de sus métodos, además cabe destacar que maneja acordes de cualquier número de notas e independientes del contexto tonal (la tonalidad es un parámetro más).

A continuación se resumen los logros conseguidos y después se detallan en apartados .

5.1.1 El Analizador Tonal

Se han conseguido **todos** los objetivos propuestos para el Analizador de Armonía Tonal, descritos en el capítulo 1.1.2. Las funcionalidades implementadas se pueden resumir en los siguientes puntos :

- Detectar acordes (grupos de notas simultáneas).
- Averiguar si el acorde se corresponde con un tipo válido.
- Calcular las propiedades de los acordes: tipo, inversiones, nota fundamental, cifrado.
- Analizar el conjunto de acordes detectados aplicando una lista variable de reglas de progresión armónica.
- Generar composiciones de acordes válidos.
- Generar ejercicios de completar propiedades previamente calculadas, incluyendo cifrado.
- Corregir los ejercicios realizados por el alumno.
- Mostrar los mensajes de error y otras indicaciones dentro de la propia partitura sin solaparse e indicando gráficamente el origen del problema.

Junto a los mencionados objetivos funcionales, se han conseguido también otros logros valiosos que merecen ser destacados y que se refieren a características no-funcionales, o bien a extensiones más allá de los objetivos. A continuación se destacan algunas de las características más relevantes:

- El Analizador de acordes es *genérico*:
 - Acordes de cualquier número de notas.
 - *Acordes independientes del contexto (tonalidad)*.
- El Analizador de acordes es adaptable y extensible: parametrizado y modular.
- El mecanismo de inferencia de las reglas de armonía es adaptable:
 - Facilita selección de reglas a aplicar.
 - Facilita creación de nuevas reglas y adaptación de las existentes.
- La arquitectura de clases de acordes es versátil:
 - Permite representar acordes en cualquier nivel de abstracción y contexto.
 - Métodos y funciones para manipulación de acordes y sus atributos.
 - Taxonomía reusable y genérica para procesar acordes

5.1.2 Arquitectura de clases para representar y procesar acordes

Entre los logros relevantes de este trabajo, merece destacarse la jerarquía de clases utilizada para representar acordes. Estas clases resultan muy prácticas, porque reflejan los distintos niveles evolutivos de un acorde según el contexto en el que se encuentre.

Gracias a que se han minimizado las dependencias innecesarias, las clases se pueden trasladar fácilmente a cualquier lenguaje orientado a objetos. Además, las clases están pensadas para acordes con cualquier número de notas.

Al observar la estructura de las clases, se puede comprender qué propiedades de los acordes son las relevantes en cada contexto de uso, y en cada nivel de procesamiento, y por ello sirven no solo para almacenar la información de los acordes, sino para orientar en los pasos que conviene seguir al procesar acordes.

A continuación se describe mediante un diagrama la jerarquía creada y se explican los mencionados niveles en la evolución de los acordes.

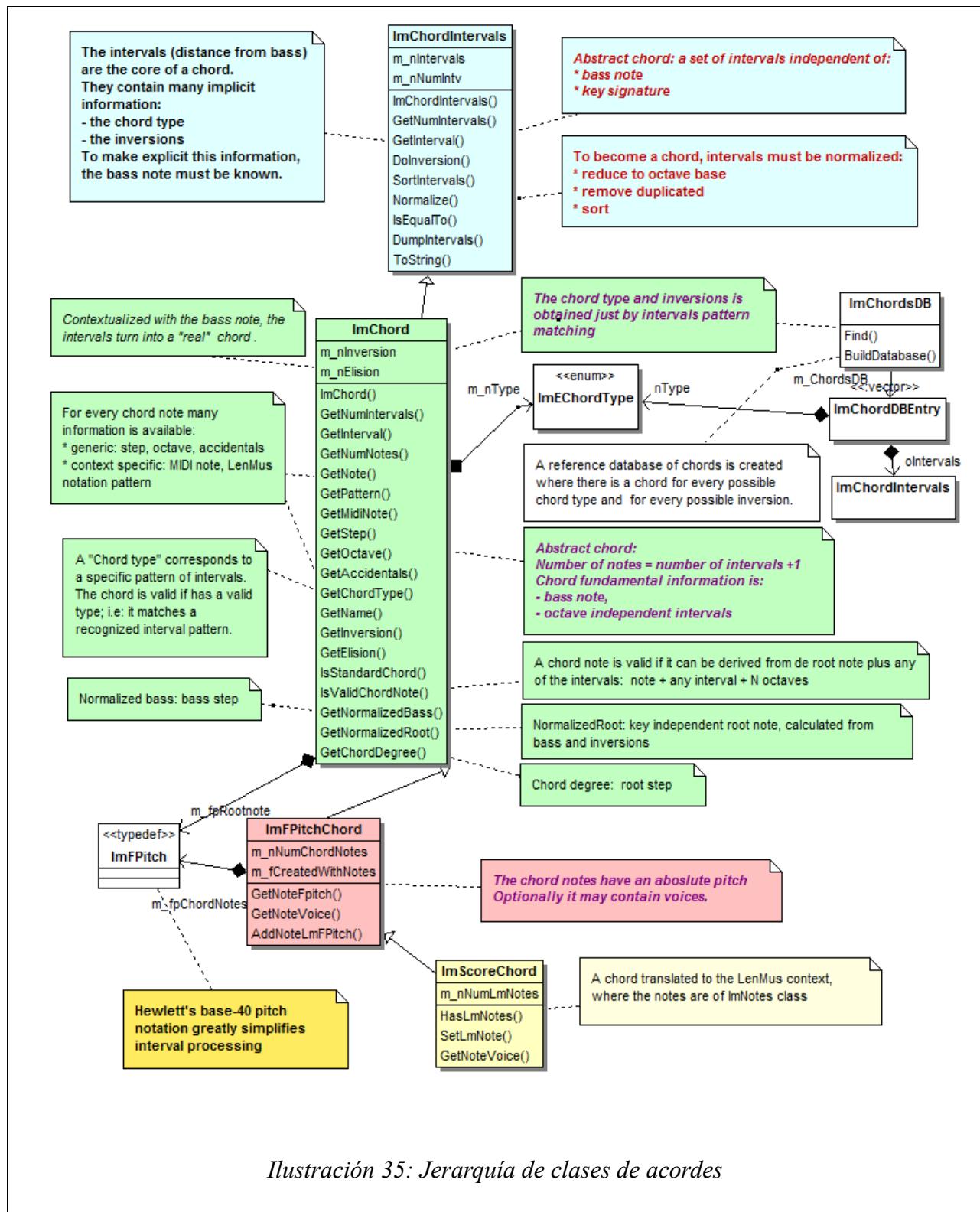


Ilustración 35: Jerarquía de clases de acordes

La jerarquía de clases mostrada es el fruto de la experiencia adquirida en el desarrollo de este analizador, y puede resultar útil para procesar armonía en cualquier entorno *orientado a objetos*.

En la siguiente tabla se detallan las clases de acorde y su nivel *evolutivo*, donde cada nivel es una *especialización* del anterior.

Nombre	Nivel	Descripción	Extensión respecto al nivel anterior
<i>lmChordIntervals</i>	0	Conjunto de intervalos sin limitaciones.	
<i>lmChordIntervals</i>	1	Acorde <i>potencial</i> . Se han normalizado los intervalos, lo cual permite verificar ya si cumple las reglas de formación de acordes.	Intervalos <i>normalizados</i> : <ul style="list-style-type: none">• Reducidos al rango de octava• Duplicados eliminados• Ordenados
<i>lmChord</i>	2	Acorde <i>nominal</i> : cumple las reglas de formación de acordes. No considera notas repetidas: nº de notas = nº de intervalos + 1. El acorde tiene tipo	Se conocen la nota fundamental, el bajo, las inversiones, el tipo, e incluso las notas elididas.
<i>lmFPitchChord</i>	3	Acorde real: con todas las notas reales.	Se permite notas repetidas.
<i>lmScoreChord</i>	4	Acorde LenMus: las notas son objetos de una partitura LenMus.	Tiene voces armónicas. Permite realizar operaciones de la infraestructura LenMus sobre las notas, como por ejemplo colorearlas.

Tabla 5: Jerarquía de clases de acordes

Seguidamente se incluyen extractos del código fuente perteneciente estas clases.

```
// lmChordIntervals: A generic chord (a list of intervals)
class lmChordIntervals
{
public:
    lmChordIntervals(int nNumIntv, lmFIntval* pFI);
    lmChordIntervals(lmEChordType nChordType, int nInversion);
    lmChordIntervals(wxString sIntervals);
    lmChordIntervals(int nNumNotes, wxString* pNotes);
    lmChordIntervals(int nNumNotes, lmFPitch fNotes[]);
    lmChordIntervals(int nNumNotes, lmNote** pNotes);
    lmChordIntervals(int nStep, lmEKeySignatures nKey, int nNumIntervals, int
nInversion);
    ~lmChordIntervals();

    //accessors
    inline int GetNumIntervals() { return m_nNumIntv; }
    inline lmFIntval GetInterval(int i) { return m_nIntervals[i]; }

    //operations
    void DoInversion();
    void SortIntervals();
    void Normalize();

    bool IsEqualTo(lmChordIntervals* tOther);

protected:
    lmFIntval      m_nIntervals[lmINTERVALS_IN_CHORD];
    int            m_nNumIntv;
};
```

Código de la clase *lmChordIntervals*, para representar acordes de niveles 0 (*intervalos*) y 1 (*intervalos normalizados*).

```

// Aware: fundamental information in a chord is only:
//   BASS note CONTEXT-INDEPENDENT (key independent)
//   e.g.: lmFPitch OCTAVE-INDEPENDENT (0..39) or STEP + accidentals
//   INTERVALS, ELISIONS (if allowed)
// Directly derived from the above are:
//   TYPE, INVERSIONS, ROOT NOTE
// (remember: root note == bass note only if no inversions)
class lmChord : public lmChordIntervals
{
public:
    //build a chord from root note and type
    lmChord(wxString sRootNote, lmEChordType nChordType, int nInversion = 0,
             lmEKeySignatures nKey = earmDo);
    //build a chord from the root note and the figured bass
    lmChord(wxString sRootNote, lmFiguredBass* pFigBass, lmEKeySignatures nKey =
earmDo);
    //build a chord from a list of notes in LDP source code
    lmChord(int nNumNotes, wxString* pNotes, lmEKeySignatures nKey = earmDo);
    //build a chord from a list of score note pointers
    lmChord(int nNumNotes, lmNote** pNotes, lmEKeySignatures nKey = earmDo);
    lmChord(int nNumNotes, lmFPitch fNotes[], lmEKeySignatures nKey);
    //build a chord from a list of intervals (as strings)
    lmChord(wxString sRootNote, wxString sIntervals, lmEKeySignatures nKey);
    // build a chord from "essential" information
    lmChord(int nDegree, lmEKeySignatures nKey, int nNumIntervals, int nNumInversions,
int octave);
    virtual ~lmChord();

    //access to intervals
    inline int GetNumIntervals();
    lmFInterval GetInterval(int i); //1..n

    //access to notes
    inline int GetNumNotes();
    lmFPitch GetNote(int i); //0..n-1
    wxString GetPattern(int i); //0..n-1
    lmMPitch GetMidiNote(int i); //0..n-1
    inline int GetStep(int i);
    inline int GetOctave(int i) ;
    inline int GetAccidentals(int i);

    //chord info
    lmEChordType GetChordType();
    wxString GetNameFull();
    int GetInversion();
    inline int GetElision();
    // A note is valid in a chord if it can be derived from de root note
    // plus any of the intervals, i.e:
    //   note + any interval + N octaves
    int IsValidChordNote(lmFPitch fNote);
    lmFPitch GetNormalizedBass() { return m_fpRootNote % lm_p8; }
    // key independent root note, calculated from bass and inversions
    lmFPitch GetNormalizedRoot();
    lmStepType GetChordDegree();
    bool IsEqualTo(lmChord* tOther);
};

}

```

Código de la clase *lmChord*, para representar acordes de nivel 2 (*acorde nominal*).

```
// lmChord is an "abstract" chord: defined by intervals.
//   lmChord: Number of notes = number of intervals +1
// lmFPitchChord is a "real" chord: it contains a set of actual notes
//   IT ALLOWS DUPLICATED NOTES.
// lmScoreChord: lmFPitchChord with "real" notes in a score context
class lmFPitchChord: public lmChord
{
public:
    // Constructors from notes
    //   (the notes can not be added afterwards)
    //build a chord from a list of ordered notes
    lmFPitchChord(int nNumNotes, lmFPitch fNotes[], lmEKeySignatures nKey = earmDo);
    lmFPitchChord(int nNumNotes, lmNote** pNotes, lmEKeySignatures nKey = earmDo);
    // Constructors without notes
    //   (the notes can be added afterwards)
    //   build a chord from "essential" information
    lmFPitchChord(int nDegree, lmEKeySignatures nKey, int nNumIntervals, int
nNumInversions, int octave);
    virtual ~lmFPitchChord() {};

    int GetNumNotes() {return m_nNumChordNotes;}
    wxString ToString();
    lmFPitch GetNoteFpitch(int nIndex) {return m_fpChordNotes[nIndex];} ;
    // aware: to be used only after using constructor without notes
    // return the number of notes
    int AddNoteLmFPitch(lmFPitch fNote);

protected:
    int m_nNumChordNotes;
    lmFPitch m_fpChordNotes[lmNOTES_IN_CHORD];
    bool m_fCreatedWithNotes;
};
```

Código de la clase *lmFPitchChord*, para representar acordes de nivel 3 (*acorde en una partitura*).

```

class lmScoreChord: public lmFPitchChord
{
public:
    //build a chord from a list of score note pointers
    lmScoreChord(int nNumNotes, lmNote** pNotes, lmEKeySignatures nKey = earmDo);
    // Constructors without notes
    //      (the notes can be added afterwards)
    //      build a chord from "essential" information
    lmScoreChord(int nDegree, lmEKeySignatures nKey, int nNumIntervals, int
nNumInversions, int octave);
    virtual ~lmScoreChord() {};

    bool HasLmNotes() {return m_nNumLmNotes > 0 && m_nNumLmNotes == m_nNumChordNotes; }

    // aware: this is only to associate the score note (lmNote) to a note in lmFPitch
    // already exists
    // it is not to add a note!
    bool SetLmNote(lmNote* pNote);

    lmNote* GetNoteLmNote(int nIndex);
    int GetNoteVoice(int nIndex);
    int GetNumLmNotes();
    wxString ToString();

};


```

Código de la clase *lmScoreChord*, para representar acordes de nivel 4 (*acorde en una partitura de LenMus*).

5.1.3 Algoritmos generales para procesar acordes

También muchos de los algoritmos empleados para el procesamiento de la partitura son altamente reutilizables gracias su mínimo acoplamiento, reducido a simples parámetros.

Veamos a continuación algunos ejemplos destacables.

5.1.3.1 Algoritmo para calcular el tipo de movimiento armónico

```
//  
// Algoritmo para calcular el tipo de movimiento armónico  
// (necesario en varias reglas de progresión armónica)  
//  
int GetHarmonicMovementType( lmFPitch fVoice10, lmFPitch fVoicell,      lmFPitch  
fVoice20, lmFPitch fVoice21)  
{  
    int nMovType = -10;  
  
    int nD1 = GetHarmonicDirection(fVoicell - fVoice10);  
    int nD2 = GetHarmonicDirection(fVoice21 - fVoice20);  
  
    if (nD1 == nD2)      {  
        nMovType = lm_eDirectMovement;  
    }  
    else if (nD1 == -nD2) {  
        nMovType = lm_eContraryMovement;  
    }  
    else {  
        assert ( (nD1 == 0 && nD2 != 0) || (nD2 == 0 && nD1 != 0) );  
        nMovType = lm_eObliqueMovement;  
    }  
    return nMovType;  
}
```

5.1.3.2 Algoritmo para saber si una nota puede pertenecer a un acorde

```
//  
// Algoritmo para saber si una nota puede pertenecer a un acorde  
//  
lmChord::IsValidChordNote(lmFPitch fNote)  
{  
    lmFPitch fpNormalizedRoot = this->GetNormalizedBass();  
    for (int nI=0; nI <= m_nNumIntv; nI++)  
    {  
        // note that valid intervals are: 0 .. m_nNumIntv  
        lmFPitch fpNormalizedNoteDistance = (fNote - GetInterval(nI)) % lm_p8;  
        if ( fpNormalizedRoot == fpNormalizedNoteDistance)  
        {  
            return true;  
        }  
    }  
    return false;  
}
```

5.1.3.3 Algoritmo para aplicar una inversiones a un acorde

```
//
```

```

// Algoritmo para aplicar una inversión a un acorde
//
void lmChordIntervals::DoInversion()
{
    //Do first inversion of list of intervals.
    //Note that second inversion can be obtained by invoking this method
    //two times. The third inversion, by invoking it three times, etc.

    lmFInterval nNewIntervals[lmINTERVALS_IN_CHORD];
    for (int i=0; i < m_nNumIntv-1; i++) {
        nNewIntervals[i] = m_nIntervals[i+1] - m_nIntervals[0];
        if (nNewIntervals[i] < 0)
            nNewIntervals[i] += lm_p8;
    }

    nNewIntervals[m_nNumIntv-1] = lm_p8 - m_nIntervals[0];

    //transfer results
    for (int i=0; i < m_nNumIntv; i++)
        m_nIntervals[i] = nNewIntervals[i];
}

```

5.1.3.4 Algoritmo para detectar acordes a partir de las notas

```

// 
// Algoritmo en pseudocódigo para detectar acordes a partir de las notas
// basado en mantener una lista con las notas activas en cada instante
//
Do while there are notes in the score
    If new note
        if new time (i.e. note-time > current-time ) then
            update current list of active notes: remove notes with end-time < TIME
            if number of active notes > 1
                chord detected: process it
            set new current-time
            add new note to the list of active notes

```

Para la lista de notas activas se ha utilizado una estructura como la que sigue:

```

-----+
// A list of notes
//   with individual absolute end time
//   with global absolute current time
-----+
typedef struct lmActiveNoteInfoStruct {
    lmNote* pNote;
    float rEndTime;
    lmActiveNoteInfoStruct(lmNote* pNoteS, float rEndTImeS)
    {
        pNote = pNoteS;
        rEndTime = rEndTImeS;
    }
} lmActiveNoteInfo;

class lmActiveNotes
{

```

```
public:  
    lmActiveNotes();  
    ~lmActiveNotes();  
  
    void SetTime(float rNewCurrentTime);  
    inline float GetTime() { return m_rcurrentTime; };  
    int GetNotes(lmNote** pNotes);  
    void AddNote(lmNote* pNote, float rEndTime);  
    void RecalculateActiveNotes();  
    int GetNumActiveNotes();  
  
protected:  
    void ResetNotes();  
  
    float m_rcurrentTime;  
    std::list<lmActiveNoteInfo*> m_ActiveNotesInfo;  
};
```

5.1.4 Algoritmo generador de armonías correctas

Como ya se ha relatado en el apartado de dificultades encontradas, la *generación* de secuencias de acordes de cuatro voces resultó mucho más compleja de lo previsto, y se necesitó un gran esfuerzo hasta conseguir implementar y refinar un algoritmo eficiente para conseguirlo.

Debido a la complejidad de la tarea, se ha necesitado recurrir a técnicas de *Inteligencia Artificial* como búsquedas con *backtracking* acotadas por *heurísticas*.

Este algoritmo final es efectivo y original, a la vez que relativamente sencillo, e independiente de la plataforma LenMus, y fácilmente portable a otros entornos.

El proceso consiste simplemente en generar un acorde de tipo *nominal* (nivel 2), es decir, un acorde *genérico*, con *bajo* e *intervalos*, pero **no** con **notas** concretas. A partir de este acorde de nivel 2 se generan acordes equivalentes, pero de nivel 3, es decir, acordes *reales*, formados por un grupo de notas, entre las cuales habrá alguna repetida, puesto que tenemos cuatro voces y el acorde es de *triada*.

Podemos decir que sacamos distintas implementaciones o instancias de un mismo acorde, hasta dar con una que verifique correctamente todas las restricciones armónicas.

A continuación se describe este algoritmo en *pseudocódigo*:

Analizador tonal en software libre

Algoritmo de generación de una secuencia de N acordes correctos. Los acordes son de triada y con 4 voces (bajo, tenor, barítono, soprano).

Resumen:

Para generar cada acorde nuevo:

- 1) Generar aleatoriamente un acorde *nominal* (tipo `lmChord`)
- 2) A partir de ese acorde, usar sus intervalos para calcular un acorde real (tipo `lmFPitchChord`) compatible.
- 3) Analizar la armonía de todos los acordes generados hasta el momento

Desarrollo:

Hasta que tengamos N acordes correctos repetir:

Generar el siguiente acorde:

- 1) Generar aleatoriamente un acorde *nominal* (tipo `lmChord`), es decir, un acorde con intervalos, pero sin notas reales

Preparar los parámetros del constructor de acordes *nominales*:

grado, tonalidad, numero-de-intervalos, inversiones, octava, donde.

octava: aleatorio entre 2 y 3

grado: aleatorio entre 0 y 6

inversiones: si se permiten: aleatorio entre 0 y 2; si no: 0

numero-de-intervalos: 2 (acorde de triada)

Generar el acorde nominal, con los argumentos anteriores

- 2) A partir de ese acorde, usar sus intervalos para calcular un acorde real (tipo `lmFPitchChord`) compatible, es decir:
calcular las 4 voces del acorde.

2.1) voz *bajo*: debe ser la nota 0 del acorde,
pero hay que cuidar de que si se sale del pentagrama inferior,
conviene restarle una octava

2.2) Calcular las otras voces. Esta es la parte más interesante
y delicada...

Para calcular cada una de las otras 3 voces hacer:

`voz[i] = bajo + n octavas + intervalo[k]`

donde debemos elegir un "n" y un "k" para cada voz

"k": es el intervalo que se sumará al bajo.

Debemos considerar las siguientes limitaciones:

- puede ser 0, 1, o 2, pero:

- * usar 0 implica duplicar la fundamental (habitual en armonía)
- * si hay 'quinta elidida', no se puede usar el 2
- * no se permite usar el mismo para las 3 voces

"n": es el número de octavas que se suman. En teoría puede ser
cualquier número entero, pero en nuestro caso, para no salirnos
de las partituras, lo dejamos a cero, salvo para correcciones
puntuales, que son las siguientes:

- una voz debe ser superior a la anterior; si no: subir una octava
- evitar que la voz tenor quede a menos de una tercera del bajo;
si no: subir una octava

motivo: bajo y tenor son las únicas voces consecutivas para las

- que se permite una distancia mayor de octava. Lo aprovechamos subiendo una octava si podemos, para disminuir la probabilidad de error por distancia elevada entre tenor y barítono.
- intentar que el barítono quede en el pentagrama superior, subiendo si es necesario una octava, pero con cuidado de no distanciarse más de una octava del tenor, porque daría error.
- 3) Analizar la armonía de todos los acordes generados hasta el momento: Lanzar el analizador de acordes sobre la secuencia generada hasta el momento, y en caso de algún error: eliminar el último acorde y generar otro en su lugar
 - 4) Una vez tenemos una secuencia correcta de acordes, ya solo quedan detalles menores como:
 - calcular el bajo cifrado correspondiente a cada acorde
 - mostrar el grado de cada acorde
 - ocultar/mostrar la información que el ejercicio requiera (p.ej. mostrar solo las voces bajo)
 - generar la partitura y mostrársela al usuario

Una forma de apreciar y seguir más de cerca el proceso de generación de acordes es observar las trazas que aparecen en la ventana de seguimiento del compilador, cuando se ejecuta usando la versión de depuración.

Se observa que tras generar un nuevo acorde, se procede a analizar toda la secuencia de acordes generada hasta el momento, y si aparece algún error, se rechaza el nuevo acorde. Aquí se muestra solo la traza de los dos primeros acordes. La altura de las notas se indica en el formato 40 de Hewlett, en el cual cada octava supone un incremento de 40.

```

19:27:11: ===== START WITH CHORD 0 =====
19:27:11: Creating lmScoreChord: step:6 octave:2 inversions:2 key:0
19:27:11:   FPitchStepsInterval (Step 1 oct:1) 49 - (Step 6, octave 0) 38 = 11
19:27:11:   FPitchStepsInterval (Step 3 oct:1) 60 - (Step 6, octave 0) 38 = 22
19:27:11: Bass reduced one octave to : 100
19:27:11: Bass voice V4,FINAL: 100 (f2)
19:27:11: <><> CHORD 0: bass STEP:6, octave:2, key:0 inversions:2 ROOT:118 (f2) ===
19:27:11:           lmChord: [ 5ª disminuida, 2ª inversión, Bass:f, Root:b, Intervals:a4(18) M6(29)
Pattern: f3 b3 d4; Notes: f2]
19:27:11:           nIntvB 1 0 2
19:27:11:           V3, after applying interval 1 : 118 (b2), [ni:0]
19:27:11: V3, FINAL: 118 (b2)
19:27:11:           V2, after applying interval 0 : 100 (f2), [ni:1]
19:27:11: Added octave to voice V2: 140
19:27:11: Raise to 2nd staff: added octave to voice V2: 180 (min:148)
19:27:11: V2, FINAL: 180 (f4)
19:27:11:           V1, after applying interval 2 : 129 (d3), [ni:2]
19:27:11: Added octave to voice V1: 169
19:27:11: Added octave to voice V1: 209
19:27:11: V1, FINAL: 209 (d5)
19:27:11: ##BEFORE 0 ANALYSIS OF CHORD LINK ERRORS OF CHORD 0: 5ª disminuida, 2ª inversión,
Bass:f, Root:b, Intervals:a4(18) M6(29) Pattern: f3 b3 d4; Notes: f2 b2 f4 d5
19:27:11: *** AnalyzeHarmonicProgression N:1
19:27:11: Chord 0 to analyze: 5ª disminuida, 2ª inversión, Bass:f, Root:b, Intervals:a4(18)
M6(29) Pattern: f3 b3 d4; Notes: f2 b2 f4 d5
19:27:11: Evaluating rule 2, description: No parallel motion of perfect octaves, perfect fifths,
and unisons
19:27:11: Rule 2 final error count 0
19:27:11: Total error count after rule 2: 0 errors
19:27:11: Evaluating rule 3, description: No resulting fifths and octaves

```

Analizador tonal en software libre

```
19:27:11: Rule 3 final error count 0
19:27:11:     Total error count after rule 3: 0 errors
19:27:11: Evaluating rule 8, description: Voices interval not greater than one octave (except bass-tenor)
19:27:11: Check chord 0
19:27:11: Rule not applicable: not root position: 2 inversions
19:27:11: Rule 8 final error count 0
19:27:11:     Total error count after rule 8: 0 errors
19:27:11: Evaluating rule 9, description: Do not allow voices crossing. No duplicates (only for root position and root duplicated)
19:27:11: Rule not applicable: not root position: 2 inversions
19:27:11: Rule 9 final error count 0
19:27:11:     Total error count after rule 9: 0 errors
19:27:11: ##RESULT: 0 LINK ERRORS
19:27:11: <<<< CHORD 0 OK!!! after 0 attempts >>>>>>
19:27:11: FIGURED BASS:6 4
19:27:11:     Staff 1, V1 (index:0) 209 [d5], pattern: (n d5 q p1 v1 (stem up))
19:27:11: V:1 added pattern: (n d5 q p1 v1 (stem up)) ***
19:27:11: SetLmNote 3 209 OK, total LmNotes:1
19:27:11:     Ej:3 V:1 NEGRO
19:27:11:     Staff 1, V2 (index:1) 180 [f4], pattern: (n f4 q p1 v2 (stem down))
19:27:11: V:2 added pattern: (n f4 q p1 v2 (stem down)) ***
19:27:11: SetLmNote 2 180 OK, total LmNotes:2
19:27:11:     Ej:3 V:2 NEGRO
19:27:11:     Staff 2, V3 (index:2) 118 [b2], pattern: (n b2 q p2 v3 (stem up))
19:27:11: V:3 added pattern: (n b2 q p2 v3 (stem up)) ***
19:27:11: SetLmNote 1 118 OK, total LmNotes:3
19:27:11:     Ej:3 V:3 NEGRO
19:27:11:     Staff 2, V4 (index:3) 100 [f2], pattern: (n f2 q p2 v4 (stem down))
19:27:11: V:4 added pattern: (n f2 q p2 v4 (stem down)) ***
19:27:11: SetLmNote 0 100 OK, total LmNotes:4
19:27:11:     Ej:3 V:4 NEGRO
19:27:11: FINAL CHORD 0: 5ª disminuida, 2ª inversión, Bass:f, Root:b, Intervals:a4(18) M6(29)
Pattern: f3 b3 d4; Notes: f2 b2 f4 d5
19:27:11: ====== START WITH CHORD 1 ======
19:27:11: Creating_lmScoreChord: step:4 octave:2 inversions:1 key:0
19:27:11: FPitchStepsInterval (Step 6 oct:0) 38 - (Step 4, octave 0) 26 = 12
19:27:11: FPitchStepsInterval (Step 1 oct:1) 49 - (Step 4, octave 0) 26 = 23
19:27:11: Bass voice V4,FINAL: 118 (b2)
19:27:11: <><> CHORD 1: bass STEP:4, octave:2, key:0 inversions:1 ROOT:106 (b2) ===
19:27:11: lmChord: [ Perfecta mayor, 1ª inversión, Bass:b, Root:g, Intervals:m3(11) m6(28)
Pattern: b2 d3 g3; Notes: b2]
19:27:11:     nIntvB 1 2 0
19:27:11:     V3, after applying interval 1 : 129 (d3), [ni:0]
19:27:11: Raise Tenor: added octave to voice V3: 169
19:27:11: V3, FINAL: 169 (d4)
19:27:11:     V2, after applying interval 2 : 146 (g3), [ni:1]
19:27:11: Added octave to voice V2: 186
19:27:11: V2, FINAL: 186 (g4)
19:27:11:     V1, after applying interval 0 : 118 (b2), [ni:2]
19:27:11: Added octave to voice V1: 158
19:27:11: Added octave to voice V1: 198
19:27:11: V1, FINAL: 198 (b4)
19:27:11: ###BEFORE 0 ANALYSIS OF CHORD LINK ERRORS OF CHORD 1: Perfecta mayor, 1ª inversión,
Bass:b, Root:g, Intervals:m3(11) m6(28) Pattern: b2 d3 g3; Notes: b2 d4 g4 b4
19:27:11: *** AnalyzeHarmonicProgression N:2
19:27:11: Chord 0 to analyze: 5ª disminuida, 2ª inversión, Bass:f, Root:b, Intervals:a4(18)
M6(29) Pattern: f3 b3 d4; Notes: f2 b2 f4 d5
19:27:11: Chord 1 to analyze: Perfecta mayor, 1ª inversión, Bass:b, Root:g, Intervals:m3(11)
m6(28) Pattern: b2 d3 g3; Notes: b2 d4 g4 b4
19:27:11: Evaluating rule 2, description: No parallel motion of perfect octaves, perfect fifths,
and unisons
19:27:11: Check chord 2
19:27:11: Rule 2 final error count 0
19:27:11:     Total error count after rule 2: 0 errors
19:27:11: Evaluating rule 3, description: No resulting fifths and octaves
19:27:11: Check chords 0 TO 1
19:27:11: Notes: b2-->d4 f2-->b2 Movement type:Direct INTERVAL:10 (m10)
19:27:11: Notes: f4-->g4 f2-->b2 Movement type:Direct INTERVAL:13 (m13)
19:27:11: Notes: f4-->g4 b2-->d4 Movement type:Direct INTERVAL:4 (p4)
19:27:11: Notes: d5-->b4 f2-->b2 Movement type:Contrary INTERVAL:15 (p15)
19:27:11: Notes: d5-->b4 b2-->d4 Movement type:Contrary INTERVAL:6 (M6)
19:27:11: Notes: d5-->b4 f4-->g4 Movement type:Contrary INTERVAL:3 (M3)
19:27:11: Rule 3 final error count 0
19:27:11:     Total error count after rule 3: 0 errors
```

```

19:27:11: Evaluating rule 8, description: Voices interval not greater than one octave (except bass-tenor)
19:27:11: Check chord 0
19:27:11: Rule not applicable: not root position: 2 inversions
19:27:11: Check chord 1
19:27:11: Rule not applicable: not root position: 1 inversions
19:27:11: Rule 8 final error count 0
19:27:11: Total error count after rule 8: 0 errors
19:27:11: Evaluating rule 9, description: Do not allow voices crossing. No duplicates (only for root position and root duplicated)
19:27:11: Rule not applicable: not root position: 2 inversions
19:27:11: Rule not applicable: not root position: 1 inversions
19:27:11: Rule 9 final error count 0
19:27:11: Total error count after rule 9: 0 errors
19:27:11: ##RESULT: 0 LINK ERRORS
19:27:11: <<<< CHORD 1 OK!!! after 0 attempts >>>>>>
19:27:11: FIGURED BASS:6
19:27:11: Staff 1, V1 (index:0) 198 [b4], pattern: (n b4 q p1 v1 (stem up))
19:27:11: V:1 added pattern: (n b4 q p1 v1 (stem up)) ***
19:27:11: SetLmNote 3 198 OK, total LmNotes:1
19:27:11: Ej:3 V:1 NEGRO
19:27:11: Staff 1, V2 (index:1) 186 [g4], pattern: (n g4 q p1 v2 (stem down))
19:27:11: V:2 added pattern: (n g4 q p1 v2 (stem down)) ***
19:27:11: SetLmNote 2 186 OK, total LmNotes:2
19:27:11: Ej:3 V:2 NEGRO
19:27:11: Staff 2, V3 (index:2) 169 [d4], pattern: (n d4 q p2 v3 (stem up))
19:27:11: V:3 added pattern: (n d4 q p2 v3 (stem up)) ***
19:27:11: SetLmNote 1 169 OK, total LmNotes:3
19:27:11: Ej:3 V:3 NEGRO
19:27:11: Staff 2, V4 (index:3) 118 [b2], pattern: (n b2 q p2 v4 (stem down))
19:27:11: V:4 added pattern: (n b2 q p2 v4 (stem down)) ***
19:27:11: SetLmNote 0 118 OK, total LmNotes:4
19:27:11: Ej:3 V:4 NEGRO
19:27:11: FINAL_CHORD 1: Perfecta mayor, 1ª inversión, Bass:b, Root:g, Intervals:m3(11) m6(28)
Pattern: b2 d3 g3; Notes: b2 d4 g4 b4
19:27:11: ===== START WITH CHORD 2 =====
....
```

5.1.5 Estructura genérica para representar reglas

Como se ha comentado anteriormente, para aplicar las reglas de análisis de la progresión armónica se precisaba un mecanismo muy flexible que permitiese crear fácilmente nuevas reglas, deshabilitarlas o eliminarlas.

Se ha buscado una forma de implementar este gestor genérico de reglas que resulte sencilla de usar y de comprender. El resultado se describe en los siguientes párrafos.

Clase genérica (*virtual*) para gestionar reglas:

```

class lmRule
{
public:
    lmRule(int nRuleID);
    virtual ~lmRule() {};
    virtual int Evaluate(wxString& sResultDetails, int pNumFailuresInChord[],
ChordInfoBox* pBox )=0;
    bool IsEnabled();
    void Enable( bool fVal );
    int GetRuleId() {};
};
```

Macros para facilitar la creación rápida de reglas:

```
#define LM_CREATE_CHORD_RULE(classname, id) \
class classname : public lmRule \
{ \
public: \
    classname() : lmRule(id) {}; \
    int Evaluate(wxString& sResultDetails, int pNumFailuresInChord[], ChordInfoBox* \
pBox); \
};
```

Clase para agrupar las reglas en listas:

```
class lmRuleList
{
public:
    lmRuleList(lmScoreChord** pChD, int nNumChords);
    ~lmRuleList();
    bool AddRule(lmRule* pNewRule, const wxString& sDescription );
    bool DeleteRule(int nRuleId);
    lmRule* GetRule(int nRuleId);
protected:
    void CreateRules();
    std::map<int, lmRule*> m_Rules;
};

lmRuleList::~lmRuleList()
{
    // Iterate over the map and delete lmRule
    std::map<int, lmRule*>::iterator it;
    for(it = m_Rules.begin(); it != m_Rules.end(); ++it)
    {
        delete it->second;
    }
    m_Rules.clear();
}

bool lmRuleList::AddRule(lmRule* pNewRule, const wxString& sDescription )
{
    int nRuleId = pNewRule->GetRuleId();
    pNewRule->SetDescription(sDescription);
    std::map<int, lmRule*>::iterator it = m_Rules.find(nRuleId);
    if(it != m_Rules.end())
    {
        return false;
    }
    if ( nRuleId >= lmCVR_FirstChordValidationRule && nRuleId <=
lmCVR_LastChordValidationRule)
    {
        m_Rules.insert(std::pair<int, lmRule*>(nRuleId, pNewRule));
    }
    return true;
}
```

Forma de crear reglas (requiere simplemente implementar el método *Evaluate*):

```

//  

// Add rules  

//  

LM_CREATE_CHORD_RULE(lmRuleNoParallelMotion, lmCVR_NoParallelMotion)  

LM_CREATE_CHORD_RULE(lmRuleNoResultingFifthOctaves, lmCVR_NoResultingFifthOctaves)  

LM_CREATE_CHORD_RULE(lmRuleNoVoicesCrossing, lmCVR_NoVoicesCrossing)  

LM_CREATE_CHORD_RULE(lmNoIntervalHigherThanOctave, lmCVR_NoIntervalHigherThanOctave)  

// return number of errors  

int lmRuleNoParallelMotion::Evaluate(wxString& sResultDetails, int  

pNumFailuresInChord[], ChordInfoBox* pBox )  

{  

    // ...
}

```

Forma de aplicar las reglas: simplemente llamando al método *Evaluate*.

```

for (nR .... )  

{  

    pRule = tRules.GetRule(nR);  

    if ( pRule != NULL && pRule->IsEnabled())  

    {  

        nNumErrors += pRule->Evaluate(sStr, &nNumChordError[0], pChordErrorBox);  

    }
}

```

5.2 Lecciones aprendidas

5.2.1 Estrategia para el procesamiento de la Armonía Tonal

Además de la funcionalidad del analizador de armonía que se ha integrado en LenMus, uno de los frutos más destacados de este trabajo es precisamente el conocimiento adquirido sobre el procesamiento de la armonía musical de una partitura. Esta experiencia se traduce en una serie de directrices o consejos genéricos, aplicables a cualquier intento de procesar acordes desde una notación musical.

No se consideran los casos de intentar procesar los acordes desde un formato de *bajo nivel*; esto es, por debajo del nivel simbólico del Lenguaje Musical, pues para eso ya existe mucho material teórico y, además, como ya se ha explicado, no parece una buena idea abordar el procesamiento de acordes desde niveles de representación inferiores.

El seguimiento de estas sencillas directrices puede ayudar a simplificar significativamente la complicada tarea del procesamiento de la armonía musical.

La información debe basarse en los intervalos

Se observa que los libros de teoría de la armonía suelen definir a los acordes como *grupos de notas simultáneas*, y esto, aun sin ser falso, representa una visión demasiado simplista que induce a error, pues son los intervalos, y no las notas, los que deciden la *consonancia* y la *disonancia* y, por tanto, la armonía.

Si la estrategia se enfoca a las notas en lugar de a los intervalos, se arrastrará innecesariamente toda la *complejidad contextual* de las notas, principalmente la *tonalidad* y las *alteraciones*. Además, se acabará duplicando información, pues cuando se precisen los datos de los intervalos, se guardarán junto con los de las notas, lo que supone una redundancia innecesaria y una complejidad costosa de mantener.

Con una metáfora química, para la armonía los *intervalos* serían como los *átomos*, y los *acordes* como las *moléculas*.

Al buscar acordes a partir de notas, es esencial comprender la preponderancia de los intervalos con respecto a notas. Los intervalos deben ser los datos sobre los que basar el trabajo de

procesamiento, mientras que las notas deben ser consideradas simplemente como propiedades accesorias de los intervalos.

Respetar niveles de abstracción de los acordes

Durante el proceso de obtención de los acordes, estos van pasando por sucesivas etapas o niveles de abstracción. Es importante darse cuenta de esos niveles y respetarlos, usando una clase distinta en cada nivel.

Es importante entender los *niveles de abstracción* de un acorde y reflejarlos en la arquitectura de *clases* o *estructuras de datos* que se usen para procesarlo. En particular, hay que diferenciar bien entre los siguientes tipos o estados o clases de acordes:

1. Grupo de intervalos: diferencia de cada nota respecto a la más baja.
2. Grupo *normalizado* de intervalos: reducir a la base de una octava, eliminar duplicados y ordenar.
3. Acorde *nominal*: grupo de intervalos que se corresponden con un tipo de acorde válido.
4. Acorde real: formado por notas concretas, algunas de las cuales pueden estar duplicadas, y en el contexto de una armadura tonal específica.

Recordamos cómo François Pachet en su estudio "*Musical Harmonization with Constraints: A Survey*" [PACH01], destaca la importancia de respetar los niveles de abstracción de los acordes, diferenciando entre un acorde y un grupo de notas :

" *In the preceding approaches, chords have always been treated as simple groups of notes (which they are in some sense). However, from the constraint viewpoint, this is a mistake. We have shown that by considering explicit chord variables on top of the basic note variables, one could dramatically divide the theoretical complexity of the problem*"

Atenerse a la escala cromática mientras sea posible

La escala diatónica es una distorsión llena de irregularidades superpuesta a la escala cromática, que es homogénea y simple. El Lenguaje Musical tradicional representa la escala diatónica, no la cromática, y por ello se tiende a realizar el procesamiento directamente sobre una representación la

escala diatónica, lo que supone enfrentarse a importantes dificultades como, por ejemplo, la gestión de las alteraciones y la tonalidad.

Por todo ello es aconsejable que los algoritmos de procesamiento sean independientes de la escala utilizada, y en general usar la escala cromática para realizar las operaciones de cálculos con las notas, dejando el uso de la escala diatónica para la interfaz con el usuario.

5.2.2 Entrada en notación simbólica

Probablemente más del noventa por ciento de los estudios teóricos sobre cómo procesar música - tanto análisis como síntesis - son relativos a una entrada o salida a nivel de *audio*, no de Lenguaje Musical. Pero, como ya se ha ido explicando a lo largo de este trabajo, el nivel de audio es demasiado *bajo* para representar la música, pues no permite albergar toda la información contenida en una partitura.

Aunque muchos estudiosos del tema se empeñan en seguir usando el audio como *lenguaje* para de trabajo, hay que insistir en que no es el adecuado, ya que se pierde información al extraer la notación musical, por muy sofisticados que sean los algoritmos utilizados.

Es comprensible que muchos investigadores se sientan más cómodos procesando *ondas* que procesando *música*. De esta forma tienen acceso al extenso conocimiento acumulado en el campo del *procesamiento digital de señales (DSP)* y su vasto repositorio de recetas algorítmicas, pero, sencillamente, este no es el mejor camino si se pretende usar los ordenadores para el aprendizaje de la música.

Para representar la música debe usarse un lenguaje simbólico con similar capacidad expresiva que el propio Lenguaje Musical, y por tanto de su mismo nivel.

No hay que olvidar tampoco que MIDI, aun siendo mucho más cercano a la música que el formato audio, también es insuficientemente *expresivo* como para poder representar completamente el Lenguaje Musical, lo que implica una casi segura pérdida de información si se utiliza para ello.

La experiencia en este desarrollo indica que si la representación es la adecuada, para cualquier problema de procesamiento, siempre hay una solución viable. Sin embargo, si la representación no es suficiente, enseguida aparecen limitaciones o dificultades insalvables.

Enfrentarse al procesamiento de la armonía, y en general de la música, ya es suficientemente complicado aun partiendo de datos en formato simbólico, al nivel del Lenguaje Musical; no parece

razonable complicarlo más aún partiendo de datos en un formato de bajo nivel como el audio o incluso el MIDI.

Para representar la música conviene usar una *notación musical*, no el formato **audio**

MIDI está pensado para **generar** música, no para **representar** música

Traducir de un lenguaje a otro más expresivo y de más alto nivel exige inyectar información, mientras que en sentido inverso supone perder información. Ambos aspectos suponen una complicación adicional, además de un riesgo. El esfuerzo dedicado a llenar los huecos de los traductores podrían dedicarse a mejorar los algoritmos de procesamiento de la música, que en principio es más importante.

Si el formato de bajo nivel no es una exigencia de la aplicación, conviene evitar complicaciones utilizando notaciones expresivas del alto nivel.

5.2.3 Ventajas de la notación-40 de Hewlett

El Lenguaje Musical tiene algunas particularidades *extrañas*, que lo alejan de la posibilidad de formalizarlo sin problemas. Una de las que más desconcertante resulta para una mentalidad racional es que, por motivos de compatibilidad hacia atrás- de cuando la escala estaba mal afinada o *temperada* -, se deben mantener representaciones distintas (diferentes *alteraciones*) para sonidos iguales (*enarmónicos*).

Lo *milagroso* de la notación 40 de Hewlett [HEW92], es que, además de poder representar de forma directa la altura de una nota, tanto en escala diatónica como cromática, también mantiene la *compatibilidad hacia atrás* con la escala *mal temperada*, y puede representar a los *enarmónicos* con *alteraciones* diferentes.

Esta notación permite representar los intervalos (distancia entre notas) como simples restas de los respectivos valores en notación 40. Resulta sorprendente que cualquier operación de sumas y

restas que se realice sobre notas o intervalos en la notación 40, siempre mantiene el resultado correcto en las tres escalas (diatónica, cromática y *mal temperada*).

Estas ventajas se aprecian muy especialmente al trabajar con la armonía, pues debemos reiterar de nuevo que la armonía se basa en intervalos más que en notas. Con esta notación, además de sencillez, se consigue la máxima claridad en las operaciones con intervalos.

5.2.4 Escasez de estructuras y algoritmos a nivel de Lenguaje Musical

Existe una extensa y variada bibliografía sobre el tema "música y ordenador", pero la mayor parte de ese material consiste en algoritmos y estrategias para procesar señales de audio digital. También se pueden encontrar con facilidad estudios sobre el procesamiento musical de datos MIDI, pero si lo que se buscan trabajos y productos que ayuden a procesar música representada ya en una notación de alto nivel, entonces el material de calidad escasea claramente.

En cuanto al procesamiento automatizado de la música, existe un vacío de conocimiento teórico sobre cómo trasladar el Lenguaje Musical a Lenguaje Informático. Es decir, no hay información de calidad sobre cómo abordar la creación estructuras de datos que representen los símbolos musicales de forma que resulten fácilmente procesables, ni de algoritmos, y que resuelvan tareas específicas de procesamiento musical.

Quizás el problema es que faltan mejores y más extendidas notaciones musicales, o quizás el problema es el contrario: hay demasiada variedad de notaciones, lo que provoca que ninguna se consolide. Posiblemente falten estándares para representar la música adecuadamente en todas sus cualidades o dimensiones. Posiblemente se necesite más trabajo en cuanto a taxonomías y ontologías, aunque estos avances solo abordan el aspecto de las estructuras de representación de la música, pero no los algoritmos para su procesamiento.

El hecho es que escasean las aplicaciones y librerías auxiliares para procesar la música públicamente disponibles que sean realmente sólidas, fiables y prácticas. Por otro lado, las aplicaciones existentes son muy dependientes de su contexto tecnológico, y adolecen de una clara falta de interoperatividad.

Un aspecto positivo del presente trabajo es precisamente que los avances que realiza son *consolidables*, en el sentido de que son suficientemente *abstractos* como para poder ser reutilizados en otros contextos tecnológicos. En particular, se puede destacar que los algoritmos tienen muy

escasa dependencia del lenguaje de programación, de la notación y de la representación de la música utilizada.

5.2.5 Refactorización y *arquitectura emergente*

¿Es cierto lo que postulan en XP y TDD de que la arquitectura emerge por sí misma tras codificar y refactorizar?

En nuestro caso es cierto que la arquitectura de clases para los acordes fue, en cierta medida, surgiendo, o visionándose con más claridad, en las sucesivas refactorizaciones realizadas tras codificar los incrementos. Pero, evidentemente, no surgió espontáneamente, sino tras varios importantes y atrevidos esfuerzos de rediseño.

Puede ser una lección aprendida: para conseguir ver la arquitectura correcta es preciso plantearse el rediseño seriamente, es decir, realizarlo con valentía y decisión, asumiendo que puede ser preciso realizar importantes modificaciones y eliminar buena parte del código.

En general, existe una cierta reticencia psicológica a modificar algo que funciona, y también nos cuesta eliminar completamente partes innecesarias de código. Sin embargo, es preciso superar estas resistencias internas si se quiere conseguir un diseño óptimo: sencillo, robusto, refinado y reutilizable.

Las clases tienen tendencia a engordar con constructores, métodos y miembros que se añaden para necesidades puntuales y transitorias o simplemente 'por si acaso'. Luego cuesta hacer una limpieza, tanto por las reticencias mencionadas, como por el temor a estropear lo que funciona. Para vencer estas reticencias es absolutamente aconsejable una buena batería automatizada de pruebas, que permita verificar fácilmente, tras cualquier modificación, que no se pierde o se altera la funcionalidad .

En buena medida, el éxito del rediseño depende de la confianza, y ésta, a su vez, depende de una extensa y fiable batería de pruebas automatizadas.

En nuestro caso, ocurrió que durante gran parte del proyecto se usaba solo una clase para manejar los acordes en cualquiera de sus contextos, lo que hacía que resultase cada vez más difícil crear y manejar objetos de esta clase. Cuando ya teníamos conseguida la mayor parte de la funcionalidad, nos planteamos un rediseño, limpieza y reestructuración de las clases de los

acordes. Esto resultaba ineludible, ya que se necesitaba reutilizar las clases de los acordes para otros tipos de ejercicios de armonía, y la clase única no resultaba práctica ni manejable.

De esta forma nos lanzamos durante varias semanas a un valiente rediseño que sin duda mereció la pena, ya que fue entonces cuando nos 'emergió' la arquitectura y distinguimos con claridad la arquitectura de las clases de los acordes.

Al verlo ahora con la perspectiva del proyecto finalizado, hemos aprendido algunas aspectos, como que no conviene esperar tanto como hicimos nosotros para hacer el rediseño. Se debe plantear hacerlo regularmente. No se debe posponer mucho, porque se corre el riesgo de que resulte demasiado complicado.

Otra lección que aprendimos fue la importancia de disponer de una batería de pruebas suficientemente completa como para dar la confianza de que todo sigue funcionando correctamente tras cualquier cambio. La importancia de las pruebas la fuimos valorando más a medida que avanzábamos, pero en un principio estábamos demasiado concentrados en buscar los algoritmos adecuados y quizás descuidamos algo respaldar el desarrollo con suficientes pruebas. En este sentido, hemos aprendido también que es muy aconsejable seguir el planteamiento propuesto por TDD de que las pruebas deben crearse antes de comenzar a diseñar o codificar.

6 Mejoras y extensiones

El Analizador Tonal desarrollado aún se sitúa en la etapa de prototipo. Para consolidarlo, sea dentro de LenMus o como utilidad independiente, se precisa un cierto trabajo de refinamiento, tanto en su funcionalidad como en su interfaz.

Para independizarlo de LenMus, simplemente se precisa extraer la parte de visualización de mensajes, que por razones prácticas se basa en objetos TextBox específicos de LenMus. Aparte de este cambio, hay muchas otras posibilidades de mejora del Analizador Tonal, como las que se mencionan a continuación.

6.1 Extensiones recomendables

6.1.1 Extensiones funcionales

Las siguientes extensiones representan mejoras notables en cuanto a la funcionalidad necesaria para el aprendizaje de la armonía y resultarían relativamente poco costosas de implementar:

- Ampliar el número de reglas de progresión armónica. Actualmente solo están implementadas cuatro reglas.
- Completar la funcionalidad de las *elisiones*; actualmente solo parcialmente implementada
- Detectar y analizar el ajuste de los acordes a los acentos.
- Detectar la *tonalidad* y sus cambios (*modulaciones*).
- Reconocer y analizar las notas *ajenas* a los acordes: *notas de paso, apoyaturas, anticipaciones, retardos, bordaduras y escapadas*.
- Reconocer *cadencias*.
- Ampliar las notaciones de cifrado.

6.1.2 Mejoras en los ejercicios

- Implementar la ventana de opciones, para permitir adaptar los ejercicios a las necesidades y deseos de alumnos y profesores. Entre las opciones posibles para los ejercicios se pueden mencionar las siguientes:

- Permitir variar la tonalidad (actualmente fija en Do Mayor)
- Elegir si mostrar o no los grados de los acordes
- Elegir si se permiten las inversiones
- Elegir si se permite las elisiones
- Permitir forzar algunas características de los acordes, como grado, tipo o inversiones

6.1.3 Mejoras en la visualización de información

- Permitir mediante un parámetro la opción de mostrar la información completa de cada acorde, que en la actualizada solo se muestra en ciertos casos de error, ya que puede resultar muy útil para alumnos principiantes y no cuesta nada obtenerla. Otra posibilidad aún mejor sería que, al pasar el ratón sobre un acorde, apareciese un texto de ayuda de tipo *tooltip*, mostrando la descripción del mismo.
- En general, se puede mejorar mucho la forma de mostrar los mensajes: por ejemplo, se podría establecer una ventana específica para ello, o bien mejorar el algoritmo para la ubicación de mensajes en la partitura con objeto de evitar solapamientos.

6.1.4 Mejoras técnicas

- Modificar el mecanismo genérico de reglas, actualmente basado en macros de C, y cambiarlo por uno más moderno y potente que utilice plantillas de C++ (*templates*).
- Sería deseable automatizar la comprobación de los errores detectados al aplicarse el analizador sobre una partitura concreta. De esta forma se podría mejorar fiabilidad y la efectividad de las pruebas de integración.

6.2 Variantes experimentales

Se describen algunas ideas basadas en el analizador desarrollado sobre las que se podrían crear variantes o nuevos desarrollos, pero que suponen objetivos y técnicas muy distintos. Son simples sugerencias y no se entra en analizar su viabilidad o coste.

6.2.1 Aplicar técnicas de *Inteligencia Artificial*

- Ampliar la inteligencia del analizador mediante un módulo capaz de evaluar la armonía implícita de una melodía, es decir, la adecuación entre una melodía y su armonía.
- Clasificar y valorar los errores. Crear perfiles de alumno en función de los errores.
- Tras corregir errores, sugerir posibles alternativas
- Realimentación: adaptar ejercicios en función de los resultados
- Reconocimiento de estilo musical en función de los acordes
- Armonías de culturas distintas a la occidental
- Generador automatizado de armonías a partir de melodía

6.2.2 Adaptación del analizador de armonía para funcionamiento como servidor Web

Se describen algunos esbozos de ideas sobre la posibilidad de crear una variante del analizador de armonía que funcione sobre un servidor web.

El analizador funcionaría como un servidor web. Aceptaría como entrada una partitura en alguna notación textual, al menos LDP. La analizaría y enviaría los resultados como una página HTML con los mensajes.

Mejora: si además se le añade el renderizador de partituras de LenMus, podría retornar una imagen con la partitura generada y los errores, al estilo LenMus.

Mejora: si se le añadiese un *plug-in* al navegador para poder representar LDP, usando el renderizador de LenMus, podría retornar la partitura en LDP que incluiría los mensajes de error.

Si se añadiese un *plug-in* al navegador para visualizar libros en el formato XML de LenMus, y se convirtiesen los objetos de ejercicios LenMus en applets o ActiveX, se podría usar LenMus directamente en el navegador, y abriría la posibilidad de crear un repositorio colaborativo de libros y ejercicios.

7 Bibliografía

7.1 Bibliografía citada

- [AAM06] Víctor Barbero, Carmen Carrión, Álvaro de los Reyes. "Analizador de armonía musical. Proyecto de la Universidad Complutense de Madrid", 2006.
- [ABA07] Federico Abad. "¿Do, Re, Qué? Guía práctica de iniciación al lenguaje musical", Berenice , 2007.
- [BECK03] Kent Beck. "Test-Driven Development: By Example", Addison-Wesley, 2003.
- [BLE10] Carlos Blé Jurado y colaboradores. "Diseño Ágil con TDD", SafeCreative, 2010.
- [BOT03] Jesús G. Boticario y Elena Gaudioso. "Sistemas Interactivos de Enseñanza/Aprendizaje", Sanz y Torres, 2003.
- [BRA99] Marcio Brandao; Geraint Wiggins and Helen Pain. "Computers in Music Education", Division of Informatics, University of Edinburgh, 1999.
- [COR02] Vanesa Cordantonopoulos. "Curso completo de Teoría de la Música", www.lapalanca.com, 2002.
- [GON04] Jesús González Barahona. "Software libre y educación", 2004, <http://www.miescuelayelmundo.org/IMG/pdf/transpas-2up.pdf>.
- [HAR02] Christopher Harte, Mark Sandler and Samer Abdallah, Emilia Gómez. "Symbolic representation of musical chords: a proposed syntax for text annotations", 2002.
- [HEW92] Walter B Hewlett . "A base-40 number-line representation of musical pitch notation", Stanford University of California, 1992.
- [HOL89] Simon Holland. "Artificial Intelligence, Education and Music. The Use of Artificial Intelligence to Encourage and Facilitate Music Composition by Novices", 1989.
- [JUR04] Bryan Jurish."Music as a formal language", Universität Potsdam, Institut für Linguistik, 2004.
- [LAP01] Horacio Alberto Lapidus. "Modalidades de Realimentación en Software de Asistencia al Aprendizaje de la Armonización de Melodías Tonales", Universidad de Los Andes , 2001.
- [MAT06] Miguel Ángel Mateu. "Armonía práctica", AB Música, 2006.
- [MIRA03] J. Mira, A.E. Delgado, J.G. Boticario, F.J. Diez. "Aspectos básicos de la Inteligencia Artificial", Sanz y Torres, 2003.

- [MUK05] Singh, Mukbir. "Collaborative learning and technology: Moving from Instructionism to Constructionism", 2005.
- [ORI06] Nicola Orio. "Music Retrieval: A Tutorial and Review", University of Padova, 2006.
- [PACH01] François Pachet, Pierre Roy. "Musical Harmonization with Constraints: A Survey", 2001.
- [PAR99] Bryan Pardo, William P. Birmingham. "Automated partitioning of tonal music. cap 5: HarmAn, a system that partitions tonal music into harmonically significant segments and labels these segments with the proper chord labels", University of Michigan, 1999.
- [RAM22] J. P. Rameau. "Tratado de Armonía reducida a sus principios naturales", 1722.
- [RYY06] Matti Ryyränen. "Transcription of the Singing Melody in Polyphonic Music", 2006, http://www.cs.tut.fi/sgn/arg/matti/ryynanen_ismir06.pdf
- [SAY95] Seymour Papert. "La máquina de los niños. Replantearse la educación en la era de los ordenadores", 1995.
- [SCH04] Ken Schwaber. "Agile Project Management with Scrum", Microsoft Press, 2004.
- [SEL97] Eleanor Selfridge-Field. "Beyond MIDI. The Handbook of Musical Codes", The MIT Press, 1997.
- [TAK03] Haruto Takeda, Naoki Saito. "Hidden Markov Model for Automatic Transcription of MIDI Signals", University of Tokyo, 2003
<http://ismir2003.ismir.net/papers/Takeda.PDF>
- [TEJ99] Jesús Tejada Giménez. "Editores de partituras y contenidos de armonía, arreglo y composición en la formación de maestros especialistas de música", Boletim da Associaçao Portuguesa de Educaçao Musical, Num.101,1999, Pag.10-14, <http://tecnologiaedu.us.es/bibliovir/pdf/11.pdf>

7.2 Bibliografía de referencia sobre enseñanza de música con ordenador

A continuación se detallan referencias bibliográficas sobre enseñanza de música asistida por ordenador y procesamiento automatizado de la música. Se han resaltado los que hacen referencia a la enseñanza de la armonía musical.

- B. Alphonse.** Music analysis by computer. *ComputerMusic Journal*, 4(2):26–35, 1980.
- M. Baker.** An artificial intelligence approach to musical grouping analysis. *Contemporary Music Review*, 3:43– 68, 1989a.
- M. Baker.** A computational approach to modeling musical grouping structure. *Contemporary Music Review*, 3:311–325, 1989b.
- M. Baker.** Design of an intelligent tutoring system for musical structure and interpretation. In K. E. M. Balaban and O. Laske, editors, *Understanding Music with AI: Perspectives on Music Cognition*, pages 467–489. The MIT Press, Cambridge, MA, 1992.
- G. J. Balzano.** The group-theoretic description of 12-fold and microtonal pitch systems. *Computer Music Journal*, 4(4):66–84, 1980.
- J. Bamberger.** Progress report: Logo music project. Technical report, A.I. Laboratory, Massachussets Institute of Technology, 1974.
- J. Bamberger.** *The Mind Behind the Musical Ear*. Harvard University Press, Cambridge, Massachussets, 1991.
- I. Bent.** *The New Grove Handbooks in Music: ANALYSIS*. The Macmillan Press Ltd, London, 1987. With a glossary by W. Drebkin.
- W. Berz W. , J. BOWMAN.** Applications of Research in Music Technology. *Music Educators National Conference*, Reston, VA, EEUU, 1994.
- Marcio Brandao.** Computers in Music Education, 1999. Division of Informatics, University of Edinburgh.
- A. Blombach.** An introductory course in computer assisted music analysis: The computer and Bach chorales. *Journal of Computer-based Instruction*, 7(3), 1981.

H. L. Burns and C. G. Capps. Foundations of intelligent tutoring systems: An introduction. In M. C. Polson and J. J. Richardson, editors, *Foundations of Intelligent Tutoring Systems*, pages 1–19. Lawrence Erlbaum Associates, New Jersey, 1988.

G. Cabra . Real Time Automatic Harmonisation (in French). PhD. Thesis, University of Paris 6, Paris, France, July 2008.

E. Camboropoulos. Towards a General Computational Strcture of Musical Structure. Ph.D. thesis, Faculty of Music, University of Edinburgh, Edinburgh, 1998.

E. Camboropoulos. A General Pitch Interval Representations: Theory and Applications. *Journal of New Music Research*, Vol. 25, pp. 231-251. 1996.

W. Chai. Automated Analysis of Musical Structure, PhD Thesis, Massachusetts Institute of Technology, MA, USA, September 2005.

J. Cook. Agent reflection in an intelligent learning environment architecture for musical composition. In M. Smith, A. Smaill, and G. Wiggins, editors, *Music Education: An Artificial Intelligence Approach, Workshops in Computing*, pages 3–23. Springer-Verlag, 1994. Proceedings of a workshop held as part of AI ED 93, Edinburgh, Scotland, 25 August 1993.

J. Cook. Knowledge Mentoring as a Framework for Designing Agents for Supporting Musical Composition Learning. Ph.D. thesis, Computing Department, Open University, UK, 1998a.

J. Cook. Mentoring, metacognition and music: Interaction analyses and implications for intelligent learning environments. *International Journal of Artificial Intelligence and Education*, 9:45–87, 1998b.

N. Cook. A guide to musical analysis. J. M. Dent and Sons Ltd, London, 1987.

R. Dannenberg, B. Thom, D. Watson. A Machine Learning Approach to Musical Style Recognition. School of Computer Science, Carnegie Mel lon University. 1997.

R. Dannenberg, M. Sanchez, A. Joseph, P. Capeli, R. Joseph, and R. Saul. A computer-based multi-media tutor for beginning piano students. *Interface*, 19(2-3):155–173, 1990.

J. Dempsey, G. Sales. Interactive Instruction and Feedback. Educational Technology Publications, New Jersey, 1993.

P. Desain and H. Honing. Loco: Composition microworlds in logo. In Proceedings of the 1986 International Computer Music Conference. The Hague, Netherlands, 1986.

K. Ebcioğlu. An Expert System for Harmonizing Four-part Chorales, Computer Music Journal, vol.12, no. 3,43-51 (1988).

Eigenfeldt, A., & Pasquier, P. (n.d.). Realtime Generation of Harmonic Progressions Using Constrained Markov Selection. Computer Music Journal

D.Fober, S.Letz, Y.Orlarey. Open source tools for music representation and notation. Grame - Centre national de creation musicale. 2004.

G. Gargarian. Towards a constructionist musicology. In I. Harel and S. Papert, editors, Constructionism, pages 311–333. Ablex Publishing Company, Norwood, NJ, 1993.

G. Greenberg. Music learning - compositional thinking. In Proceedings of the 1988 International Computer Music Conference, pages 150–157. Cologne, West Germany, 1988.

D. Gross. A Set of Computer Programs to Aid in Music Analysis. Ph.D. thesis, Indiana University, 1975.

D. Gross. Computer applications to music theory: a retrospective. Computer Music Journal, 8(4):35–42, 1984.

Hanlon, M., & Ledlie, T. (2002). CPU Bach : An Automatic Chorale Harmonization System. Bach, 1-8.

W. B. Hewlett, A Base-40 Number-line representation of Musical Pitch Notation, Stanford, California. <http://www.ccarh.org/publications/reprints/base40/>

F. Hoffstetter. GUIDO: an interactive computer-based system for improvement of instruction and research in ear-training. Journal of Computer-Based Instruction, 1(4):100–106, 1975.

F. Hofstetter. Computer-based aural training: The GUIDO system. Journal of Computer-Based Instruction, 7(3):84–92, 1981.

F. T. Hofstetter. Computer Literacy for Musicians. Prentice Hall, Englewood Cliffs, NJ, 1988.

S. Holland. Design considerations for a human-computer interface using 12-tone three dimensional Harmony Space to aid novices to learn aspects of harmony and composition. CITE Report No. 7, Open University, Milton Keynes., 1986.

S. Holland. Artificial Intelligence, Education and Music. The Use of Artificial Intelligence to Encourage and Facilitate Music Composition by Novices. Ph.D. thesis, IET - Open University, UK, 1989.

S. Holland. Architecture of a knowledge-based music tutor. In Guided Discovery Tutoring, Ed. Elsom-Cook, M. Paul Chapman Publishing Ltd, London. 1990.

S. Holland. Learning about harmony with harmony space: an overview. In M. Smith, A. Smaill, and G. Wiggins, editors, Music Education: An Artificial Intelligence Approach, Workshops in Computing, pages 25–40. Springer-Verlag, 1994. Proceedings of a workshop held as part of AI-ED 93, Edinburgh, Scotland, 25 August 1993.

S. Holland and M. Elsom-Cook. Architecture of a knowledge-based music tutor. In M. Elsom-Cook, editor, Guided Discovery Tutoring: A Framework for ICAI Research. Paul Chapman Publishing, Ltd., 1990.

ICMC 2000, Berlin: Workshop on Notation and Music Information Retrieval in the Computer Age.

R., Jackendoff, F. Lerdahl, F. A Deep Parallel Between Music and Language. 1980.

L. Kyogu. A System for Acoustic Chord Transcription and Key Extraction from Audio Using Hidden Markov Models Trained on Synthesized Audio. PhD. Thesis, Stanford University, CA, USA, March 2008.

T. Kirshbaum. Using a touch-table as an effective, low-cost input device in a melodic dictation program. *Journal of Computer-Based Instruction*, 13(1):14–16, 1986

T. Kohonen. A Self-Learning Musical Grammar, or Associative Memory of the Second Kind. Helsinki University of Technology Laboratory of Computer and Information Science Rakentajanaukio 2 C, SF-02150 Espoo, Finland

T. Kohonen. Self-Learning Inference Rules by Dynamically Expanding Context, Proc. of the IEEE First Annual International Conference on Neural Networks, San Diego, Cal., June 21-24, 1987,

M. Lamb. An interactive graphical modelling game for teaching musical concepts. *Journal of Computer-Based Instruction*, 1982. Autumn 1982.

M. Lamb and V. Buckley. A user-interface for teaching piano keyboard techniques. In B. Schackel, editor, *In teract*. 1985.

A. LeBlanc, Y. C. Jin, M. Obert, and C. Siivola. Effect of audience on music performance anxiety. *Journal of Research in Music Education*, 45(3):480–496, 1997.

F. Lerdahl and R. Jackendoff. A Generative Theory of Tonal Music. The MIT Press, Cambridge, Massachusetts, 1983.

- D. A. Levitt.** A Representation of Musical Dialects. Un published PhD thesis, Department of Electrical Engineering and Computer Science, Massachussets Institute of Technology, 1985.
- Lopez de Mantaras, R., & Arcos, J. L.** (2002). AI and Music From Composition to. AI Magazine, 43-58.
- R. Monelle.** Linguistics and Semiotics in Music. Harwood academic publishers, U.K., 1992.
- MTNA.** The Music Teachers National Association Guide to Music Instruction Software. Cincinnati, 1996.
- E. Narmour.** The Analysis and Cognition of Basic Melodic Structures. University of Chicago Press, London, 1990. S. R. Newcomb. LASSO: An intelligent computer-based tutorial in sixteenth-century counterpoint. Computer Music Journal, 9(4):49–61, 1985.
- B. Pardo, W. Birmingham.** Automated partitioning of tonal music, Chapter 5: HarmAn, a system that partitions tonal music into harmonically significant segments and labels these segments with the proper chord labels. Technical report, University of Michigan, 1999.
- Publisher, K., & Voluceau, D. D.** (2001). Musical Harmonization with Constraints : A Survey. Constraints, 6(1).
- C. Robbie.** Implementing a Generative Grammar for Music. MSc thesis, Department of Artificial Intelligence, University of Edinburgh, 80 South Bridge, Edinburgh, 1994.
- C. Rhodes, D. Lewis, D. Müllensiefen.** Bayesian model selection for Harmonic Labelling , University of London, 2008
- B. F. Skinner.** Why we need teaching machines. Harvard Educational Review, 31:377–398, 1961.
- J. A. Sloboda.** The Musical Mind: The Cognitive Psychology of Music. Oxford University Press, New York, 1985.
- M. Smith.** Towards an Intelligent Learning Environment for Melody Composition Through Formalisation of Narmour’s Implication-Realisation Model. Ph.D. thesis, IET, Open University, 1995.
- M. Smith and S. Holland.** MOTIVE - the development of an AI tool for beginning melody composers. In M. Smith, A. Smaill, and G. Wiggins, editors, Music Education: An Artificial Intelligence Approach, Work shops in Computing, pages 41–55. Springer-Verlag, 1994. Proceedings of a workshop held as part of AI ED 93, Edinburgh, Scotland, 25 August 1993.

S. Smoliar. A Parallel Processing Model of Musical Structures. PhD thesis, Massachussets Institute of Technology, 1971.

S. Smoliar. A computer aid for Schenkerian analysis. *Computer Music Journal*, 4(2):41–59, 1980.

S. W. Smoliar, J. A. Waterworth, and P. R. Kellock. pianoforte: A system for piano education beyond notation literacy. In *Proceedings of ACM Multimedia 95*, pages 457–465. San Francisco, CA, 1995.

L. Sorisio. Design of an intelligent tutoring system in harmony. In *Proceedings of the 1987 International Computer Music Conference*, pages 356–363. Urbana, IL, 1987.

C. Stuart Sapp. Computational Chord-Root Identification in Symbolic Musical Data: Rationale, Methods, and Applications. Center for the History and Analysis of Recorded Music. University of London. 2007.

K. Swanwick. A Basis for Music Education. NFER Publishing, London, 1979. M. Thomas. Vivace: A rule-based AI system for composition. In *Proceedings of the 1985 International Computer Music Conference*, pages 267–274. Vancouver, British Columbia, 1985.

J. C. Tobias. Knowledge representation in the harmony intelligent tutoring system. In *Proceedings of the First Workshop on Artificial Intelligence and Music*, pages 112–124. Minneapolis, Minnesota, 1988.

J. Tejada, C. Angulo. El ordenador y las nuevas tecnologías en la enseñanza de la música. *Revista Música y educación*. , ISSN 0214-4786, <http://www.musicalis.es/>, Nº 13, 1993, pags. 49-54.

J. Tejada, C. Angulo. El ordenador y las nuevas tecnologías en la enseñanza de la música: La comunicación de los instrumentos musicales electrónicos entre sí y con el ordenador: Midi y dispositivos. *Revista Música y educación*, Nº 14, 1993, pags. 33-48.

J. Tejada, C. Angulo. El ordenador y las nuevas tecnologías en la enseñanza de la música. (III) Ordenadores y sus aplicaciones musicales (Software). *Revista Música y educación*, Nº 15, 1993, pags. 49-74.

J. Tejada. Sistemas multimedia y adiestramiento tecnológico. *Revista Música y educación*, Nº 35.

J. Tejada. Editores de partituras y contenidos de armonía, arreglo y composición en la formación de especialista de música. Una experiencia práctica. *Boletín de la Asociación Española de Documentación Musical*, vol. 6, nº 1. pp. 25-7

- J. Tejada.** Tres modelos cognitivos en la tecnología educativa. Revista Bolteim da Associaçao Portuguesa de Educaçao Musical, Nº 101. 1999. <http://tecnologiaedu.us.es/bibliovir/pdf/11.pdf>.
- J. Tejada.** Software musicale e formazione: un'esperienza dalla Spagna. Musica Domani, 111. pp. 18-21. 1999
- J. Tejada.** Editores de partituras y elaboración de documentación técnica para el autoaprendizaje. Boletín de la Asociación Española de Documentación Musical, vol. 6, nº 1. pp. 25-77. 1999.
- J. Tejada.** Manual impreso minimalista versus manual hipermedia: contraste empírico de dos tipos de materiales de adiestramiento informático de un editor de partituras para usuarios inexpertos. Revista electrónica de LEEME , 4. . acceso doc. 1999.
- J. Tejada.** Programas de captura, edición y reproducción de audio (4 partes). Música y Educación, 42. pp.121. Música y Educación, 43. pp.75-76. Música y Educación, 44. pp.123-124. Música y Educación, 45. pp. 88-100. 2000.
- J. Tejada.** El adiestramiento auditivo y la teoría de la música en los programas informáticos. Música y Educación, 47. pp. 101-106. 2001.
- J. Tejada.** Tecnología musical y educación musical en la escuela obligatoria, Música y Educación, 53. pp. 97-100. 2003.
- J. Tejada.** Construcción de materiales audiovisuales de aprendizaje instrumental (4 partes). Música y Educación, 55. pp. 107-111. Música y Educación, 56. pp. 151-156. Música y Educación, 57. pp. 167-169. Música y Educación, 59. pp. 107-112. 2003.
- J. Tejada.** Música y mediación de la tecnología en sus procesos de aprendizaje. Educación XXI, 7, pp. 15-26. 2005.
- J. Tejada.** Procesos musicales creativos y tecnología en Ed. Secundaria. Música y Educación, 62. pp. 115-120. 2005.
- B. Temperley.** The Cognition of Basic Musical Structures, The MIT Press, 2001
- Yi, L.** Automatic Generation of Four-part Harmony. 1970
- Yogev, N., & Lerch, A. (2008).** A System for Automatic Audio Harmonization
- Y. Yoshinori and K. Nagaoka.** Computerised methods for evaluating musical performances and for providing instruction techniques for keyboard instruments. Computing and Education, 9(2), 1985.

8 Anexos

8.1 Código fuente

Se adjunta el contenido de los ficheros de código fuente que forman parte del desarrollo de es analizador tonal.

8.1.1 Processor.cpp

RESUMEN: Analizador de armonía.

URL:

<https://lenmus.svn.sourceforge.net/svnroot/lenmus/trunk/src/app/Processor.cpp>

```
-----
// LenMus Phonascus: The teacher of music
// Copyright (c) 2002-2010 LenMus project
//
// This program is free software; you can redistribute it and/or modify it under the
// terms of the GNU General Public License as published by the Free Software Foundation,
// either version 3 of the License, or (at your option) any later version.
//
// This program is distributed in the hope that it will be useful, but WITHOUT ANY
// WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
// PARTICULAR PURPOSE. See the GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License along with this
// program. If not, see <http://www.gnu.org/licenses/>.
//
// For any comment, suggestion or feature request, please contact the manager of
// the project at ceciliost@users.sourceforge.net
//-----

#ifndef __GNUG__ && !defined(NO_GCC_PRAGMA)
#pragma implementation "Processor.h"
#endif

// for (compilers that support precompilation, includes <wx/wx.h>.
#include <wx/wxprec.h>

#ifndef __BORLANDC__
#pragma hdrstop
#endif

#ifndef WX_PRECOMP
#include <wx/wx.h>
#else
#include <wx/sizer.h>
#include <wx/panel.h>
#endif

#include "Processor.h"

// access to main frame
#include "../app/MainFrame.h"
extern lmMainFrame* GetMainFrame();

#include "toolbox/ToolsBox.h"
#include "ScoreDoc.h"
#include "../exercises/auxctrols/UrlAuxCtl.h"
```

Analizador tonal en software libre

```
#include "../score/Score.h"
#include "../score/VStaff.h"
#include "../score/Instrument.h"
#include "../score/AuxObj.h"           //lmScoreLine
#include "../graphic/ShapeNote.h"
#include "ScoreCommand.h"

//access to error's logger
#include "../app/Logger.h"
extern lmLogger* g_pLogger;
#include "../auxmusic/Chord.h"
#include "../auxmusic/HarmonyExercisesData.h"

//-----
// Implementation of class lmHarmonyProcessor
//-----

IMPLEMENT_DYNAMIC_CLASS(lmHarmonyProcessor, lmScoreProcessor)

lmHarmonyProcessor::lmHarmonyProcessor()
 : lmScoreProcessor()
{
    pBoxSize = new wxSize(400, 60);
    pErrorBoxSize = new wxSize(580, 80);
    pBigErrorBoxSize = new wxSize(580, 120);

    tFont.sFontName = _T("Comic Sans MS");
    tFont.nFontSize = 6;
    tFont.nFontStyle = wxFONTSTYLE_NORMAL;
    tFont.nFontWeight = wxFONTWEIGHT_NORMAL;

    // Text box for general information
    // -200 --> Box X position shifted to left
    // 400 --> Initial Box Y position at the bottom
    // 0 --> Line X end position: centered with chord
    // 100 --> Line Y end position: slightly shifted down
    pInfoBox = new ChordInfoBox(pBoxSize, &tFont, -200, 500, 0, 100, -50);

    // Text box for error information
    // -200 --> Box X position shifted to left
    // -200 --> Initial Box Y position at the TOP
    // 0 --> Line X end position: centered with chord
    // 100 --> Line Y end position: slightly shifted down
    // +50 --> Increment y position after each use --> go downwards
    pChordErrorBox = new ChordInfoBox(pErrorBoxSize, &tFont, -150, -200, 0, 100, +50);
    pBigChordErrorBox = new ChordInfoBox(pBigErrorBoxSize, &tFont, -150, -200, 0, 100, +50);
}

lmHarmonyProcessor::~lmHarmonyProcessor()
{
    delete pBoxSize;
    delete pErrorBoxSize;
    delete pBigErrorBoxSize;
    delete pInfoBox;
    delete pChordErrorBox;
    delete pBigChordErrorBox;
}

bool lmHarmonyProcessor::ProcessScore(lmScore* pScore, void* pOptions)
{
    wxLogMessage(_T("ProcessScore") );

    //This method checks the score and show errors
    //Returns true if it has done any change in the score

    //As an example, I will put red and a green lines pointing to fourth and
    //sixth notes, respectively, and add some texts
    bool fScoreModified = false;

    // total number of chords (both valid and invalid chords)
    int nNumChords = 0;
    // all the information of each chord
    lmScoreChord* tChordDescriptor[lmMAX_NUM_CHORDS];
    // aux variable to know the active notes in each time step
    lmActiveNotes tActiveNotesList;
```

```

lmNote* pPossibleChordNotes[lmNOTES_IN_CHORD];

float rAbsTime = 0.0f;
float rTimeAtStartOfMeasure = 0.0f;

//Get the instrument
lmInstrument* pInstr = pScore->GetFirstInstrument();
lmVStaff* pVStaff = pInstr->GetVStaff();

lmNote* pCurrentNote;
float rCurrentNoteAbsTime = -2.0f;
float rRelativeTime = -2.0f;
wxString sStatusStr;

// TODO: improve positioning of the textbox...
pInfoBox->ResetPosition();

int nBadChords = 0;

wxColour colourForGenericErrors = wxColour(255,10,0,128); // R, G, B, Transparency: RED
wxColour colourForSpecialErrors = wxColour(255,0,0,128);

/*---

Algorithm of chord detection, keeping a list of "active notes"

When NEW NOTE
    if NEW TIME (i.e. higher) then
        update current list of active notes: remove notes with end-time < TIME
        analize possible chord in previous state: with current list of active notes
        set new time
    add new note to the list of active notes

---*/
// loop to process notes/rests in first staff of first instrument
int nNote = 0;
lmSOIterator* pIter = pVStaff->CreateIterator();
lmEKeySignatures nKey;

while(!pIter->EndOfCollection())
{
    // process only notes, rests and barlines
    lmStaffObj* pSO = pIter->GetCurrent();
    if (pSO->IsBarline())
    {
        // new measure starts. Update current time
        rTimeAtStartOfMeasure += pSO->GetTimePos();
        rAbsTime = rTimeAtStartOfMeasure;
    }
    else if (pSO->IsNoteRest())
    {
        // we continue in previous measure. Update current time if necessary
        rRelativeTime = pSO->GetTimePos();
        rAbsTime = rTimeAtStartOfMeasure + rRelativeTime;

        // process notes
        if (pSO->IsNote())
        {
            // It is a note. Count it
            ++nNote;
            pCurrentNote = (lmNote*) pSO;

            // calculate note's absolute time
            rCurrentNoteAbsTime = rTimeAtStartOfMeasure + rRelativeTime;

            // check new starting time (to analyze previous chord candidate)
            if ( IsHigherTime(rCurrentNoteAbsTime, tActiveNotesList.GetTime()) )
            {
                /*-----
                    if NEW-TIME (i.e. higher) then
                        update current list of active notes: remove notes with end-time < CURRENT-
TIME
                        analize possible chord in previous state: with current list of active
notes
                -----*/
            }
        }
    }
}

```

Analizador tonal en software libre

```
        set new time
        add new note to the list of active notes
----*/
// Get the key
if (pCurrentNote->GetApplicableKeySignature())
    nKey = pCurrentNote->GetApplicableKeySignature()->GetKeyType();
else
    nKey = earmDo;

// Get the notes
int nNumActiveNotes = tActiveNotesList.GetNotes(&pPossibleChordNotes[0]);
if (nNumActiveNotes > 0)
{
    // sort the notes
    SortChordNotes(nNumActiveNotes, &pPossibleChordNotes[0]);

    //
    // Get the chord from the notes
    //
    tChordDescriptor[nNumChords] = new lmScoreChord
        (nNumActiveNotes, &pPossibleChordNotes[0], nKey);
    wxLogMessage(_T(" ProcessScore: Chord %d : %s")
        , nNumChords, tChordDescriptor[nNumChords]->ToString().c_str());

    if (!tChordDescriptor[nNumChords]->IsStandardChord())
    {
        sStatusStr = wxString::Format(_T(" Bad chord %d: %s; Notes: %s")
            , nNumChords+1
            , tChordDescriptor[nNumChords]->lmChord::ToString().c_str()
            , tActiveNotesList.ToString().c_str());
        pInfoBox->DisplayChordInfo(pScore, tChordDescriptor[nNumChords]
            , colourForSpecialErrors, sStatusStr);
        nBadChords++;
    }
    nNumChords++;
    fScoreModified = true; // repaint
}

// set new time and recalculate list of active notes
tActiveNotesList.SetTime( rCurrentNoteAbsTime );
}

// add new note to the list of active notes
tActiveNotesList.AddNote(pCurrentNote, rCurrentNoteAbsTime + pCurrentNote-
>GetDuration());
} // if (pSO->IsNote())

} // [else if (!pSO->IsNoteRest()) ] ignore other staff objects

pIter->MoveNext();

} // while
delete pIter;           //Do not forget this. We are not using smart pointers!

// Analyze the remaining notes
//
tActiveNotesList.RecalculateActiveNotes( );

// TODO: not sure if this is the correct way of getting the key here...
// Get the key
if (pCurrentNote->GetApplicableKeySignature())
    nKey = pCurrentNote->GetApplicableKeySignature()->GetKeyType();
else
    nKey = earmDo;

// Get the notes
int nNumActiveNotes = tActiveNotesList.GetNotes(&pPossibleChordNotes[0]);
if (nNumActiveNotes > 0 )
{
    // Sort the notes
    SortChordNotes(nNumActiveNotes, &pPossibleChordNotes[0]);

    //
    // Get the chord from the notes
    //
```

```

// 
tChordDescriptor[nNumChords] = new lmScoreChord
    (nNumActiveNotes, &pPossibleChordNotes[0], nKey);
wxLogMessage(_T(" ProcessScore: END Chord %d : %s")
    , nNumChords, tChordDescriptor[nNumChords]->ToString().c_str());

if (!tChordDescriptor[nNumChords]->IsStandardChord())
{
    sStatusStr = wxString::Format(_T(" Bad chord %d: %s; Notes: %s")
        , nNumChords+1
        , tChordDescriptor[nNumChords]->lmChord::ToString().c_str()
        , tActiveNotesList.ToString().c_str());
    pInfoBox->DisplayChordInfo(pScore, tChordDescriptor[nNumChords]
        , colourForSpecialErrors, sStatusStr);
    nBadChords++;
}
nNumChords++;
}

wxLogMessage(_T("ProcessScore:ANALYSIS of %d chords: ", nNumChords);

int nNumHarmonyErrors = AnalyzeHarmonicProgression(&tChordDescriptor[0], nNumChords,
pChordErrorBox);

wxLogMessage(_T("ANALYSIS RESULT of %d chords: bad: %d, Num Errors:%d ")
    , nNumChords, nBadChords, nNumHarmonyErrors);

if (nBadChords == 0 && nNumHarmonyErrors == 0)
{
    wxString sOkMsg = _T(" Harmony is OK.");
    wxLogMessage( sOkMsg );
    if (nHarmonyExerciseChordsToCheck && nHarmonyExcerciseType == 3)
    {
        // In exercise 3 we just check the figured bass of automatically generated chords
        // therefore there is no need to say that the chords are correct
    }
    else
    {
        pInfoBox->DisplayChordInfo(pScore, tChordDescriptor[nNumChords-1], *wxGREEN, sOkMsg );
    }
    fScoreModified = true; // repaint
}

// todo: where to locate the box? restart: pChordErrorBox->SetYPosition(-200);

///////////////////////////////
// 
// Exercise specific checks
// 
///////////////////////////////

int nExerciseErrors = nNumHarmonyErrors;

if (nHarmonyExerciseChordsToCheck)
{
    wxLogMessage(_T(" *** HARMONY EXERCISE CHECK. Type: %d, Chords:%d ***")
        , nHarmonyExcerciseType, nHarmonyExerciseChordsToCheck);

    // Check: total number of chords
    if ( nHarmonyExerciseChordsToCheck != nNumChords)
    {
        wxString sMsg = wxString::Format(
            _T("Missing chords; Expected:%d, Actual: %d")
            , nNumChords, nHarmonyExerciseChordsToCheck);
        pChordErrorBox->DisplayChordInfo(pScore, tChordDescriptor[nNumChords-1]
            , colourForGenericErrors, sMsg);
        wxLogMessage(_T(" Error: %s"), sMsg.c_str() );
    }

    // For exercise 3: read the FiguredBass introduced by the user
    //
    lmSOIterator* pIter = pVStaff->CreateIterator();
    while(!pIter->EndOfCollection())
    {
        // process only notes, rests and barlines
    }
}

```

```

lmStaffObj* pSO = pIter->GetCurrent();
if (pSO->IsFiguredBass())
{
    pHE_UserFiguredBass[gnHE_NumUserFiguredBass++] = (lmFiguredBass*) pSO;
}
pIter->MoveNext();
}
delete pIter;           //Do not forget this. We are not using smart pointers!

lmEChordType nChordType;
int nInversions;
// Check
for (int nChordCount=0;
     nChordCount<nNumChords && nChordCount<nmAX_HARMONY_EXERCISE_CHORDS && nChordCount <
nHarmonyExerciseChordsToCheck;
     nChordCount++)
{
    int nChordExerciseErrors = 0;

    nChordType = tChordDescriptor[nChordCount]->GetChordType();
    nInversions = tChordDescriptor[nChordCount]->GetInversion();
    wxLogMessage(_T("Chord %d [%s] Type:%d, %d inversions")
                 , nChordCount+1, tChordDescriptor[nChordCount]->ToString().c_str()
                 , nChordType, nInversions, pHE_Chords[nChordCount]->GetChordType());

    // if chord is not valid, no need to say anything: the error message was already shown
    if ( nChordType != lmINVALID_CHORD_TYPE )
    {
        // Check the number of inversions
        // todo: consider to allow inversions as an option
        if ( (!bInversionsAllowedInHarmonyExercises) && nInversions > 0 )
        {
            nChordExerciseErrors++;
            wxString sMsg = wxString::Format(
                _T("Chord %d [%s] is not at root position: %d inversions")
                , nChordCount+1
                , tChordDescriptor[nChordCount]->lmFPitchChord::ToString().c_str()
                , nInversions);
            pChordErrorBox->DisplayChordInfo(pScore, tChordDescriptor[nChordCount]
                                              , colourForGenericErrors, sMsg);
        }
    }

    // Debug only: display chord notes
    wxLogMessage(_T(" Chord %d has %d notes ")
                 , nChordCount+1
                 , tChordDescriptor[nChordCount]->GetNumNotes() );
    for (int nxc=0; nxc<tChordDescriptor[nChordCount]->GetNumNotes(); nxc++)
    {
        wxLogMessage(_T(" NOTE %d: %s")
                     , nxc, FPitch_ToAbsLDPName(tChordDescriptor[nChordCount]-
>GetNoteFpitch(nxc)).c_str() );
    }

    if ( nHarmonyExcerciseType == 1 )
    {
        // Check absolute value of bass note (it was a prerequisite of the exercise)
        lmFPitch nExpectedBassNotePitch = pHE_Chords[nChordCount]->GetNoteFpitch(0);
        lmFPitch nActualBassNotePitch = tChordDescriptor[nChordCount]-
>GetNoteFpitch(0);
        wxString sMsg = wxString::Format(_T("Chord %d [%s]: bass note:%s, expected:%s
"))
                     , nChordCount+1
                     , tChordDescriptor[nChordCount]->lmFPitchChord::ToString().c_str()
                     , NormalizedFPitch_ToAbsLDPName(nActualBassNotePitch).c_str()
                     , NormalizedFPitch_ToAbsLDPName(nExpectedBassNotePitch).c_str()
                     );
        wxLogMessage( sMsg );
        if ( nExpectedBassNotePitch != nActualBassNotePitch )
        {
            nChordExerciseErrors++;
            pChordErrorBox->DisplayChordInfo(pScore, tChordDescriptor[nChordCount]
                                              , colourForGenericErrors, sMsg);
            wxLogMessage(_T(" Error: %s"), sMsg.c_str() );
        }
    }
}

```

```

// In any exercise: check the chord degree
// remember: chord degree <==> root note
//           root note <==> bass note ONLY IF NO INVERSIONS
int nExpectedRootStep = FPitch_Step(pHE_Chords[nChordCount]->GetNormalizedRoot());
int nActualRootStep = FPitch_Step(tChordDescriptor[nChordCount]-
>GetNormalizedRoot());
wxString sMsg = wxString::Format(
    _T("Chord %d [%s]: Degree:%s(root:%s), expected: %s(root:%s)")
    , nChordCount+1
    , tChordDescriptor[nChordCount]->lmFPitchChord::ToString().c_str()
    , GetChordDegreeString(nActualRootStep).c_str()
    , NormalizedFPitch_ToAbsLDPName(tChordDescriptor[nChordCount]-
>GetNormalizedRoot()).c_str()
    , GetChordDegreeString(nExpectedRootStep).c_str()
    , NormalizedFPitch_ToAbsLDPName(pHE_Chords[nChordCount]-
>GetNormalizedRoot()).c_str()
);
wxLogMessage( sMsg );
if ( nActualRootStep != nExpectedRootStep )
{
    nChordExerciseErrors++;
    pChordErrorBox->DisplayChordInfo(pScore, tChordDescriptor[nChordCount]
        , colourForGenericErrors, sMsg);
    wxLogMessage(_T(" Error: %s"), sMsg.c_str() );
}

// Exercise 2: check the soprano note
if ( nHarmonyExcerciseType == 2 )
{
    if (tChordDescriptor[nChordCount]->GetNumNotes() <= 3)
    {
        wxString sMsg = wxString::Format( _T("Chord %d [%s], Soprano voice is
missing")
            , nChordCount+1, tChordDescriptor[nChordCount]-
>lmFPitchChord::ToString().c_str());
    }
    else
    {
        // CHECK SOPRANO NOTE ( index: 3 )
        lmFPitch nExpectedSopranoNotePitch = pHE_Chords[nChordCount]-
>GetNoteFpitch(3);
        lmFPitch nActualSopranoNotePitch = tChordDescriptor[nChordCount]-
>GetNoteFpitch(3);
        wxString sMsg = wxString::Format( _T("Chord %d [%s]: soprano note:%s,
expected:%s ")
            , nChordCount+1
            , tChordDescriptor[nChordCount]->lmFPitchChord::ToString().c_str()
            , FPitch_ToAbsLDPName(nActualSopranoNotePitch).c_str()
            , FPitch_ToAbsLDPName(nExpectedSopranoNotePitch).c_str()
        );
        wxLogMessage( sMsg );
        // The soprano note must match exactly the original generated by lenmus
        if (nActualSopranoNotePitch != nExpectedSopranoNotePitch )
        {
            nChordExerciseErrors++;
            pChordErrorBox->DisplayChordInfo(pScore, tChordDescriptor[nChordCount]
                , colourForGenericErrors, sMsg);
            wxLogMessage(_T(" Error: %s"), sMsg.c_str() );
        }
    }
}
// Check: the chord
// todo: this check might be removed
//       idea: If the chords are not "equivalent" then 'something is wrong'
//              but it would be better to say exactly what is wrong
//       info: IsEqualTo checks the intervals and the bass
//
// I am not sure if we the calculated chord in exercises 1 and 2 is the only
solution
//
// Also consider to leave this check: it shows the correct answer (the correct
chord)
//
// IDEA: do this check only if no other exercise errors in this chord
if ( nChordExerciseErrors == 0 ) // NO EXERCISE ERRORS

```

```

    {
        if ( ! pHE_Chords[nChordCount]->IsEqualTo( tChordDescriptor[nChordCount] ) )
        {
            nChordExerciseErrors++;
            wxString sMsg = wxString::Format(
                _T("Chord %d [%s] is not the expected: %s")
                , nChordCount+1
                , tChordDescriptor[nChordCount]->lmChord::ToString().c_str()
                , pHE_Chords[nChordCount]->lmChord::ToString().c_str()
            );
            pBigChordErrorBox->DisplayChordInfo(pScore, tChordDescriptor[nChordCount]
                , colourForGenericErrors, sMsg);
        }
    }

    nExerciseErrors += nChordExerciseErrors;

} // if chord is valid
} // for chords

// EXERCISE 3
if ( nHarmonyExcerciseType == 3 )
{
    wxString sMsg = wxString::Format(
        _T("Exercise 3: Number of figured bass introduced by user:%d, expected:%d")
        , gnHE_NumUserFiguredBass, nNumChords);
    wxLogMessage(sMsg.c_str());

    // Check chord count
    if ( gnHE_NumUserFiguredBass != nNumChords )
    {
        nExerciseErrors++;
        pChordErrorBox->DisplayChordInfo(pScore, tChordDescriptor[nNumChords-1]
            , colourForGenericErrors, sMsg);
        wxLogMessage( _T("DISPLAYED ERROR :%s"), sMsg.c_str());
    }

    // check each FiguredBass
    for (int nFB=0; nFB<gnHE_NumUserFiguredBass && nFB<nNumChords; nFB++)
    {
        nChordType = tChordDescriptor[nFB]->GetChordType();

        // if chord is not valid, no need to say anything: the error message was already
shown
        if ( nChordType != lmINVALID_CHORD_TYPE )
        {
            sMsg = wxString::Format(
                _T("Chord %d [%s] FiguredBass: user: %s, expected:%s")
                , nFB+1
                , tChordDescriptor[nFB]->lmFPitchChord::ToString().c_str()
                , pHE_UserFiguredBass[nFB]->GetFiguredBassString().c_str()
                , pHE_FiguredBass[nFB]->GetFiguredBassString().c_str()
            );
            if ( pHE_UserFiguredBass[nFB]->GetFiguredBassString()
                != pHE_FiguredBass[nFB]->GetFiguredBassString()
            )
            {
                nExerciseErrors++;
                pChordErrorBox->DisplayChordInfo(pScore, tChordDescriptor[nFB],
colourForGenericErrors, sMsg);
            }
        }
    }

    wxLogMessage(_T(" SUMMARY"));
    wxLogMessage(_T(" Created %d chords:"), nNumChords );
    for (int i = 0; i <nNumChords; i++)
        wxLogMessage(_T(" Chord %d: %s"), i, pHE_Chords[i]->ToString().c_str() );
    wxLogMessage(_T(" Checked %d chords:"), nHarmonyExerciseChordsToCheck );
    for (int i = 0; i <nHarmonyExerciseChordsToCheck; i++)
        wxLogMessage(_T(" Chord %d: %s"), i, tChordDescriptor[i]->ToString().c_str() );
}

```

```

wxLogMessage(_T(" Bad chords: %d, harmony errors:%d"), nBadChords, nNumHarmonyErrors);
wxLogMessage(_T(" Exercise errors: %d"), nExerciseErrors);

if (nExerciseErrors == 0)
{
    wxString sOkMsg = _T(" Exercise is OK.");
    pInfoBox->DisplayChordInfo(pScore, tChordDescriptor[nNumChords-1], *wxGREEN, sOkMsg );
}

for (int i = 0; i <nHarmonyExerciseChordsToCheck; i++)
{
    if (pHE_Chords[i] != NULL)
    {
        delete pHE_Chords[i];
        pHE_Chords[i] = 0;
    }
    if (pHE_FiguredBass[i] != NULL)
    {
        delete pHE_FiguredBass[i];
        pHE_FiguredBass[i] = 0;
    }
}

nHarmonyExerciseChordsToCheck = 0;

}

for (int i = 0; i <nNumChords; i++)
{
    delete tChordDescriptor[i];
    tChordDescriptor[i] = 0;
}

return fScoreModified; //true -> score modified
}

```

8.1.2 Harmony.cpp

RESUMEN: Funciones y clases auxiliares.

URL:

<https://lenmus.svn.sourceforge.net/svnroot/lenmus/trunk/src/auxmusic/Harmony.cpp>

```

-----
// LenMus Phonascus: The teacher of music
// Copyright (c) 2002-2010 LenMus project
//
// This program is free software; you can redistribute it and/or modify it under the
// terms of the GNU General Public License as published by the Free Software Foundation,
// either version 3 of the License, or (at your option) any later version.
//
// This program is distributed in the hope that it will be useful, but WITHOUT ANY
// WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
// PARTICULAR PURPOSE. See the GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License along with this
// program. If not, see <http://www.gnu.org/licenses/>.
//
// For any comment, suggestion or feature request, please contact the manager of
// the project at ceciliost@users.sourceforge.net
//
-----

#ifndef __GNUG__
#define __GNUG__ 1
#endif

#ifndef NO_GCC_PRAGMA
#define __attribute__(x) __attribute__((x))
#endif

#pragma implementation "Harmony.h"
#endif

```

Analizador tonal en software libre

```
// for (compilers that support precompilation, includes <wx/wx.h>.
#include <wx/wxprec.h>

#ifndef __BORLANDC__
#pragma hdrstop
#endif

#include "Harmony.h"
#include "../score/VStaff.h"

#include "../app/MainFrame.h"
extern lmMainFrame* GetMainFrame();
#include "../app/ScoreDoc.h" // DisplayChordInfo(GetMainFrame() ->GetActiveDoc() ->GetScore()

// Global functions

// return: total number of errors
int AnalyzeHarmonicProgression(lmScoreChord** pChordDescriptor, int nNCH, ChordInfoBox*
pChordErrorBox)
{
    wxLogMessage(_T("**** AnalyzeHarmonicProgression N:%d "), nNCH);
    for (int i = 0; i<nNCH; i++)
    {
        wxLogMessage(_T("    Chord %d to analyze: %s"), i, pChordDescriptor[i]->ToString().c_str());
    }

    int nNumChordError[lmMAX_NUM_CHORDS]; // number of errors in each chord

    lmRuleList tRules(pChordDescriptor, nNCH);

    wxString sStr;
    sStr.clear();

    int nNumErrors = 0; // TODO: decide: total num of errors or num of chords with error
    lmRule* pRule;
    // TODO: create a method of the list to evaluate ALL the rules ?
    for (int nR = lmCVR_FirstChordValidationRule; nR<lmCVR_LastChordValidationRule; nR++)
    {
        pRule = tRules.GetRule(nR);
        if ( pRule == NULL)
        {
//todo: remove this message?           wxLogMessage(_T(" Rule %d is NULL !!!!"), nR);
        }
        else if (pRule->IsEnabled())
        {
            wxLogMessage(_T("Evaluating rule %d, description: %s")
                , pRule->GetRuleId(), pRule->GetDescription().c_str());
            nNumErrors += pRule->Evaluate(sStr, &nNumChordError[0], pChordErrorBox);
            wxLogMessage(_T("    Total error count after rule %d: %d errors"), pRule->GetRuleId(),
nNumErrors );
            if (nNumErrors > 0)
            {
                // TODO: anything else here?
            }
        }
    }

    return nNumErrors;
}

// Harmonic direction of an interval:
// descending (-1) when the interval is negative
// ascending (1) when positive
// linear when 0
int GetHarmonicDirection(int nInterval)
{
    if (nInterval > 0)
        return 1;
    else if (nInterval < 0)
        return -1;
    else
        return 0;
}
```

```

// Harmonic motion of 2 voices: calculated from the harmonic direction of each voice
// parallel: both voices have the same direction
// contrary: both voices have opposite direction
// oblique: one direction is linear and the other is not
int GetHarmonicMovementType( lmFPitch fVoice10, lmFPitch fVoice11, lmFPitch fVoice20, lmFPitch
fVoice21)
{
    int nMovType = -10;

    int nD1 = GetHarmonicDirection(fVoice11 - fVoice10);
    int nD2 = GetHarmonicDirection(fVoice21 - fVoice20);

    if (nD1 == nD2)
    {
        nMovType = lm_eDirectMovement;
    }
    else if (nD1 == -nD2)
    {
        nMovType = lm_eContraryMovement;
    }
    else
    {
        assert ( (nD1 == 0 && nD2 != 0) || (nD2 == 0 && nD1 != 0) );
        nMovType = lm_eObliqueMovement;
    }
    return nMovType;
}

int GetIntervalNumberFromFPitchDistance(lmFPitch n2, lmFPitch n1) //@@@ todo remove!!!
{
    lmFIntval nDistance = abs (n2 - n1);
    int nIntervalNumber = FIntval_GetNumber(nDistance);
    wxLogMessage(_T("\t\t GetIntervalNumberFromFPitchDistance: %d-%d D:%d I:%d ")
        , n2, n1, nDistance, nIntervalNumber);
    return nIntervalNumber;
}

// todo: move this to "Pitch" file o merge this with FPitch_ToAbsLDPName
// This is just FPitch_ToAbsLDPName but WITHOUT OCTAVE
static wxString m_sNoteName[7] =
    {_T("c"), _T("d"), _T("e"), _T("f"), _T("g"), _T("a"), _T("b") };
wxString NormalizedFPitch_ToAbsLDPName(lmFPitch fp)
{
    wxString sAnswer;
    switch(FPitch_Accidentals(fp)) {
        case -2: sAnswer = T("--"); break;
        case -1: sAnswer = _T("-"); break;
        case 0: sAnswer = _T(""); break;
        case 1: sAnswer = _T("+"); break;
        case 2: sAnswer = _T("x"); break;
        default:
            return wxEmptyString;
    }
    sAnswer += m_sNoteName[FPitch_Step(fp)];
    return sAnswer;
}

static wxString m_sNumeralsDegrees[7] =
    {_T(" I"), _T(" II"), _T("III"), _T(" IV"), _T(" V"), _T(" VI"), _T("VII")};

wxString GetChordDegreeString(lmStepType nStep )
{
    if (nStep < lmMIN_STEP || nStep > lmMAX_STEP)
    {
        wxLogMessage(_T("GetDegreeString: Invalid step %d"), nStep);
        nStep = 0;
    }

    return m_sNumeralsDegrees[nStep];
}

```

Analizador tonal en software libre

```
// Get interval in FPitch from:  
//   chord degree (root note step)  
//   key signature  
//   interval index (1=3rd, 2=5th, etc)  
// TODO: Used in harmony, but if it useful in general, move it to a better place such as Pitch  
file  
lmFIntval FPitchInterval(int nRootStep, lmEKeySignatures nKey, int nInterval)  
{  
    // steps: 0 .. 6 (lmSTEP_C .. lmSTEP_B)  
    assert (nRootStep>=lmSTEP_C && nRootStep <= lmSTEP_B);  
  
    // aware: in harmony by default an interval has 2 steps, therefore step2 = step1 + 2*N  
    lmFPitch fpPitch = FPitchStepsInterval(nRootStep, (nRootStep+(nInterval*2))%(lmSTEP_B+1),  
nKey);  
    return (lmFIntval) fpPitch;  
}  
  
//-----  
  
void SortChordNotes(int nNumNotes, lmNote** pInpChordNotes)  
{  
    wxASSERT(nNumNotes < lmNOTES_IN_CHORD);  
    // Classic Bubble sort  
    int nCount, fSwapDone;  
    lmNote* auxNote;  
    do  
    {  
        fSwapDone = 0;  
        for (nCount = 0; nCount < nNumNotes - 1; nCount++)  
        {  
            wxASSERT(pInpChordNotes[nCount] != NULL);  
            wxASSERT(pInpChordNotes[nCount+1] != NULL);  
            if (pInpChordNotes[nCount]->GetFPitch() > pInpChordNotes[nCount+1]->GetFPitch() )  
            {  
                auxNote = pInpChordNotes[nCount];  
                pInpChordNotes[nCount] = pInpChordNotes[nCount + 1];  
                pInpChordNotes[nCount + 1] = auxNote;  
                fSwapDone = 1;  
            }  
        }  
    }while (fSwapDone);  
}  
  
void SortChordNotes(int nNumNotes, lmFPitch fInpChordNotes[])  
{  
    wxASSERT(nNumNotes < lmNOTES_IN_CHORD);  
    wxASSERT(fInpChordNotes != NULL);  
    // Classic Bubble sort  
    int nCount, fSwapDone;  
    lmFPitch auxNote;  
    do  
    {  
        fSwapDone = 0;  
        for (nCount = 0; nCount < nNumNotes - 1; nCount++)  
        {  
            if (fInpChordNotes[nCount] > fInpChordNotes[nCount+1] )  
            {  
                auxNote = fInpChordNotes[nCount];  
                fInpChordNotes[nCount] = fInpChordNotes[nCount + 1];  
                fInpChordNotes[nCount + 1] = auxNote;  
                fSwapDone = 1;  
            }  
        }  
    }while (fSwapDone);  
}  
  
#if 0  
/*  
This method is not used anymore. lmChordInfo has evolved to lmChord and lmChordIntervals, which  
use a different approach to extract the intervals.  
But the process of obtaining the intervals from the notes is essential in chord processing,  
therefore, we leave this method as an example of to do it.  
*/  
void GetIntervalsFromNotes(int nNumNotes, lmNote** pInpChordNotes, lmChordInfo* tOutChordInfo)  
{
```

```

wxASSERT(nNumNotes < lmNOTES_IN_CHORD);
wxASSERT(pInpChordNotes != NULL);
wxASSERT(tOutChordInfo != NULL);
wxASSERT(pInpChordNotes[0] != NULL);

lmFIntval fpIntv;
int nCurrentIntvIndex = 0;
int nExistingIntvIndex = 0;
for (int nCount = 1; nCount < nNumNotes; nCount++)
{
    wxASSERT(pInpChordNotes[nCount] != NULL);
    fpIntv = (lmFIntval) (pInpChordNotes[nCount]->GetFPitch() - pInpChordNotes[0]->GetFPitch());

    if (fpIntv >= lm_p8)
    {
        fpIntv = fpIntv % lm_p8;
    }
#endif _LM_DEBUG_
    wxLogMessage(_T("[GetIntervalsFromNotes note %d: %d note 0: %d] INTERVAL: %d")
                 , nCount, pInpChordNotes[nCount]->GetFPitch(), pInpChordNotes[0]->GetFPitch(), fpIntv);
#endif
    // Update chord interval information
    for (nExistingIntvIndex=0; nExistingIntvIndex<nCurrentIntvIndex; nExistingIntvIndex++)
    {
        if (tOutChordInfo->nIntervals[nExistingIntvIndex] == fpIntv)
            break; // interval already exists
    }
    if (nExistingIntvIndex < nCurrentIntvIndex)
    {
        wxLogMessage(_T(" Interval %d: IGNORED, already in %d")
                     , fpIntv, nExistingIntvIndex);
    }
    else
    {
        if (fpIntv == 0)
        {
            // Ignored 0 Interval
        }
        else
        {
            // Add interval
            tOutChordInfo->nIntervals[nCurrentIntvIndex] = fpIntv;
            nCurrentIntvIndex++;
        }
    }
}

tOutChordInfo->nNumNotes = nNumNotes;
tOutChordInfo->nNumIntervals = nCurrentIntvIndex;

// Sort Intervals
// Classic bubble sort
int fSwapDone;
lmFIntval auxIntv;
do
{
    fSwapDone = 0;
    for (int nCount = 0; nCount < tOutChordInfo->nNumIntervals - 1; nCount++)
    {
        if (tOutChordInfo->nIntervals[nCount] > tOutChordInfo->nIntervals[nCount+1] )
        {
            auxIntv = tOutChordInfo->nIntervals[nCount];
            tOutChordInfo->nIntervals[nCount] = tOutChordInfo->nIntervals[nCount + 1];
            tOutChordInfo->nIntervals[nCount + 1] = auxIntv;
            fSwapDone = 1;
        }
    }
}while (fSwapDone);

// Set the non-used intervals to NULL
for (int i=tOutChordInfo->nNumIntervals; i<lmINTERVALS_IN_CHORD; i++)
{
    tOutChordInfo->nIntervals[i] = lmNULL_FIntval;
}

```

```

}

#endif

// 
// Message box to display the results if the chord analysis
//
// Remember:
//     x: relative to object; positive: right
//     y: relative to top line; positive: down
ChordInfoBox::ChordInfoBox(wxSize* pSize, lmFontInfo* pFontInfo
                           , int nBoxX, int nBoxY, int nLineX, int nLineY, int nBoxYIncrement)
{
    Settings(pSize, pFontInfo, nBoxX, nBoxY, nLineX, nLineY, nBoxYIncrement);
}
void ChordInfoBox::Settings(wxSize* pSize, lmFontInfo* pFontInfo
                           , int nBoxX, int nBoxY, int nLineX, int nLineY, int nBoxYIncrement)
{
    m_ntConstBoxXstart = nBoxX;
    m_ntConstInitialBoxYStart = nBoxY;
    m_ntConstLineXstart = nLineX;
    m_ntConstLineYStart = nLineY;
    m_ntConstBoxYIncrement = nBoxYIncrement;
    m_pFontInfo = pFontInfo;
    m_pSize = pSize;

    assert(m_pFontInfo != NULL);
    assert(m_pSize != NULL);

    m_ntCurrentBoxYStart = m_ntConstInitialBoxYStart;
}
void ChordInfoBox::ResetPosition()
{
    m_ntCurrentBoxYStart = m_ntConstInitialBoxYStart;
}
void ChordInfoBox::SetYPosition(int nYpos)
{
    m_ntCurrentBoxYStart = nYpos;
}
void ChordInfoBox::DisplayChordInfo(lmScore* pScore, lmScoreChord* pChordDsct, wxColour colour,
wxString &sText)
{
    if (pChordDsct == NULL )
    {
        wxLogMessage(
            _T(" DisplayChordInfo ERROR: Chord descriptor is NULL. Msg: %s")
            , sText.c_str());
        return; // todo: improvement: in this case, display a box but not attached to any note ?
    }
    if ( pChordDsct->GetNumNotes() < 1)
    {
        wxLogMessage(_T(" DisplayChordInfo ERROR: NO notes to attach the textbox"));
        return; // todo: improvement: in this case, display a box but not attached to any note ?
    }

    int m_nNumChordNotes = pChordDsct->GetNumNotes();
    if ( ! pChordDsct->HasLmNotes())
    {
        wxLogMessage(_T(" DisplayChordInfo ERROR: NO score notes!"));
        return;
    }
    lmTextStyle* pStyle = pScore->GetStyleName(*m_pFontInfo);

    // Display chord info in score with a line and text
    assert(m_nNumChordNotes > 0);
    assert(m_nNumChordNotes < 20);

    for (int i = 0; i<m_nNumChordNotes; i++)
    {
        assert(pChordDsct->GetNoteLmNote(i) != NULL);
        pChordDsct->GetNoteLmNote(i)->SetColour(colour);
    }

    // Line end: the first note
}

```

```

lmStaffObj* cpSO = pChordDsct->GetNoteLmNote(m_nNumChordNotes-1);
lmTPoint lmTBoxPos(m_ntConstBoxXstart, m_ntCurrentBoxYStart);
lmTPoint lmTLinePos(m_ntConstLineXstart, m_ntConstLineYStart);
lmAuxObj* pTxtBox = cpSO->AttachTextBox(lmTBoxPos, lmTLinePos, sText, pStyle, *m_pSize,
colour);

// here increment the static variables
m_ntCurrentBoxYStart += m_ntConstBoxYIncrement;
}

void DrawArrow(lmNote* pNote1, lmNote* pNote2, wxColour color)
{
    //get VStaff
    lmVStaff* pVStaff = pNote1->GetVStaff();

    //get note heads positions
    lmURect uBounds1 = pNote1->GetNoteheadShape()->GetBounds();
    lmURect uBounds2 = pNote2->GetNoteheadShape()->GetBounds();

    //start point
    lmUPoint uStart( uBounds1.GetWidth(), 0 );
    uStart.y = pNote1->GetShiftToNotehead();           //center of notehead

    //end point
    lmUPoint uEnd(uBounds2.GetRightTop() - uBounds1.GetRightTop());
    uEnd.y += uStart.y;

    //convert to tenths
    lmTenths xtStart = pVStaff->LogicalToTenths(uStart.x) + 8.0;
    lmTenths ytStart = pVStaff->LogicalToTenths(uStart.y);
    lmTenths xtEnd = pVStaff->LogicalToTenths(uEnd.x) - 8.0;
    lmTenths ytEnd = pVStaff->LogicalToTenths(uEnd.y);

    //create arrow
    pNote1->AttachLine(xtStart, ytStart, xtEnd, ytEnd, 2, lm_eLineCap_None,
                         lm_eLineCap_Arrowhead, lm_eLine_Solid, color);
    pNote1->SetColour(color);
    pNote2->SetColour(color);
}

/* AWARE:

lmChord is the basic chord, defined by ROOT NOTE and intervals
aware: only the root note is REAL; the rest of notes, obtained with GetNote(i) (where i>0)
are just POSSIBLE notes

for certain operations we need to know all the real notes of the chord

lmFPitchChord is a lmChord that can contain also notes in "lmFPitch".
The lmFPitch notes can be added after the construction, except the root note (bass voice) that is
already in lmChord
remember: the ROOT NOTE is REQUIRED in ANY chord;
it is defined in the construction and should never be modified afterwards

Possible constructors :
1 created with NO notes; then added with AddNoteFromLmNote or AddNoteLmFPitch
    GetNoteFpitch returns a 0 if the note has not been added
2 created with notes. NO POSSIBLE TO ADD NOTES. [controlled with m_fCreatedWithNotes]
check the number of notes available with GetNumNotes()
Notes can not be removed nor modified

lmScoreChord is a 'real score chord': it contains actual notes of the score (lmNotes)
it is a lmFPitchChord that can contain also notes in "lmNotes"

The lmNotes notes can be added after the construction

aware: the root note might NOT be present. The root note is alway present in lmFPitch, but it
may not have
    a corresponding lmNote.

Possible constructors :
1 created with NO notes; then
    added with AddNoteFromLmNote

```

Analizador tonal en software libre

```
    adds both lmFPitch and lmNotes
    added ONLY lmFPitch with AddNoteLmFPitch or AddNoteFromInterval
        it only adds lmFPitch !!!
            To add the missing lmNote, use SetLmNote
                since the lmFPitch of this note must exist
                    it must be the same as the note->GetFPitch(). This is checked.
            GetNoteFpitch returns a 0 if the note has not been added
        2 created with notes. NO POSSIBLE TO ADD NOTES.
*/
//
// class lmFPitchChord
//

// Constructors

// Create a chord from a list of ORDERED score notes in LmFPitch
lmFPitchChord::lmFPitchChord(int nNumNotes, lmFPitch fNotes[], lmEKeySignatures nKey)
: lmChord(nNumNotes, fNotes, nKey)
{
    assert(nNumNotes<lmNOTES_IN_CHORD);
    for (int i = 0; i<nNumNotes; i++)
    {
        assert(IsValidChordNote(fNotes[i]));
        if (i == 0)
        {
            assert( (fNotes[0] % lm_p8) == GetNormalizedBass() );
        }
        m_fpChordNotes[i] = fNotes[i];
    }
    for (int i = nNumNotes; i<lmNOTES_IN_CHORD; i++)
    {
        m_fpChordNotes[i] = 0;
    }
    m_nNumChordNotes = nNumNotes;
    m_fCreatedWithNotes = true;
}

lmFPitchChord::lmFPitchChord(int nNumNotes, lmNote** pNotes, lmEKeySignatures nKey)
: lmChord(nNumNotes, pNotes, nKey)
{
    assert(nNumNotes<lmNOTES_IN_CHORD);
    for (int i = 0; i<nNumNotes; i++)
    {
        assert(IsValidChordNote(pNotes[i]->GetFPitch()) );
        if (i == 0)
        {
            assert( (pNotes[0]->GetFPitch() % lm_p8) == GetNormalizedBass() );
            // actual bass note must be consistent with chord bass note
        }
        m_fpChordNotes[i] = pNotes[i]->GetFPitch();
    }
    for (int i = nNumNotes; i<lmNOTES_IN_CHORD; i++)
    {
        m_fpChordNotes[i] = 0;
    }
    m_nNumChordNotes = nNumNotes;
    m_fCreatedWithNotes = true;
}

// Creates a chord from "essential" information
lmFPitchChord::lmFPitchChord(int nDegree, lmEKeySignatures nKey, int nNumIntervals, int
nNumInversions, int octave)
: lmChord(nDegree, nKey, nNumIntervals, nNumInversions, octave)
{
    m_nNumChordNotes = 0;
    m_fCreatedWithNotes = false;
}

// todo: consider to implement IsEqualTo in lmFPitchChord as
//       lmChord::IsEqualTo + notes comparison
//       The question is:
//       Are two chords equal if both are of the same type but have different notes?

int lmFPitchChord::AddNoteLmFPitch(lmFPitch fNote)
{
```

```

if (m_fCreatedWithNotes)
{
    wxLogMessage(_T(" lmFPitchChord::AddNoteLmFPitch ERROR, it was created with %d notes ")
                 , m_nNumChordNotes );
}
else
{
    assert(m_nNumChordNotes<lmNOTES_IN_CHORD);
    if ( this->IsValidChordNote(fNote) )
    {
        m_fpChordNotes[m_nNumChordNotes] = fNote;
        m_nNumChordNotes++;
        SortChordNotes(m_nNumChordNotes, m_fpChordNotes); // sort notes so that voice <=> index
    }
    else
    {
        wxLogMessage(_T(" lmFPitchChord::AddNoteLmFPitch ERROR note %d [%s] does not belong to
chord %s")
                     , fNote, FPitch_ToAbsLDPName(fNote).c_str() , this->ToString().c_str() );
    }
}
return m_nNumChordNotes;
}

bool lmFPitchChord::IsBassDuplicated()
{
    // remember: the lowest note is the BASS note, not the root note
    //           (bass == root only if NO INVERSIONS)

    // Normalize wit "% lm_p8" to remove octave information
    for (int i=1; i<m_nNumChordNotes; i++)
    {
        if ( (m_fpChordNotes[i] % lm_p8) == this->GetNormalizedBass() )
            return true;
    }
    return false;
}

void lmFPitchChord::RemoveAllNotes()
{
    m_nNumChordNotes = 0;
    for (int i = 0; i<lmNOTES_IN_CHORD; i++)
    {
        m_fpChordNotes[i] = 0;
    }
}

wxString lmFPitchChord::ToString()
{
    // extend the parent information
    wxString sStr = this->lmChord::ToString();
    sStr += _T("; Notes:");
    for (int nN = 0; nN<m_nNumChordNotes; nN++)
    {
        sStr += _T(" ");
        sStr += FPitch_ToAbsLDPName(m_fpChordNotes[nN]).c_str();
    }
    return sStr;
}

// 
// class lmScoreChord
//

// Creates a chord from a list of ordered score notes
lmScoreChord::lmScoreChord(int nNumNotes, lmNote** pNotes, lmEKeySignatures nKey)
: lmFPitchChord(nNumNotes, pNotes, nKey)
{
    // Whenever a lmNote is added, it should be checked that
    // - this note matches the corresponding in lmFPitch (m_fpChordNotes)
}

```

Analizador tonal en software libre

```
// - the note is valid: it can be obtained from the bass note by adding an interval and +- octaves
m_nNumLmNotes = 0;
assert(nNumNotes<lmNOTES_IN_CHORD);
for (int i = 0; i<nNumNotes; i++)
{
    assert( pNotes[i]->GetFPitch() == m_fpChordNotes[i] );
    if (i == 0)
    {
        // ENSURE THE lmFPitch of the note is the same as pNote-> GetFPitch()
        assert( (pNotes[0]->GetFPitch() % lm_p8) == GetNormalizedBass() );
        // actual bass note must be consistent with chord bass note
    }
    m_pChordNotes[i] = pNotes[i];
    m_nNumLmNotes++;
}
for (int i = nNumNotes; i<lmNOTES_IN_CHORD; i++)
{
    m_pChordNotes[i] = 0;
}

tChordErrors.nErrList = 0;
}

lmScoreChord::lmScoreChord(int nDegree, lmEKeySignatures nKey, int nNumIntervals, int nNumInversions, int octave)
: lmFPitchChord(nDegree, nKey, nNumIntervals, nNumInversions, octave)
{
    m_nNumLmNotes = 0;
    for (int i = 0; i<lmNOTES_IN_CHORD; i++)
    {
        m_pChordNotes[i] = 0;
    }
    tChordErrors.nErrList = 0;
}

// aware: this is only to associate the lmNote to a note in lmFPitch that already exists
// it is not to add a note!
bool lmScoreChord::SetLmNote(lmNote* pNote)
{
    assert(pNote);

    for (int nIndex = 0; nIndex < lmNOTES_IN_CHORD; nIndex++)
    {
        if (m_fpChordNotes[nIndex] == pNote->GetFPitch())
        {
            m_pChordNotes[nIndex] = pNote;
            m_nNumLmNotes++;
            wxLogMessage(_T(" SetLmNote %d %d OK, total LmNotes:%d"), nIndex,
m_fpChordNotes[nIndex], m_nNumLmNotes);
            return true;
        }
    }
    wxLogMessage(_T(" SetLmNote ERROR!! %d (%s) , not found in %d notes")
        , pNote->GetFPitch(), FPitch_ToAbsLDPName(pNote->GetFPitch()).c_str(), m_nNumChordNotes);
    return false;
}

lmNote* lmScoreChord::GetNoteLmNote(int nIndex)
{
    if (m_pChordNotes[nIndex] != 0)
        return m_pChordNotes[nIndex];
    else
        return 0;
}
int lmScoreChord::GetNoteVoice(int nIndex)
{
    if (m_pChordNotes[nIndex] != 0)
        return m_pChordNotes[nIndex]->GetVoice();
    else
    {
        assert(nIndex<m_nNumChordNotes);
        return m_nNumChordNotes-nIndex;
    }
}
```

```

    }

int lmScoreChord::GetNumLmNotes() // todo: not necessary: remove
{
    return m_nNumLmNotes;
/* other possibility:
int nCount = 0;
for (int i = 0; i<lmNOTES_IN_CHORD; i++)
{
    if (m_pChordNotes[i] != 0)
        nCount++;

}
return nCount; -*/
}

void lmScoreChord::RemoveAllNotes()
{
    this->lmFPitchChord::RemoveAllNotes();
    m_nNumLmNotes = 0;
    for (int i = 0; i<lmNOTES_IN_CHORD; i++)
    {
        m_pChordNotes[i] = 0;
    }
}

wxString lmScoreChord::ToString()
{
    wxString sStr = this->lmFPitchChord::ToString();
/*- todo remove: the notes should be the same as in lmFPitchChord
sStr += wxString::Format(_T("; %d lmNotes:"), m_nNumLmNotes);
for (int nN = 0; nN<m_nNumLmNotes; nN++)
{
    if (m_pChordNotes[nN] != 0 && m_fpChordNotes[nN] != 0)
    {
        sStr += _T(" ");
        sStr += m_pChordNotes[nN]->GetPrintName().c_str();
    }
} */
    return sStr;
}

//-----
// class lmActiveNotes
//-----

lmActiveNotes::lmActiveNotes()
    : m_rCurrentTime(0.0f)
{
}

lmActiveNotes::~lmActiveNotes()
{
    std::list<lmActiveNoteInfo*>::iterator it;
    it=m_ActiveNotesInfo.begin();
    while( it != m_ActiveNotesInfo.end())
    {
        delete *it;
        it = m_ActiveNotesInfo.erase(it);
    }
}

void lmActiveNotes::SetTime(float rNewcurrentTime)
{
    m_rCurrentTime = rNewcurrentTime;
    RecalculateActiveNotes();
}

void lmActiveNotes::ResetNotes()
{
}

```

Analizador tonal en software libre

```
m_ActiveNotesInfo.clear();
}

int lmActiveNotes::GetNumActiveNotes()
{
    return (int)m_ActiveNotesInfo.size();
}

int lmActiveNotes::GetNotes(lmNote** pNotes)
{
    assert(pNotes != NULL);
    std::list<lmActiveNoteInfo*>::iterator it;
    int nCount = 0;
    for(it=m_ActiveNotesInfo.begin(); it != m_ActiveNotesInfo.end(); ++it, nCount++)
    {
        pNotes[nCount] = (*it)->pNote;
    }
    return nCount;
}

void lmActiveNotes::AddNote(lmNote* pNoteS, float rEndTimeS)
{
    lmActiveNoteInfo* plmActiveNoteInfo = new lmActiveNoteInfo(pNoteS, rEndTimeS);
    m_ActiveNotesInfo.push_back( plmActiveNoteInfo );
}

void lmActiveNotes::RecalculateActiveNotes()
{
    std::list<lmActiveNoteInfo*>::iterator it;
    it=m_ActiveNotesInfo.begin();
    while(it != m_ActiveNotesInfo.end())
    {
        // AWARE: EQUAL time considered as finished (TODO: CONFIRM by music expert)
        if ( ! IsHigherTime( (*it)->rEndTime, m_rCurrentTime ) )
        {
            delete *it;
            it = m_ActiveNotesInfo.erase(it); // aware: "it = " needed to avoid crash in loop....
        }
        else
            it++;
    }
}

// TODO: method used for debug. Keep it?
wxString lmActiveNotes::ToString()
{
    wxString sRetStr;
    std::list<lmActiveNoteInfo*>::iterator it;
    for(it=m_ActiveNotesInfo.begin(); it != m_ActiveNotesInfo.end(); ++it)
    {
        sRetStr += wxString::Format(_T(" %s "), (*it)->pNote->GetPrintName().c_str());
    }
    return sRetStr;
}

// 
// class lmRuleList
// 

lmRuleList::lmRuleList(lmScoreChord** pChD, int nNumChords)
{
    CreateRules();
    SetChordDescriptor(pChD, nNumChords);
};

// TODO: ADD MORE HARMONY RULES
//      To add a rule:
//      1) Create the class (recommended to use the macro LM_CREATE_CHORD_RULE)
//      2) Add an instance in AddRule
//      3) Implement the Evaluate method
///////////////////////////////
// Todo: select the applicable rules somehow? use IsEnabled?
// 
```

```

// Add rules
//

LM_CREATE_CHORD_RULE(lmRuleNoParallelMotion, lmCVR_NoParallelMotion)
LM_CREATE_CHORD_RULE(lmRuleNoResultingFifthOctaves, lmCVR_NoResultingFifthOctaves)
LM_CREATE_CHORD_RULE(lmRuleNoVoicesCrossing, lmCVR_NoVoicesCrossing)
LM_CREATE_CHORD_RULE(lmNoIntervalHigherThanOctave, lmCVR_NoIntervalHigherThanOctave)

void lmRuleList::CreateRules()
{
    AddRule( new lmRuleNoParallelMotion(),
        _T("No parallel motion of perfect octaves, perfect fifths, and unisons") );
    AddRule( new lmRuleNoResultingFifthOctaves(),
        _T("No resulting fifths and octaves") );
    AddRule( new lmRuleNoVoicesCrossing(),
        _T("Do not allow voices crossing. No duplicates (only for root position and root
duplicated)") );
    AddRule( new lmNoIntervalHigherThanOctave(),
        _T("Voices interval not greater than one octave (except bass-tenor)") );
}

//


// lmChordError
//
bool lmChordError::IncludesError(int nBrokenRule)
{
    if ( nBrokenRule < lmCVR_FirstChordValidationRule || nBrokenRule >
lmCVR_LastChordValidationRule)
        return false; // invalid rule
    wxLogMessage(_T("IncludesError %d , ErrList:%u , %u")
        , nBrokenRule, nErrList, (nErrList & (1 << nBrokenRule) ) != 0 );
    return (nErrList & (1 << nBrokenRule) ) != 0;
}
void lmChordError::SetError(int nBrokenRule, bool fVal)
{
    assert ( nBrokenRule >= lmCVR_FirstChordValidationRule && nBrokenRule <=
lmCVR_LastChordValidationRule);
    nErrList |= ( (fVal? 1:0) << nBrokenRule );
}

//


// class Rule
//
lmRule::lmRule(int nRuleID)
{
    m_fEnabled = true;
    m_pChordDescriptor = NULL;
    m_sDetails = _T("nothing");
    m_nRuleId = nRuleID;
};

//


// Definition of rules of harmony
//

// return number of errors
int lmRuleNoParallelMotion::Evaluate(wxString& sResultDetails, int pNumFailuresInChord[],
ChordInfoBox* pBox )
{
    sResultDetails = _T("Rule: No parallel motion ");
    if ( m_pChordDescriptor == NULL)
    {
        wxLogMessage(_T(" lmRuleNoParallelMotion: m_pChordDescriptor NULL "));
        return false;
    }
    wxColour colour( 200, 50, 50 );
    int nErrCount = 0;
    int nNumNotes;
    int nVoiceInterval[lmNOTES_IN_CHORD];
    sResultDetails = _T("");
    // Analyze all chords
}

```

Analizador tonal en software libre

```
for (int nC=1; nC<m_nNumChords; nC++)
{
    wxLogMessage(_T("Check chord %d "), (nC)+1);

    pNumFailuresInChord[nC] = 0;

    // num notes: min of both chords
    nNumNotes = (m_pChordDescriptor[nC]->GetNumNotes() < m_pChordDescriptor[nC-1]-
>GetNumNotes()) ?
                m_pChordDescriptor[nC]->GetNumNotes(): m_pChordDescriptor[nC-1]-
>GetNumNotes();
    for (int nN=0; nN<nNumNotes; nN++)
    {
        nVoiceInterval[nN] = (m_pChordDescriptor[nC]->GetNoteFpitch(nN)
                               - m_pChordDescriptor[nC-1]->GetNoteFpitch(nN) ) % lm_p8 ;

        // check if it is parallel with any previous note
        for (int i=0; i<nN; i++)
        {
            if ( nVoiceInterval[i] == nVoiceInterval[nN])
            {
                lmFIntval nInterval = abs(
                    m_pChordDescriptor[nC]->GetNoteFpitch(nN)
                    - m_pChordDescriptor[nC]->GetNoteFpitch(i) );
                int nIntervalNumber = FIntval_GetNumber(nInterval);

                wxLogMessage(_T(" >>> Check parallel motion in chord %d, notes:%d %d,
INTERVAL:%d(%s) %d"))
                    ,nC, i, nN, nIntervalNumber
                    , FIntval_GetIntvCode(nInterval).c_str()
                    , nInterval);

                if ( nIntervalNumber == 1 || nIntervalNumber == 5 )
                {
                    wxString sType = FIntval.GetName(nInterval);
                    pNumFailuresInChord[nC] = pNumFailuresInChord[nC] +1;

                    int nFullVoiceInterval = abs ( m_pChordDescriptor[nC]->GetNoteFpitch(i)
                        - m_pChordDescriptor[nC-1]->GetNoteFpitch(i) );

//TODO: accumulate messages?                                sResultDetails += wxString::Format(
                sResultDetails = wxString::Format(
                    _T("Parallel motion of %s, chords: %d, %d; v%d %s-->%s, v%d %s-->%s,
Interval: %s")
                    ,sType.c_str(), (nC-1)+1, (nC)+1
                    , m_pChordDescriptor[nC]->GetNoteVoice(i)
                    , FPitch_ToAbsLDPName(m_pChordDescriptor[nC-1]-
>GetNoteFpitch(i)).c_str()
                    , FPitch_ToAbsLDPName(m_pChordDescriptor[nC]->GetNoteFpitch(i)).c_str()
                    , m_pChordDescriptor[nC]->GetNoteVoice(nN)
                    , FPitch_ToAbsLDPName(m_pChordDescriptor[nC-1]-
>GetNoteFpitch(nN)).c_str()
                    , FPitch_ToAbsLDPName(m_pChordDescriptor[nC]-
>GetNoteFpitch(nN)).c_str()
                    , FIntval_GetIntvCode(nInterval).c_str()
                    );

                    wxLogMessage( sResultDetails );

                    if (pBox && m_pChordDescriptor[nC]->HasLmNotes())
                    {
                        pBox->DisplayChordInfo(
                            GetMainFrame()->GetActiveDoc()->GetScore()
                            , m_pChordDescriptor[nC], colour, sResultDetails);

                        // display failing notes in red TODO: this could be improved...
                        m_pChordDescriptor[nC]->GetNoteLmNote(nN)->SetColour(*wxCYAN);
                        m_pChordDescriptor[nC]->GetNoteLmNote(i)->SetColour(*wxBLUE);
                        m_pChordDescriptor[nC-1]->GetNoteLmNote(nN)->SetColour(*wxCYAN);
                        m_pChordDescriptor[nC-1]->GetNoteLmNote(i)->SetColour(*wxBLUE);

                        DrawArrow(
                            m_pChordDescriptor[nC-1]->GetNoteLmNote(nN),
                            m_pChordDescriptor[nC]->GetNoteLmNote(nN),

```

```

        wxColour(*wxRED) );
    DrawArrow(
        m_pChordDescriptor[nC-1]->GetNoteLmNote(i),
        m_pChordDescriptor[nC]->GetNoteLmNote(i),
        wxColour(*wxRED) );
    }

    m_pChordDescriptor[nC]->tChordErrors.SetError( this->GetRuleId(), true);
    nErrCount++;
}
}

}

wxLogMessage(_T(" Rule %d final error count %d"), this->GetRuleId(), nErrCount);
return nErrCount;
}

// return number of errors
int lmRuleNoResultingFifthOctaves::Evaluate(wxString& sResultDetails
                                              , int pNumFailuresInChord[], ChordInfoBox* pBox)
{
    wxString sMovTypes[] =
        {_T("Direct"), _T("Oblique"), _T("Contrary")};

    // Forbidden to arrive to a fifth or octave by means of a direct movement ( both: same delta
    sign)
    // exceptions:
    // - voice is soprano (TODO: consider: tenor, contralto??) and distance is 2th
    // - TODO: consider: fifth and one sound existed??
    sResultDetails = _T("Rule: No resulting fifth/octaves ");
    if ( m_pChordDescriptor == NULL)
    {
        wxLogMessage(_T(" lmRuleNoResultingFifthOctaves: m_pChordDescriptor NULL "));
        return 0;
    }

    int nDifColour = this->GetRuleId() * 2;    //todo: pensar forma de cambiar algo el color en cada
    regla?
    int nTransp = 128; // todo: ¿usar transparencia?
    wxColour colour( 200, 20+nDifColour, 20+nDifColour, nTransp);
    int nErrCount = 0;
    int nNumNotes;
    int nVoiceMovementType;
    // Analyze all chords
    for (int nC=1; nC<m_nNumChords; nC++)
    {
        wxLogMessage(_T("Check chords %d TO %d"), nC-1, nC);

        pNumFailuresInChord[nC] = 0;

        // num notes: min of both chords
        nNumNotes = (m_pChordDescriptor[nC]->GetNumNotes() < m_pChordDescriptor[nC-1]-
>GetNumNotes()) ?
                    m_pChordDescriptor[nC]->GetNumNotes(): m_pChordDescriptor[nC-1]-
>GetNumNotes();

        // for all the notes in the chord...
        for (int nN=0; nN<nNumNotes; nN++)
        {
            // check type of movement with any previous note
            for (int i=0; i<nN; i++)
            {
                nVoiceMovementType = GetHarmonicMovementType(
                    m_pChordDescriptor[nC-1]->GetNoteFpitch(nN), m_pChordDescriptor[nC]-
>GetNoteFpitch(nN),
                    m_pChordDescriptor[nC-1]->GetNoteFpitch(i), m_pChordDescriptor[nC]-
>GetNoteFpitch(i));

                lmFIntval nInterval = abs(
                    m_pChordDescriptor[nC]->GetNoteFpitch(nN)
                    - m_pChordDescriptor[nC-1]->GetNoteFpitch(i) );
            }
        }
    }
}

```

```

int nIntervalNumber = FIntval_GetNumber(nInterval);

wxLogMessage(_T(" Notes: %s-->%s %s-->%s Movement type:%s INTERVAL:%d (%s)")
    , FPitch_ToAbsLDPName(m_pChordDescriptor[nC-1]->GetNoteFpitch(nN)).c_str()
    , FPitch_ToAbsLDPName(m_pChordDescriptor[nC]->GetNoteFpitch(nN)).c_str()
    , FPitch_ToAbsLDPName(m_pChordDescriptor[nC-1]->GetNoteFpitch(i)).c_str()
    , FPitch_ToAbsLDPName(m_pChordDescriptor[nC]->GetNoteFpitch(i)).c_str()
    , sMovTypes[nVoiceMovementType].c_str()
    , nIntervalNumber, FIntval_GetIntvCode(nInterval).c_str());

if ( nVoiceMovementType == lm_eDirectMovement && ( nIntervalNumber == 1 ||
nIntervalNumber == 5 ) )
{
    // Incorrect, unless: voice interval is 2th and voice is > 0 (not BASS)
    lmFIntval nVoiceInterval = abs(
        m_pChordDescriptor[nC]->GetNoteFpitch(nN)
        - m_pChordDescriptor[nC]->GetNoteFpitch(i) );
    int nVoiceIntervalNumber = FIntval_GetNumber(nVoiceInterval);

    if ( nVoiceIntervalNumber == 2 && nN > 0 )
    {
        wxLogMessage(_T(" Exception!, voice not BASS and voice interval is 2th!
"));
    }
    else
    {
        wxString sType;
        if (nInterval > 80) // current limitation in FIntval.GetName
            sType = _T("higher than 2 octaves");
        else
            sType = FIntval.GetName(nInterval);

        sResultDetails = wxString::Format(
            _T("Direct movement resulting %s. Chords:%d,%d. Voices:%d %s-->%s and %d %s-->%s.
Interval: %s")
            , sType.c_str(), (nC-1)+1, (nC)+1
            , m_pChordDescriptor[nC]->GetNoteVoice(nN)
            , FPitch_ToAbsLDPName(m_pChordDescriptor[nC-1]->GetNoteFpitch(nN)).c_str()
            , FPitch_ToAbsLDPName(m_pChordDescriptor[nC]->GetNoteFpitch(nN)).c_str()
            , m_pChordDescriptor[nC]->GetNoteVoice(i)
            , FPitch_ToAbsLDPName(m_pChordDescriptor[nC-1]->GetNoteFpitch(i)).c_str()
            , FPitch_ToAbsLDPName(m_pChordDescriptor[nC]->GetNoteFpitch(i)).c_str()
            , FIntval_GetIntvCode(nInterval).c_str());

        if (pBox && m_pChordDescriptor[nC-1]->HasLmNotes() &&
m_pChordDescriptor[nC]->HasLmNotes())
        {
            DrawArrow(
                m_pChordDescriptor[nC-1]->GetNoteLmNote(nN),
                m_pChordDescriptor[nC]->GetNoteLmNote(nN),
                wxColour(*wxCYAN) );
            DrawArrow(
                m_pChordDescriptor[nC-1]->GetNoteLmNote(i),
                m_pChordDescriptor[nC]->GetNoteLmNote(i),
                wxColour(*wxCYAN) );
        }

        wxLogMessage( sResultDetails );
    }

    if (pBox && m_pChordDescriptor[nC]->HasLmNotes() )
    {
        pBox->DisplayChordInfo(GetMainFrame()->GetActiveDoc()->GetScore()
            , m_pChordDescriptor[nC], colour, sResultDetails);

        // display failing notes in red (TODO: improve error display?)
        m_pChordDescriptor[nC]->GetNoteLmNote(nN)->SetColour(*wxRED);
        m_pChordDescriptor[nC]->GetNoteLmNote(i)->SetColour(*wxRED);
    }

    m_pChordDescriptor[nC]->tChordErrors.SetError( this->GetRuleId(), true);
    nErrCount++;
}
}

```

```

        }
    }

    wxLogMessage(_T(" Rule %d final error count %d"), this->GetRuleId(), nErrCount);
    return nErrCount;
}

// return number of errors
int lmRuleNoVoicesCrossing::Evaluate(wxString& sResultDetails, int pNumFailuresInChord[]
                                     , ChordInfoBox* pBox)
{
    sResultDetails = _T("Rule: No voices crossing:");
    if ( m_pChordDescriptor == NULL )
    {
        wxLogMessage(_T(" lmRuleNoVoicesCrossing: m_pChordDescriptor NULL "));
        return 0;
    }
    int nDifColour = this->GetRuleId() * 2; //todo: consider to apply a different color for each
rule
    int nTransp = 128; // todo: consider to user transparency
    wxColour colour( 200, 20+nDifColour, 20+nDifColour, nTransp );
    int nErrCount = 0;
    int nNumNotes;
    int nVoice[2];
    int nPitch[2];
    // Analyze all chords
    for (int nC=0; nC<m_nNumChords; nC++)
    {
        pNumFailuresInChord[nC] = 0;

        // Apply rule only if:
        // chord in root position (o inversions)
        // root note is duplicated
        if ( m_pChordDescriptor[nC]->GetInversion() != 0 )
        {
            wxLogMessage(_T(" Rule not applicable: not root position: %d inversions"),
m_pChordDescriptor[nC]->GetInversion());
            continue;
        }
        if ( m_pChordDescriptor[nC]->GetInversion() == 0 && ! m_pChordDescriptor[nC]->IsBassDuplicated() )
        {
            wxLogMessage(_T(" Rule not applicable: not root position but root note not
duplicated"));
            continue;
        }

        nNumNotes = m_pChordDescriptor[nC]->GetNumNotes() ;

        // for all the notes in the chord...
        for (int nN=0; nN<nNumNotes; nN++)
        {
            // check crossing TODO: ENSURE VOICES HAVE A VALUE!!
            for (int i=1; i<nN; i++)
            {
                nVoice[1] = m_pChordDescriptor[nC]->GetNoteVoice(nN);
                nVoice[0] = m_pChordDescriptor[nC]->GetNoteVoice(i);
                nPitch[1] = m_pChordDescriptor[nC]->GetNoteFpitch(nN);
                nPitch[0] = m_pChordDescriptor[nC]->GetNoteFpitch(i);
                if ( nVoice[1] > nVoice[0] &&
                    nPitch[1] <= nPitch[0] )
                {
                    sResultDetails = wxString::Format(
                        _T("Chord:%d: Voice crossing.  Voice%d(%s) <= Voice%d(%s) ")
                        , (nC)+1
                        , nVoice[1], FPitch_ToAbsLDPName(m_pChordDescriptor[nC]-
>GetNoteFpitch(nN)).c_str()
                        , nVoice[0], FPitch_ToAbsLDPName(m_pChordDescriptor[nC]-
>GetNoteFpitch(i)).c_str()
                        );
                    wxLogMessage( sResultDetails );

                    if (pBox && m_pChordDescriptor[nC]->HasLmNotes())

```

```

    {
        pBox->DisplayChordInfo(GetMainFrame()->GetActiveDoc()->GetScore()
            , m_pChordDescriptor[nC], colour, sResultDetails);

        // display failing notes in red (TODO: mejorar indicacion de errores)
        m_pChordDescriptor[nC]->GetNoteLmNote(nN)->SetColour(*wxRED);
        m_pChordDescriptor[nC]->GetNoteLmNote(i)->SetColour(*wxRED);
    }

    m_pChordDescriptor[nC]->tChordErrors.SetError( this->GetRuleId(), true);
    nErrCount++;
}
}

wxLogMessage(_T(" Rule %d final error count %d"), this->GetRuleId(), nErrCount);
return nErrCount;
}

int lmNoIntervalHigherThanOctave::Evaluate(wxString& sResultDetails, int pNumFailuresInChord[]
                                            , ChordInfoBox* pBox)
{
    sResultDetails = _T(" Rule: no interval higher than octave");
    if ( m_pChordDescriptor == NULL)
    {
        wxLogMessage(_T(" lmNoIntervalHigherThanOctave: m_pChordDescriptor NULL "));
        return 0;
    }
    int nDifColour = this->GetRuleId() * 2;
    int nTransp = 128;
    wxColour colour( 200, 20+nDifColour, 20+nDifColour, nTransp);
    int nErrCount = 0;
    int nNumNotes;
    int nInterval;

    // Analyze all chords
    for (int nC=0; nC<m_nNumChords; nC++)
    {
        wxLogMessage(_T("Check chord %d "), nC);

        pNumFailuresInChord[nC] = 0;

        // Apply rule only if:
        // chord in root position (o inversions)
        // root note is duplicated
        if ( m_pChordDescriptor[nC]->GetInversion() != 0 )
        {
            wxLogMessage(_T(" Rule not applicable: not root position: %d inversions"),
m_pChordDescriptor[nC]->GetInversion());
            continue;
        }
        if ( m_pChordDescriptor[nC]->GetInversion() == 0 && ! m_pChordDescriptor[nC]-
>IsBassDuplicated() )
        {
            wxLogMessage(_T(" Rule not applicable: not root position bass note is not
duplicated"));
            continue;
        }

        nNumNotes = m_pChordDescriptor[nC]->GetNumNotes() ;

        // TODO: confirm: only applicable to 4 voices
        if ( nNumNotes != 4 )
        {
            wxLogMessage(_T(" Rule not applicable: not 4 notes (%d)"), nNumNotes);
            continue;
        }
        // for all the voices in the chord...
        for (int nN=1; nN<4; nN++)
        {
            lmFIntval nLimit;
            if (nN == 1)
                nLimit = lm_p8*2; // up to 2 octaves allowed for bass-tenor
        }
    }
}

```

```

        else
            nLimit = lm_p8; // only one octave allowed for the rest

        // TODO: ensure correspondance VOICE - order
        nInterval = m_pChordDescriptor[nC]->GetNoteFpitch(nN)
                    - m_pChordDescriptor[nC]->GetNoteFpitch(nN-1);

        wxLogMessage(_T(" Notes %d - %d: interval: %d "), nN, nN-1, nInterval);

        if ( nInterval > nLimit )
        {
            sResultDetails = wxString::Format(
                _T("Chord %d: Interval %s higher than octave between voices %d (%s) and %d (%s)")
                , (nC)+1
                , FIntval_GetIntvCode(nInterval).c_str()
                , m_pChordDescriptor[nC]->GetNoteVoice(nN)
                , FPitch_ToAbsLDPName(m_pChordDescriptor[nC]->GetNoteFpitch(nN)).c_str()
                , m_pChordDescriptor[nC]->GetNoteVoice(nN-1)
                , FPitch_ToAbsLDPName(m_pChordDescriptor[nC]->GetNoteFpitch(nN-1)).c_str()
            );

            wxLogMessage( sResultDetails );

            if (pBox && m_pChordDescriptor[nC]->HasLmNotes())
            {
                pBox->DisplayChordInfo(GetMainFrame()->GetActiveDoc()->GetScore()
                                         , m_pChordDescriptor[nC], colour, sResultDetails);

                // display failing notes in red (TODO: mejorar indicacion de errores)
                m_pChordDescriptor[nC]->GetNoteLmNote(nN)->SetColour(*wxRED);
                m_pChordDescriptor[nC]->GetNoteLmNote(nN-1)->SetColour(*wxRED);

                DrawArrow(
                    m_pChordDescriptor[nC]->GetNoteLmNote(nN-1),
                    m_pChordDescriptor[nC]->GetNoteLmNote(nN),
                    wxColour(*wxBLUE) );
            }
        }

        m_pChordDescriptor[nC]->tChordErrors.SetError( this->GetRuleId(), true);
        nErrCount++;
        wxLogMessage(_T(" Rule %d partial error count %d"), this->GetRuleId(), nErrCount);
    }

    wxLogMessage(_T(" Rule %d final error count %d"), this->GetRuleId(), nErrCount);
    return nErrCount;
}

// lmRuleList
// lmRuleList::~lmRuleList()
{
    // Iterate over the map and delete lmRule
    std::map<int, lmRule*>::iterator it;
    for(it = m_Rules.begin(); it != m_Rules.end(); ++it)
    {
        delete it->second;
    }
    m_Rules.clear();
}

bool lmRuleList::AddRule(lmRule* pNewRule, const wxString& sDescription )
{
    int nRuleId = pNewRule->GetRuleId();
    pNewRule->SetDescription(sDescription);

    std::map<int, lmRule*>::iterator it = m_Rules.find(nRuleId);
    if(it != m_Rules.end())
    {
}

```

```

        wxLogMessage(_T(" AddRule: Rule %d already stored !"), nRuleId);
        return false;
    }
    if ( nRuleId >= lmCVR_FirstChordValidationRule && nRuleId <= lmCVR_LastChordValidationRule)
    {
        m_Rules.insert(std::pair<int, lmRule*>(nRuleId, pNewRule));
    }
    else
    {
        wxLogMessage(_T(" AddRule: INVALID rule id: %d"), nRuleId );
    }
    return true;
}

bool lmRuleList::DeleteRule(int nRuleId)
{
    std::map<int, lmRule*>::iterator it = m_Rules.find(nRuleId);
    if(it == m_Rules.end())
    {
        wxLogMessage(_T(" DeleteRule: Rule %d not stored !"), nRuleId);
        return false;
    }
    m_Rules.erase(it);
    return true;
}
lmRule* lmRuleList::GetRule(int nRuleId)
{
    std::map<int, lmRule*>::iterator it = m_Rules.find(nRuleId);
    if(it == m_Rules.end())
        return NULL;
    else
        return it->second;
}
void lmRuleList::SetChordDescriptor(lmScoreChord** pChD, int nNumChords)
{
    // Iterate over the map and set Chord Descriptor to each item
    // Note: Use a const_iterator if we are not going to change the values
    //       for(mapType::const_iterator it = data.begin(); it != data.end(); ++it)
    std::map<int, lmRule*>::iterator it;
    for(it = m_Rules.begin(); it != m_Rules.end(); ++it)
    {
        it->second->SetChordDescriptor( pChD, nNumChords );
    }
}

```

8.1.3 TheoHarmonyCtrl.cpp

RESUMEN: Generación de los ejercicios.

URL:

<https://lenmus.svn.sourceforge.net/svnroot/lenmus/trunk/src/exercises/>

TheoHarmonyCtrl.cpp

```

//-----
// LenMus Phonascus: The teacher of music
// Copyright (c) 2002-2010 LenMus project
//
// This program is free software; you can redistribute it and/or modify it under the
// terms of the GNU General Public License as published by the Free Software Foundation,
// either version 3 of the License, or (at your option) any later version.
//
// This program is distributed in the hope that it will be useful, but WITHOUT ANY
// WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
// PARTICULAR PURPOSE. See the GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License along with this
// program. If not, see <http://www.gnu.org/licenses/>.

```

```

// For any comment, suggestion or feature request, please contact the manager of
// the project at ceciliost@users.sourceforge.net
// -----
#ifndef __GNUG__ && !defined(NO_GCC_PRAGMA)
#pragma implementation "TheoHarmonyCtrol.h"
#endif

// For compilers that support precompilation, includes <wx.h>.
#include <wx/wxprec.h>

#ifndef __BORLANDC__
#pragma hdrstop
#endif

#include "TheoHarmonyCtrol.h"

#include "../app/Processor.h"
#include "../app/toolbox/ToolNotes.h"
#include "../score/VStaff.h"
#include "../score/Instrument.h"
#include "Constrains.h"
#include "Generators.h"
#include "../auxmusic/Conversion.h"

#include "../ldp_parser/LDPParser.h"

#include "../globals/Colors.h"
extern lmColors* g_pColors;

// access to error's logger
#include "../app/Logger.h"
extern lmLogger* g_pLogger;

// access to MIDI manager to get default settings for instrument to use
#include "../sound/MidiManager.h"

// access to main frame
#include "../app/MainFrame.h"
extern lmMainFrame* GetMainFrame();

#include "../app/toolbox/ToolsBox.h"

#include "../auxmusic/HarmonyExercisesData.h"

class lmEditorMode;

// -----
// Implementation of lmTheoHarmonyCtrol
// -----
//IDs for controls
const int lmID_LINK_SETTINGS = wxNewId();
const int lmID_LINK_GO_BACK = wxNewId();
const int lmID_LINK_NEW_PROBLEM = wxNewId();

IMPLEMENT_CLASS(lmTheoHarmonyCtrol, lmFullEditorExercise)

BEGIN_EVENT_TABLE(lmTheoHarmonyCtrol, lmFullEditorExercise)
    LM_EVT_URL_CLICK    (lmID_LINK_SETTINGS, lmEBookCtrol::OnSettingsButton)
    LM_EVT_URL_CLICK    (lmID_LINK_GO_BACK, lmEBookCtrol::OnGoBackButton)
    LM_EVT_URL_CLICK    (lmID_LINK_NEW_PROBLEM, lmFullEditorExercise::OnNewProblem)
END_EVENT_TABLE()

lmTheoHarmonyCtrol::lmTheoHarmonyCtrol(wxWindow* parent, wxWindowID id,
                                       lmHarmonyConstrains* pConstrains, wxSize nDisplaySize,
                                       const wxPoint& pos, const wxSize& size, int style)
: lmFullEditorExercise(parent, id, pConstrains, pos, size, style )
{
    //initializations
    m_pConstrains = pConstrains;
    m_pProblemScore = (lmScore*)NULL;
}

```

```

        CreateControls();
    }

lmTheoHarmonyCtrol::~lmTheoHarmonyCtrol()
{
    //AWARE: As score andEditMode ownership is transferred to the Score Editor window,
    //they MUST NOT be deleted here.
}

lmEditorMode* lmTheoHarmonyCtrol::CreateEditMode()
{
    //This method is invoked each time a new problem is generated, in order to define
    //editor customizations for the created exercise. Ownership of created lmEditorMode object
    //is transferred to the document and deleted there.

    lmEditorMode* pEditMode = new lmEditorMode( CLASSINFO(lmScoreCanvas),
CLASSINFO(lmHarmonyProcessor) );
    pEditMode->ChangeToolPage(lmPAGE_NOTES, CLASSINFO(lmToolPageNotesHarmony) );
    pEditMode->SetModeName(_T("TheoHarmonyCtrol"));
    pEditMode->SetModeVers(_T("1"));

    return pEditMode;
}

wxDialog* lmTheoHarmonyCtrol::GetSettingsDlg()
{
    //Returns a pointer to the dialog for customizing the exercise.

    //TODO: Create the dialog class and implement it. The uncomment following code:
    //wxDialog* pDlg = new lmDlgCfgTheoHarmony(this, m_pConstrains, m_pConstrains->IsTheoryMode());
    //return pDlg;
    return (wxDialog*)NULL;
}

void lmTheoHarmonyCtrol::SetNewProblem()
{
    //This method creates a problem score, satisfying the restrictions imposed
    //by exercise options and user customizations.

    // TODO: Possible exercise options:
    //       exercise 1,2:
    //           inversions allowed
    //           elision allowed?
    //           allowed random key signature (if not: use fixed C major)
    //

    // TODO: Possible improvements:
    //       generalize for chords of N notes
    //       make "number of measures" dependant from the "key signature"?
    //       calculate numerals from chord info + key signature + mode

    //

    // all-exercises generic data
    const int lmNUM_HARMONY_EXERCISES = 3;
    const int nNUM_INTERVALS_IN_N_HARMONY_EXERCISE = 2;
    int nNumMeasures = 2;
    // each-exercise specific data
    wxString sExerciseDescription;
    wxString sNote = _T("q"); // todo: calculate note note duration from time signature
    wxString sLDPGoBack = wxString::Format(_T("(musicData (goBack %s) )"), sNote.c_str());

    // Carlos jun-09
    // Three types of problem
    // 1) fixed bass
    // 2) fixed soprano
    // 3) chord notation

    // select a random exercise type
    lmRandomGenerator oGenerator;
//@todo provisionalmente cambiado:    nHarmonyExcerciseType = oGenerator.RandomNumber(1,
lmNUM_HARMONY_EXERCISES);
    static int nEx = 0;
    if (++nEx > lmNUM_HARMONY_EXERCISES)
        nEx = 1;
    nHarmonyExcerciseType = nEx; //@ cada vez un tipo distinto
}

```

```

// todo: inversions allowed: make it an exercise option
if ( nHarmonyExcerciseType == 3)
    bInversionsAllowedInHarmonyExercises = true;
else
    bInversionsAllowedInHarmonyExercises = false;

wxString sPattern;
lmLDPParser parserLDP;
lmLDPNode* pNode;
lmVStaff* pVStaff;
lmNote* pNoteToAttach = NULL;
wxString sExerciseTitle;
bool bDebugDisplayHiddenNotes = false; // activate only for debug

if ( nHarmonyExcerciseType >= 1 && nHarmonyExcerciseType <= lmNUM_HARMONY_EXERCISES )
{
    // Prepare a score that meets the restrictions

    m_nKey = oGenerator.GenerateKey( m_pConstrains->GetKeyConstrains() );

    if (nHarmonyExcerciseType == 3)
        sExerciseDescription = wxString::Format(_T(" Cipher the chords"));
    else
        sExerciseDescription = wxString::Format(
            _T(" Fixed %s; root position. Complete the chord notes."),
            , (nHarmonyExcerciseType == 1? _T("bass"): _T("soprano")) );

    sExerciseTitle = wxString::Format(_T(" Exercise type %d : %s "))
        , nHarmonyExcerciseType, sExerciseDescription.c_str());

    //create a score with a bass line

    wxLogMessage(_T(" CLEARING DATA "));
    for (int nC=0; nC < nMAX_HARMONY_EXERCISE_CHORDS; nC++)
    {
        if (pHE_Chords[nC] != NULL)
        {
            wxLogMessage(_T(" deleting chord %d"), nC);
            //@ todo: to do? delete pHE_Chords[nC];
            pHE_Chords[nC] = NULL;
        }
        for (int nV=0; nV < nNUM_VOICES_IN_HARMONY_EXERCISE; nV++)
        {
            if (pHE_Notes[nC][nV] != NULL)
            {
                wxLogMessage(_T(" deleting note chord %d v:%d"), nC, nV);
                //@ todo: to do? delete pHE_Notes[nC][nV];
                pHE_Notes[nC][nV] = NULL;
            }
            nHE_NotesFPitch[nC][nV] = 0;
            sHE_Notes[nC][nV] = _T("");
            sHE_Pattern[nC][nV] = _T("");
        }
    }

    m_pProblemScore = new_score();
    lmInstrument* pInstr = m_pProblemScore->AddInstrument(
        g_pMidi->DefaultVoiceChannel(),
        g_pMidi->DefaultVoiceInstr(), _T(""));
}

pVStaff = pInstr->GetVStaff(); //add second staff: five lines, standard size
pVStaff->AddStaff(5); //G clef on first staff
pVStaff->AddClef( lmE_Sol, 1 );
pVStaff->AddClef( lmE_Fa4, 2 ); //F clef on second staff
pVStaff->AddKeySignature( m_nKey ); //key signature
pVStaff->AddTimeSignature(2,4); //2/4 time signature

lmFontInfo tNumeralFont = { _T("Times New Roman"), 11, wxFONTSTYLE_NORMAL,
                           wxFONTWEIGHT_BOLD };
lmTextStyle* pNumeralStyle = m_pProblemScore->GetStyleName(tNumeralFont);

wxString sNumeralsDegrees[7] =
{ _T(" I"), _T(" II"), _T(" III"), _T(" IV"), _T(" V"), _T(" VI"), _T(" VII") };
wxString sNumerals;

```

```

//loop the add notes
int nChordCount = 0;
int nOctave;
int nVoice;
int nRootNoteStep;
int nStaff;
for (int iN=0; iN < (nNumMeasures*2); iN+=2)
{
    //add barline for previous measure
    if (iN != 0)
        pVStaff->AddBarline(lm_eBarlineSimple);
    else
        pVStaff->AddSpacer(20);

    //two chords per measure (time signature is 2 / 4)
    for (int iM=0; iM < 2; iM++)
    {
        // Generate chords with no harmonic progression errors:
        // Loop:
        //     generate a random chord
        //     create voices in FPitch, obviously matching the chord notes
        // Until chord ok: no errors
        // With the new chord:
        //     Create figured bass
        //     Calculate numeral
        //     Create the score notes for the voices
        //
        // Display
        //     Exercise 1: bass note and numeral
        //     Exercise 2: soprano note and numeral
        //     Exercise 3: all notes and numeral
        //

        int nNumChordLinkErrors = -1;
        int nAttempts = 0;
        int nMaxAttempts = 20;
        int nInversions = 0;
        // try to create a new chord until no link error with previous chords
        pHE_Chords[nChordCount] = 0;
        wxLogMessage(_T(" ===== START WITH CHORD %d ===== "), nChordCount );
        while ( nNumChordLinkErrors != 0 && nAttempts < nMaxAttempts)
        {
            if (nAttempts)
                wxLogMessage(_T(" ***** NEW ATTEMPT (%d) for CHORD %d *****"), nAttempts,
nChordCount );
            // Root note
            nOctave = oGenerator.RandomNumber(2, 3);
            // this is done to make the notes appear more centered in the bass staff
            if (nOctave == 3) // octave 3 : notes c,d,e
                nRootNoteStep = oGenerator.RandomNumber(0, 2);
            else // octave 2 : notes f,g,a,b
                nRootNoteStep = oGenerator.RandomNumber(3, 6);

            nInversions = 0;
            if (bInversionsAllowedInHarmonyExercises)
            {
                // Calculate a random number of inversions and apply them
                nInversions = oGenerator.RandomNumber(0,
nNUM_INTERVALS_IN_N_HARMONY_EXERCISE );
            }

            if (pHE_Chords[nChordCount])
            {
                if ( nAttempts == 0 )
                {
                    wxLogMessage(_T(" @@ ERROR: impossible: chord no empty but 0 previous
attempts " ) );
                }
                else
                {
                    wxLogMessage(_T(" deleting chord %d of previous attempt "), nChordCount
);
                }
                delete pHE_Chords[nChordCount];
                pHE_Chords[nChordCount] = 0;
            }
        }
    }
}

```

```

        }

        //
        // create the chord
        //
        wxLogMessage(_T("Creating_lmScoreChord: step:%d octave:%d inversions:%d key:
%d"))

            , nRootNoteStep, nOctave, nInversions, m_nKey );
pHE_Chords[nChordCount] = new lmScoreChord(nRootNoteStep, m_nKey
            , nNUM_INTERVALS_IN_N_HARMONY_EXERCISE, nInversions, nOctave);

        //
        // This is the bass voice (root note)
nHE_NotesFPitch[nChordCount][nBassVoiceIndex] = pHE_Chords[nChordCount]-
>GetNote(0);
        //
        // but... additional limitation: bass note might be too high (if there are 2
inversions for example)
        //
        // we set a limit in d3
        if (nHE_NotesFPitch[nChordCount][nBassVoiceIndex] > FPitchK(lmSTEP_D, 3,
m_nKey) )
{
    nHE_NotesFPitch[nChordCount][nBassVoiceIndex] -= lm_p8;
    wxLogMessage(_T(" Bass reduced one octave to : %d ")
            , nHE_NotesFPitch[nChordCount][nBassVoiceIndex]
    );
}

wxLogMessage(_T(" Bass voice V%d,FINAL: %d (%s)")
            , nBassVoice
            , nHE_NotesFPitch[nChordCount][nBassVoiceIndex]
            , FPitch_ToAbsLDPName(nHE_NotesFPitch[nChordCount]
[nBassVoiceIndex]).c_str());

        //
        // Bass voice final value calculated
        // Set the bass voice note in the chord
pHE_Chords[nChordCount]->AddNoteLmFPitch(nHE_NotesFPitch[nChordCount]
[nBassVoiceIndex]);

        wxLogMessage(_T("<><> CHORD %d: bass STEP:%d, octave:%d, key:%d inversions:%d
ROOT:%d (%s) ==="))
            , nChordCount, nRootNoteStep, nOctave, m_nKey, nInversions
            , FPitchK(nRootNoteStep, nOctave, m_nKey)
            , FPitch_ToAbsLDPName( nHE_NotesFPitch[nChordCount]
[nBassVoiceIndex]).c_str()
            );
        wxLogMessage(_T("\t lmChord: [%s]"), pHE_Chords[nChordCount]-
>ToString().c_str());

        //
        // Create the rest of voices: 3,2,1 (tenor, baritone, soprano)
        // by adding intervals (and octaves) to the bass
        //

        //
        // There are three possible values, based on intervals, for each voice:
        // 1: bass + N octaves (0 interval: duplicate root)
        // 2: bass + N octaves + 1st interval
        // 3: bass + N octaves + 2nd interval
        //
        // Notes:
        // if there is 5th elided: do not apply rule 3
        // if root note is NOT duplicated: do not apply rule 1
        // For this exercise we intend to generate 'normal' (not very strange) chords,
so
        //
        // better to calculate each voice using all 3 rules and not repeating any of
them
        //
        // (it implies duplicate always the root)
lmFPitch nIntvB[3];
nIntvB[0] = oGenerator.RandomNumber(0, 2);
nIntvB[1] = (nIntvB[0] + oGenerator.RandomNumber(1, 2)) % 3 ;
nIntvB[2] = 3 - (nIntvB[0] + nIntvB[1] ) ;
wxLogMessage(_T("\t\t nIntvB %d %d %d"), nIntvB[0], nIntvB[1], nIntvB[2]);

int ni = 0; // index to nIntvB
for (nVoice = nNUM_VOICES_IN_HARMONY_EXERCISE-1; nVoice>=1; nVoice--,ni++)
{
    assert(pHE_Chords[nChordCount] != NULL);
}

```

```

        int nVoiceIndex = nVoice - 1;

        // start with BASS NOTE
        nHE_NotesFPitch[nChordCount][nVoiceIndex] = nHE_NotesFPitch[nChordCount]
[nBassVoiceIndex];

        // Apply the calculated intervals
        if (nIntvB[ni])
        {
            nHE_NotesFPitch[nChordCount][nVoiceIndex] += pHE_Chords[nChordCount]-
>GetInterval(nIntvB[ni]);
        }

        wxLogMessage(_T(" \tV%d, after applying interval %d : %d (%s), [ni:%d]")
        , nVoice
        , nIntvB[ni]
        , nHE_NotesFPitch[nChordCount][nVoiceIndex]
        , FPitch_ToAbsLDPName(nHE_NotesFPitch[nChordCount]
[nVoiceIndex]).c_str()
        , ni
        );

        // Limitation: a voice can not be lower than the previous...
        while (nHE_NotesFPitch[nChordCount][nVoiceIndex+1] >=
nHE_NotesFPitch[nChordCount][nVoiceIndex])
        {
            nHE_NotesFPitch[nChordCount][nVoiceIndex] += lm_p8;
            wxLogMessage(_T(" Added octave to voice V%d: %d ")
            , nVoice
            , nHE_NotesFPitch[nChordCount][nVoiceIndex]);
        }

        // additional limitation: increase tenor voice a octave if bass is low and
tenor is close to bass
        // to avoid voice distance problems
        // aware: tenor-bass is the only consecutive voices allowed to have a
distance higher than octave
        if (nVoiceIndex == nTenorVoiceIndex && nOctave < 3
        && (nHE_NotesFPitch[nChordCount][nTenorVoiceIndex] -
nHE_NotesFPitch[nChordCount][nBassVoiceIndex]) <= lm_M3 )
        {
            nHE_NotesFPitch[nChordCount][nTenorVoiceIndex] += lm_p8;
            wxLogMessage(_T(" Raise Tenor: added octave to voice V%d: %d ")
            , nVoice
            , nHE_NotesFPitch[nChordCount][nTenorVoiceIndex]
            );
        }

        // additional limitation: baritone voice should be in upper staff (aprox.
octave should be > 3)
        const int fUpperStaffLimit = (lm_p8*4)-lm_M3;
        // aware: do not raise more than one octave; otherwise a rule is broken
(octave distance)
        if (nVoiceIndex == nBaritoneVoiceIndex && nHE_NotesFPitch[nChordCount]
[nBaritoneVoiceIndex] < fUpperStaffLimit )
        {
            nHE_NotesFPitch[nChordCount][nBaritoneVoiceIndex] += lm_p8;
            wxLogMessage(_T(" Raise to 2nd staff: added octave to voice V%d: %d
(min:%d ")
            , nVoice
            , nHE_NotesFPitch[nChordCount][nBaritoneVoiceIndex]
            , fUpperStaffLimit);
        }

        wxLogMessage(_T("V%d,      FINAL: %d (%s)")
        , nVoice
        , nHE_NotesFPitch[nChordCount][nVoiceIndex]
        , FPitch_ToAbsLDPName(nHE_NotesFPitch[nChordCount]
[nVoiceIndex]).c_str()));

        // set this note in the chord
        pHE_Chords[nChordCount]->AddNoteLmFPitch(nHE_NotesFPitch[nChordCount]
[nVoiceIndex]);

    } // for voice...
}

```

```

// check harmonic progression errors
wxLogMessage(_T("###BEFORE %d ANALYSIS OF CHORD LINK ERRORS OF CHORD %d: %s")
    , nAttempts, nChordCount, pHE_Chords[nChordCount]->ToString().c_str());
nNumChordLinkErrors = AnalyzeHarmonicProgression(pHE_Chords, nChordCount+1, 0);

wxLogMessage(_T("##RESULT: %d LINK ERRORS"), nNumChordLinkErrors);

if (nNumChordLinkErrors == 0)
    wxLogMessage(_T("<<<< CHORD %d OK!!! after %d attempts >>>>>>"))
        , nChordCount, nAttempts, nNumChordLinkErrors);

nAttempts++;

if (nAttempts > nMaxAttempts)
{
    wxLogMessage(_T("\n **** CHORD %d TO MANY TRIES *****\n"), nChordCount);
}

// Calculate the figured bass
//
// build a chord from a list of notes in LDP source code
// lmChord(int nNumNotes, wxString* pNotes, lmEKeySignatures nKey = earmDo);
pHE_FiguredBass[nChordCount] = new lmFiguredBass(pVStaff, lmNEW_ID
    , pHE_Chords[nChordCount], m_nKey);

wxLogMessage(_T(" FIGURED BASS:%s")
    , pHE_FiguredBass[nChordCount]->GetFiguredBassString().c_str());

// At this point we already have the notes in FPitch. Now:
// Create the notes of the score
// Display notes and numerals

//
// Create each lmNote
//
wxString sUpDown[2] = { _T("up"), _T("down") };
nVoice=1; // Voices 1(soprano) (index:0) to 4(bass) (index:3)
int nNumDisplayedNotesInChord = 0;
for (nStaff=1; nStaff<3; nStaff++) // V1 (soprano) in P1 (upper staff)
{
    for (int nVoiceInStaff=1; nVoiceInStaff<3; nVoiceInStaff++, nVoice++)
    {
        assert(nVoice >= 1 && nVoice <= nNUM_VOICES_IN_HARMONY_EXERCISE);

        // aware: bass note: voice 4
        int nVoiceIndex = nVoice-1;
        sHE_Notes[nChordCount][nVoiceIndex] = wxString::Format(_T("%s")
            , FPitch_ToAbsLDPName(nHE_NotesFPitch[nChordCount]
                , [nVoiceIndex]).c_str());

        // Calculate the pattern required to create the note in the score
        sHE_Pattern[nChordCount][nVoiceIndex] = wxString::Format(_T("(n %s %s p%d v
%d (stem %s))")
            , sHE_Notes[nChordCount][nVoiceIndex].c_str()
            , sNote.c_str()
            , nStaff, nVoice, sUpDown[nVoiceInStaff-1].c_str());

        wxLogMessage(_T("\t Staff %d, V%d (index:%d) %d [%s], pattern: %s")
            , nStaff
            , nVoice
            , nVoiceIndex
            , nHE_NotesFPitch[nChordCount][nVoiceIndex]
            , sHE_Notes[nChordCount][nVoiceIndex].c_str()
            , sHE_Pattern[nChordCount][nVoiceIndex].c_str()
        );
    }
}

// Display the notes in the score
// Exercise 1: only bass (voice 1)
// Exercise 2: only soprano (voice 4)
// Exercise 3: all
// Go back:
// Exercise 1, 2: Never (only one voice is displayed per chord)

```

```

// Exercise 3: go back after voices 1,2,3

if (
    (nHarmonyExcerciseType == 1 && nVoice == nBassVoice ) ||
    (nHarmonyExcerciseType == 2 && nVoice == nSopranoVoice ) ||
    nHarmonyExcerciseType == 3
)
{
    if ( nNumDisplayedNotesInChord > 0 && nNumDisplayedNotesInChord <
nNUM_VOICES_IN_HARMONY_EXERCISE)
    {
        pNode = parserLDP.ParseText( sLDPGoBack );
        parserLDP.AnalyzeMusicData(pNode, pVStaff);
    }
    pNode = parserLDP.ParseText( sHE_Pattern[nChordCount][nVoiceIndex] );
    pHE_Notes[nChordCount][nVoiceIndex] = parserLDP.AnalyzeNote(pNode,
pVStaff);

    nNumDisplayedNotesInChord++;
    pHE_Notes[nChordCount][nVoiceIndex]->SetVoice(nVoice);
    pNoteToAttach = pHE_Notes[nChordCount][nVoiceIndex];
    wxLogMessage(_T(" V:%d added pattern: %s ***"),
        nVoice, sHE_Pattern[nChordCount][nVoiceIndex].c_str());

    // Set the actual lmNote in the chord
    pHE_Chords[nChordCount]->SetLmNote(pNoteToAttach);

    if ( bDebugDisplayHiddenNotes ) // only for the debug
    {
        wxCouleur myBlue( 10, 10, 200);
        pNoteToAttach->SetColour(myBlue);
        wxLogMessage(_T(" Ej:%d V:%d AZUL"), nHarmonyExcerciseType,
nVoice);
    }
    else     wxLogMessage(_T(" Ej:%d V:%d NEGRO"), nHarmonyExcerciseType,
nVoice);
    else     wxLogMessage(_T(" Ej:%d V:%d OCULTA"), nHarmonyExcerciseType,
nVoice);
}
}

//      Display the numeral, according the root step
//todo remove          lmStepType nRootStep = FPitch_Step(pHE_Chords[nChordCount]-
>GetNormalizedRoot());
lmStepType nRootStep = pHE_Chords[nChordCount]->GetChordDegree();
// todo: use GetDegreeString( pHE_Chords[nChordCount]->GetChordDegree() )
lmTextItem* pNumeralText = new lmTextItem(
    pNoteToAttach, lmNEW_ID, sNumeralsDegrees[nRootStep],
    lmHALIGN_DEFAULT, pNumeralStyle);
pNoteToAttach->AttachAuxObj(pNumeralText);
pNumeralText->SetUserLocation(0.0f, 230.0f );

wxLogMessage(_T("FINAL_CHORD %d: %s")
    , nChordCount, pHE_Chords[nChordCount]->ToString().c_str());

nChordCount++;
}
}
nHarmonyExerciseChordsToCheck = nChordCount;
}
wxLogMessage(_T(" CREATED EXERCISE %d with %d chords")
    , nHarmonyExcerciseType, nHarmonyExerciseChordsToCheck);
for (int i=0; i<nHarmonyExerciseChordsToCheck; i++)
    wxLogMessage(_T(" CHORD %d: %s"), i, pHE_Chords[i]->lmFPitchChord::ToString().c_str());

//add final barline
pVStaff->AddBarline(lm_eBarlineEnd);

lmFontInfo tTitleFont = {_T("Times New Roman"), 10, wxFONTSTYLE_NORMAL,
    wxFONTWEIGHT_BOLD };
lmTextStyle* pTitleStyle = m_pProblemScore->GetStyleName(tTitleFont);
lmScoreTitle* pTitle = m_pProblemScore->AddTitle(sExerciseTitle, lmHALIGN_CENTER, pTitleStyle);
lmLocation tTitlePos = g_tDefaultPos;

```

```

pTitle->SetUserLocation(tTitlePos); // only necessary if wanted to be positioned at a specific
point

//set the name and the title of the score
m_pProblemScore->SetScoreName( sExerciseTitle );
}

void lmTheoHarmonyCtrol::OnSettingsChanged()
{
    //This method is invoked when user clicks on the 'Accept' button in
    //the exercise setting dialog. You receives control just in case
    //you would like to do something (i.e. reconfigure exercise displayed
    //buttons to take into account the new exercise options chosen by the user).

    //In this exercise there is no needed to do anything
}

void lmTheoHarmonyCtrol::InitializeStrings()
{
    //This method is invoked only once: at control creation time.
    //Its purpose is to initialize any variables containing strings, so that
    //they are translated to the language chosen by user. Take into account
    //that those strings requiring translation can not be statically initialized,
    //as at compilation time we know nothing about desired language.

    //In this exercise there is no needed to translate anything
}

```

8.2 LenMus score notation language: LDP

8.2.1 Preface

LDP language is an external representation, as text, of LenMus scores. LenMus aim is not that the user has to enter scores by using the LDP language, but to offer a full featured graphical interface to enter and edit the scores. Therefore, users don't need to know anything about the LDP language. Nevertheless, for users, LDP can be of help to prepare scores as text files to be later processed and finished using the LenMus program. In fact, this is the only possibility at current program version (3.1).

8.2.2 The LDP language

The LenMus LDP notation language is a general purpose formal language for representing music scores in a platform independent, plain-text and human-readable way. This language is named LDP because I started this project in Spanish and I named this language after the Spanish acronym for 'Lenguaje de Descripción de Partituras, that is 'Language to Describe Scores'.

LDP is a pragmatic approach and its development has the direct benefit of forcing me to study in depth the problems of representing music in textual form. Although the internal language for LenMus project scores is LDP, the LenMus program will include facilities to import and export music scores to other formats, such as MusicXML.

8.2.3 A very simple introductory score

Let's see how a very simple score is encoded. The score will be just one measure, in G clef, time signature will be C major (no accidentals) and time signature will be 4 by 4. The measure will have a whole c note.

In LDP this score is described as follows:

```
(score
  (vers 1.5)
  (instrument
    (musicData
      (clef G)
      (key C)
      (time 4 4)
      (n c4 w)
      (barline)
    )
  )
)
```

Let's analyse this code. First note that in LDP all the information is structured as *elements*. An *element* is a list of keywords and data values enclosed in parenthesis. An element always starts with a keyword (i.e. vers) and it is followed by data items. These data items can be just simple data (single word values, such as '1.5' or 'G') or complex data: other elements -- those of you with an information science background will note that this is a LISP like syntax --

In the example, we can note that all the score is a list starting with the keyword 'score'. The first data item of this element is also a list, '(vers 1.5)': it starts with the keyword 'Vers' and has only a simple data item, the number 1.5.

Blank space, indentation, tabs and line breaks have no meaning: they are just a visual help for humans to improve readability, and you can use them at your desire or not use them at all. For example, the previous score can be written in a single line as:

```
(score (vers 1.5) (instrument (musicData ( ... )))
```

Or in two lines, for example as:

```
(score (vers 1.5) (instrument
  (musicData ( ... )))
```

The only rule is that you can not break keywords or simple data values. So you can not write, for example,

```
(score ( ... ))
```

LDP language is case sensitive, so for example, "score", "Score" and "SCORE" will not be taken as the same token.

To reduce the work for typing a score or to improve legibility in some cases an *abbreviated* syntax is allowed. This abbreviation consists on either:

- For elements with just one parameter, the syntax "(name param)" can be also written as "name:param". For example, the following element

```
(title right "Franz Schubert" (x 40mm) (y 30mm))
```

can also be written as:

```
(title right "Franz Schubert" x:40mm y:30mm)
```

- For elements with more than one parameter or often used, in certain cases it is allowed an abbreviated syntax, consisting, usually, in suppressing the parenthesis and reducing the element to a string. For example, in case of element **g**, beamed group, writing (g +) is equivalent to writing just g+, or in case of element **t**, tuplet, writing (t + 5 6) is equivalent to writing just t5/6.

8.2.4 Comments

To improve human legibility and understanding it is allowed to include comments. Two consecutive slash characters (//) will be interpreted as the start of a comment and all text until the end of the line will be ignored. For example:

```
// This is a score with comments
(score
  (vers 1.5)           // version 1.5 of LDP is used
  (instrument          // Start of music for the first voice
    (musicData         // start this voice
      (clef G)        // Clef: G on 2nd line
      (key C)         // Key signature: C major
```

```
(time 4 4)          // Time signature: 4 4
(n c4 w)           // Note. Pitch: C4, duration: w (whole note)
(barline)          // a simple barline
)
)
)
```

8.2.5 Language

LDP supports to use a customized set of tags for each language, so if, for example, you would prefer to encode the scores using tags in Spanish instead of tags in English, you would write:

```
(score                               //the score and language tags are the only
non-translatable
  (language es ISO-8859-1)        //specifies to use the Spanish set of tags
  (vers 1.5)
  (instrumento
    (voz
      (clave Sol)            //note the keyword 'clave' instead of 'clef'
      (tonalidad Do)
      (metrica 4 4)
      (n c4 r)              // duration: r (redonda = whole note)
      (barra)
    )
  )
)
```

8.2.6 Notation used

To describe the language, Backus Naur like form (BNF) syntax rules will be used:

- Squared brackets are placed around optional elements.
- Curly brackets are placed to enclose a list of alternatives. Elements within this list are considered exclusive and are separated by vertical bars.
- Repeatable arguments will be followed by an asterisk, meaning 'zero or more repetitions' or by a plus sign, meaning 'one or more repetitions'.

- A symbol in bold face like **this** one means a terminal symbol, that is, a string that must be written exactly as it appears here. Apart of using bold face, if this manual is printed or displayed using colours, dark blue colour will be used for terminal symbols.
- A symbol like *this* in italics means a non-terminal symbol (a category name). It must be replaced by a specific element of that category. If colour is possible they will appear in green colour.

8.2.7 Music description

8.2.7.1 *The Score, Language and Version elements*

A Score is, basically, a collection of Instruments and some optional information such as the title or the author.

```
[1 Score :: (score Language Version [Title]*  
]  
= Instrument*)
```

In case you would like to use a different set of tags than the English set or to use non Western characters (ISO-8859-1) the Language element allows to specify these characteristics.

```
[2] Language :: (language LanguageCode Charset)  
=  
[3] Charset :: Any ISO charset code name, i.e. "ISO-8859-1"  
=  
[4] LanguageCode :: A language code from ISO 639 (perhaps  
= extended with a country code from ISO 3166,  
as en-US)
```

Examples:

(language es iso-8859-1)	<i>use the Spanish set of tags</i>
(language en iso-8859-1)	<i>use the English set of tags</i>

The Version element is just a number to indicate the LDP language version used to encode the score:

```
[5] Version :: (vers VersNumber)
      =
[6] VersNumber :: An string identifying the version, i.e. 1.5
      =
```

For example:

```
(vers 1.5)      the score is encoded using LDP version 1.5
```

8.2.7.2 The Instrument element

An Instrument is a collection of one or more staves (i.e., grand-staff) describing the music.

```
[7] Instrument :: (instrument [Name [Abbreviation]])
      = [FirstSystemIndent] [MIDI_Info]
          [Staves] [MusicData])

[8] Name :: (name TextString [Font] [Location]))
      =
[9] Abbreviation :: (abbrev TextString [Font] [Location])
      = )

[10 FirstSystemIndent :: x relative location
    ]
      =
```

Instruments have usually a name. Optionally, also an abbreviated name; in these cases the name will be used at the start of the first system, and the abbreviation for succeeding systems.

The musical content is described by MusicData elements. Polyphony is possible, as notes, rests and other elements can be assigned to different voices. Any voice can move across all staves of the instrument.

Examples: Two instruments, first one on single staff and second with two staves (grand-staff):

```
(instrument (name "Flute") (musicData ...))
```

```
(instrument (staves 2) (name "Piano") (musicData ...))
```

By default, the instrument will use a 5-line staff. This can be modified using the Staves element, which describes the type and layout of the staves to use for the instrument:

```
[11 Staves ::= (staves {num | overlayered })  
]
```

For now the Staves element is just the number of five lines staves to use. One is the default value if no 'Staves' element is specified. For Grand-staff 2 must be coded. In future versions additional options will be included to allow for a full description of any type of desired staff layout.

Instead of a number the keyword **overlayered** can be used. This is useful for those cases in which it is required than multiple instruments share the same staff. Percussion parts or piano reduction scores are examples of these cases.

Warning

Command **overlayered** is not yet implemented:

If the instrument name is specified, the first system is automatically indented to print it. If no name specified or if you desire a different indentation space, it is possible to specify the desired indentation by using an x displacement element. If relative units are specified (tenths) it will be always assumed than they refer to first staff.

Examples:

First system indentation: 20mm and name 'Flute' in this space:

```
(instrument name="Flute" (dx 20mm) (musicData ...))
```

First system indentation: 20mm and no name

```
(instrument (dx 20mm) (musicData ...))
```

8.2.7.3 The MusicData element

The MusicData element is just a collection of different objects, mainly Notes and Rests, but also other elements, such as Clefs or KeySignatures, are possible.

```
[12 MusicData ::= { Note | Rest | Chord | Clef |  
= }
```

```
]KeySignature | TimeSignature |
Barline | Other }*
```

Examples:

Single note:

```
(musicData (n c4 q))
```

C major scale:

```
(musicData (n c4 q) (n d4 q) (n e4 q) (n f4 q) (n g4 q) (n a4 q) (n b4 q) (n c5 q))
```

Short fragment with a measure:

```
(musicData (clef treble) (key C) (time 2 4) (n c4 q) (n e4 q) (n g4 q) (n c5 q)
(barline))
```

8.2.7.4 The Clef element

Setting or changing the clef is done with the clef element:

```
[13 MusicData :: (clef ClefName [Tessitura] [noVisible])
```

```
]  
=
```

```
[14 Tessitura :: { +8va | -8va | +15ma | -15ma }
```

```
]  
=
```

The tessitura implied by a clef can be modified by including a tessitura option, that will draw a 8va or 15ma symbol above (+8va, +15va) or below (-8va, -15ma) the clef, and force the corresponding octave shift. The tessitura option is only usable with G and F clefs.

Warning

Tessitura option is not yet supported.

Examples:

```
(clef G)           G clef
(clef bass)        F clef
```

(clef treble +8va) *G clef with 8va symbol above*

Valid clef names are described in the following table:

ClefName	Meaning
treble, G, G2, violin	G clef on 2nd line
bass, F, F4	F clef on 4th line
baritone, F3	F clef on 3rd line
soprano, C1	C clef on 1st line
mezzosoprano, C2	C clef on 2nd line
alto, C3	C clef on 3rd line
tenor, C4	C clef on 4th line
percussion	Percussion clef

8.2.7.5 The Key element

Setting or changing the key signature is done with the key element:

```
[15 KeySignature :: (key KeyName [noVisible])  
]  
=
```

Examples:

(key F+)	<i>F sharp major key</i>
(key f+)	<i>F sharp minor key</i>
(key d-)	<i>D flat minor key</i>
(key C)	<i>C major key</i>

Valid values for tag KeyName are described in the following table:

KeyName	Key signature	KeyName	Key signature
C	C major	a	A minor
G	G major	e	E minor
D	D major	b	B minor
A	A major	f+	F sharp minor
E	E major	c+	C sharp minor
B	B major	g+	G sharp minor
F+	F sharp major	d+	D sharp minor
C+	C sharp major	a+	A sharp minor
C-	C flat major	a-	A flat minor
G-	G flat major	e-	E flat minor
D-	D flat major	b-	B flat minor
A-	A flat major	f-	F flat minor
E-	E flat major	c-	C flat minor
B-	B flat major	g-	G flat minor
F	F major	d-	D flat minor

8.2.7.6 The Time Signature element

Setting or changing the time signature is done with the TimeSignature element:

```
[16 TimeSignature :: (time      TopNumber      BottomNumber
]                      = [noVisible] )
```

Examples:

```
(time 2 4)
(time 3 8)
(time 5 8)
```

8.2.7.7 The Barline element

Element 'barline' is used to specify a barline at the end of a measure. You can specify a barline to be 'invisible' (that is, it will not be rendered). If the type of barline is not specified it is assumed to be a normal barline (simple line).

```
[17 Barline :: { barline [ BarlineType ] [ noVisible ] ) |
] = BarlineSign }

[18 BarlineType :: { simple | double | end | start | startRepetition
] = | endRepetition | doubleRepetition }

[19 BarlineSign :: { ||||| |||| :|||:: }
] =
```

To abbreviate writing barlines the Barline element can be either a full element or a graphical sign (as in abc language).

Example:

```
(n c4 e) (n e4 e) (barline end)
nc4e ne []
```

8.2.8 Notes and rests

Rests can be considered as notes with no pitch, so in LDP notes and rests are instances of a more general concept: *noterests*. A noterest is either a note or a rest.

The minimum information required to represent notes is its pitch and its duration. For rests only its duration is needed.

```
[20 Note ::= (n Pitch Duration [Other])
]

[21 Rest ::= (r Duration [Other])
```

]

8.2.8.1 Pitch

The pitch is represented by combining the note name, the octave and the accidentals.

The *note name* is a letter that represents the diatonic step:

```
[22    Pitch ::= { c | d | e | f | g | a | b }  
]  
]
```

The *octave* is an integer number indicating the octave of the note, where octave 4 is the central one ($a_4 = 440\text{Hz}$). All octaves start with note name c and ends in note name b, so note c_4 is the c note just below the a_4 note. Octaves range is 0 to 9.

Accidentals are represented by combinations of the signs plus (+), minus (-), equal (=) and the letter x (x). They represent, respectively, sharps, flats, naturals and the double sharp symbol. So, for example, ++ represents two sharps, x a double sharp, and =- represents a natural flat.

With all this, the pitch is just the combination of the previous signs. Examples:

```
+c4      a sharp C4 note (C of octave 4, the central C)  
  
--b3     a double flat B3 note (B of octave 3, the B immediately below C4)
```

Important

Accidentals are relative to the context. That is, you have to write only those accidentals that are not implied by the key signature or that are not implied by a previous accidental in the same measure.

Then, in the following example:

```
(key F) (n b4 q) (barline)
```

the quarter b4 note will be a flat b4, as it is in F major key.

8.2.8.2 Duration

```
[23 Duration ::= { number | letter } [ dots ]  
]
```

There are two possibilities two designate durations:

1. By numbers and dots: Durations are written as an apostrophe followed by the reciprocal value of the duration. For example, a quarter note is written as '4 (since it is a 1/4 note), while a 16th note is entered as '16 (since it is a 1/16 note). The number is followed by dots in case the note or rest is dotted. This method using numbers is language independent and can be always mixed with next method.
2. By a letter and dots: Each note duration is designated by a character from following table, and is followed by optional dots. This method is language dependent as a different set of characters can be defined for each supported language.

Designating notes by numbers is only allowed for notes shorter or equal than a whole. For notes longer than a whole you must use the letter notation.

Examples:

```
'8..      double dotted eighth note  
e..      also double dotted eighth note  
'128     128th note  
o        also 128th note
```

Letters to designate durations are as follows:

Duration	English US (English UK)
l	long (longa)
d	double whole (breve)
w	whole (semibreve)
h	half (minim)
q	quarter (crochet)

Duration	English US (English UK)
e	eighth, 8th (quaver)
s	sixteenth, 16th (semiquaver)
t	32th (demisemiquaver)
i	64th (hemidemisemiquaver)
o	128th (semihemidemisemiquaver)
f	256th (?)

For example: q. means a dotted quarter note and e.. means a double dotted eighth note

8.2.8.3 Abbreviated notation for notes and rests

The abbreviated syntax is formed as follows:

1. For pitch, the octave number can be omitted; if so, it is inherited from the preceding note.
2. Duration can be omitted; if so, it is inherited from the preceding note or rest.
3. Eliminate parenthesis and spaces preceding pitch and duration. Separate any other element using a comma (,).

Abbreviated and full notation can be mixed. Octave number and duration can only be omitted in abbreviated notation, not inside a full notation element.

Example 1:

Full notation:

```
(n +c4 q) (n e4 q) (r e) (n g4 e) (n =c5 q)
```

Abbreviated notation:

```
n+c4q ne re ng n=c5q
```

Example 2:

Full notation:

```
(n +c4 e g+ t+) (n e4 e) (n f4 e g- t-) (r s) (n g4 s) (n =c5 e)
(n +c4 '8 g+ t+) (n e4 '8) (n f4 '8 g- t-) (r '16) (n g4 '16) (n =c5 '8)
```

Abbreviated notation:

```
n+c4e,g+,t+ ne nf,g-,t- rs ng n=c5e
n+c4'8,g+,t+ ne nf,g-,t- r'16 ng n=c5'8
```

Mixed notation:

```
n+c4e,g+,t+ ne (n f4 e g- t-) rs ng (n =c5 e)
n+c4'8,g+,t+ ne (n f4 '8 g- t-) r'16 ng (n =c5 '8)
n+c4e,g+,t+ ne (n f4 '8 g- t-) r'16 ng (n =c5 e)
```

8.2.9 Chords

A chord is just the list of notes that form the chord:

```
[24 Chord ::= (chord Note*)  
]
```

Example:

```
(chord (n c4 q) (n e4 q) (n g4 q))      C major chord
(chord nc4q ne ng)                         abbreviated
```

8.2.10 Ties

A character **I** (lower case letter L) indicates the start of a tie that will end in the next coming note of the same voice, staff and pitch than the one on which this notation is included. No additional parameters are needed.

Example: A quarter C4 note tied to an eighth one:

```
(n c4 q 1) (n c4 e)
nc4q,1 nc
```

8.2.11 Beamed groups

When several consecutive notes have their stems beamed together this is coded by a notation **g** (group). The first parameter is a plus sign (+) or a minus sign (-) to mean, respectively, that it is the start or the end of a group. After this sign other parameters are possible. When the only parameters specified are the start/end sign, the **g** element admits an abbreviated syntax **g+** in the starting note and **g-** in the ending one. Example:

```
(n c4 q g+) (n d4 e) (n e4 e g-)  
nc4q, g+ nde ne, g-
```

A group of beamed notes, formed by a quarter C4 note, an eighth D4 note and an eighth E4 note.

8.2.12 Tuples

A tuple is a group of notes whose duration is modified; normally a graphic element (a tuplet bracket and/or a number) marks this group of notes. The notes in a tuple are usually beamed together but this is not always the case.

In LDP, to specify the notes that form a tuple, a notation **t** (tuple) must be included in the first and the last notes of the tuple.

```
[25 Tuplet :: (t { - | + ActualNotes [ NormalNotes ] [  
] = TupletOptions * ] } )
```

The first parameter is a plus sign (+) or a minus sign (-) to signal, respectively, that it is the start or the end of a tuple. After the plus sign there must be a number to indicate the tuple type.

The `ActualNotes` element describes how many notes are played in the time usually occupied by the number of `NormalNotes`. If there are different types of notes in the tuple (e.g. eighth and 16th notes) then `ActualNotes` and `NumNotes` must always refer to the shortest note type (e.g. 16th notes).

Default values for `NormalNotes`:

- triplet: three notes in time allotted for two: $(t + 3) \rightarrow (t + 3\ 2)$
- duplet: two notes in time allotted for three: $(t + 2) \rightarrow (t + 2\ 3)$
- 4-tuplet: four notes in time allotted for 6: $(t + 4) \rightarrow (t + 4\ 6)$
- 5-tuplet: five notes in time allotted for 6: $(t + 5) \rightarrow (t + 5\ 6)$

In all other cases `NormalNotes` must be specified.

Examples:

```
(n c4 e (t + 3)) (n d4 e) (n e4 e (t -))  
nc4e, (t + 3) nd ne, (t -)
```

This example is a triplet of eighth notes, that is, the three notes must be played in the time normally allotted for two eighth notes.

Additional parameters would be introduced in future to deal with renderization options.

As tuplets are very frequent, to simplify the writing, the tuplet element allows abbreviated syntax:

```
(t -)      can be abbreviated as t-
(t + x)    can be abbreviated as tx
(t + n m)  can be abbreviated as t+n,m
```

So, the previous example can be written as:

```
(n c4 e t3) (n d4 e) (n e4 e t-)
nc4e,t3 nd ne,t-
```

Element **t** does not implies beaming. Therefore elements **t** and **g** must be combined as needed. Examples:

```
(n c4 e g+) (n d4 s t3) (n c4 s) (n b3 s t- g-)
nc4e,g+ nds,t3 nc nb3,t-,g-
```

```
(n c4 s g+ t3) (n d4 s) (n e3 s t-) (n f4 s t3) (n g4 s) (n a4 s t- g-)
nc4s,g+,t3 nd ne3,t- nf4,t3 ng na4,t-,g-
```

In addition to timing information some options are possible:

```
[26 TupletOptions :: { TupletNumber | TupletBracket }
]
      =
[27 TupletNumber :: { numActual | numBoth | numNone }
]
      =
[28 TupletBracket :: { squaredBracket | curvedBracket |
]
      = noBracket }
```

The **TupletNumber** option is used to display either the number of actual notes (option **numActual**), the number of both actual and normal notes (option **numBoth**), or neither (option **numNone**). By default, it is **numActual**.

The `TupletBracket` option is used to display a bracket. It can be a squared bracket (option `squaredBracket`), a curved bracket (option `curvedBracket`), or nothing (option `noBracket`). If unspecified, the default value is `squaredBracket`.

The `TupletNumber` and `TupletBracket` options will be inherited by following tuplets until new options are set.

8.2.13 Selecting the staff

When an instrument has several staves (e.g. the upper and lower staves in a grand-staff) it is necessary a mechanism to specify in which staff a note, rest, clef, or other must be placed. For this purpose, an element **p** is used.

```
[29 StaffNumber ::= (p num)
```

```
]
```

It can be abbreviated by omitting the parenthesis and joining the **p** and the number. Example:

'(p 2)' is equivalent to 'p2'

Staffs are numbered from top to bottom, starting at staff 1. The staff number is inherited by all coming elements until a new **p** notation is found.

Example: a scale across two staves:

```
(instrument (staves 2)
  (musicData
    (clef G p1) (clef F p2) (key C)
    ng3w,p2 na nb nc4 nd,p1 ne nf ng
    (barline double)
  )
)
```

Clef G is placed in first staff (p1) and clef F in second staff (p2). The key is common to all staves of one instrument so no staff is specified. First note (G3) is placed on second staff and consecutive notes inherits this, until note D4 for which location on first staff is specified (p1). Remaining notes inherits this.

8.2.14 Polyphony

To represent several voices in staff there must be a way to move to the start of a measure to enter a second voice. The **goBack** and **goFwd** elements are intended to this, by allowing to move the internal time counter to the desired point so that more notes/rests, layered over existing ones, can be entered.

```
[30  goBack  ::: (goBack { start | Expression |  
]           = [duration] })  
  
[31  goFwd   ::: (goBack { end | Expression | [duration] }  
]           = )
```

were duration is the number of 256th notes to move forward or backwards (one quarter note equals 64 256th notes), and expression is an algebraic expression, formed by numbers, note duration letters and the plus and minus signs, to indicate the duration to move forward or backwards. Examples of expressions are:

```
3q+e -> Three quarter notes plus one eighth note;  
      it is equivalent to a duration of  $3 \times 64 + 32 = 224$ ,  
      that is, 224 notes of 256th type.  
q    -> A quarter note.  
q+e  -> A quarter note plus an eighth note, that is,  
      a quarter dotted note.
```

Constants **start** and **end** indicates to move to the start or the end of current measure, respectively.

Important

goBack and **goFwd** can only be used to move inside a measure.

Note that a **goFwd** element behaves like an invisible rest.

For example, following score is generated by next code:

```
(score  
  (vers 1.5) (language en iso-8859-1)  
  (instrument
```

```
(musicData
  (clef bass)
  (key e)
  (time 4 4)
  (n g3 q)
  (n a3 q)
  (n g3 q)
  (n e3 q)
  //go back to a3 note to add a second voice
  (goBack 128)
  (n e4 e. (stem up))
  (n d4 s)
  (barline)
)
)
)
```

8.2.15 Elements to control format

8.2.15.1 **newSystem**

Forces to start a new system after current measure.

```
[32 newSystem :: (newSystem)
]
```

Example:

```
(newSystem)
```

8.2.15.2 **stem**

Normally stem direction and size is automatically computed by the program, according to usual engraving rules. Nevertheless, the stem of a note can be fully specified by using the Stem element. It forces stem direction.

```
[33     Stem ::= (stem { up | down } )
]
```

Example:

```
(n e4 q (stem down))
(n e4 q stem:down)
```

8.2.15.3 The Location element

The location element is used to specify the coordinates of the point at which an object must be rendered.

```
[34 Location :: [ { (x coordinate) | (dx displacement) }
] = [
{ (y coordinate) | (dy displacement) }
]
```

Tag **x** refers to the horizontal direction and tag **y** to the vertical one. For example:

```
(x 40mm) (y 30mm)
```

means that the vertical location must be 30 millimetres from the top of the page and the horizontal location must be 40 millimetres from the left edge of the page.

The origin of the page (point x=0, y=0) is always at the top left corner of the page, more exactly the intersection point of left and top page margin lines.

The x coordinate increments always to right and y coordinate to bottom. Coordinates can always be negative.

Sometimes it is useful to specify a location not in absolute coordinates but relative to another object position. For example, to put a text above a specific note or a barline.

In these cases tags **dx** and **dy** should be used instead of **x** and **y**, respectively, where the 'd' means 'displacement'. The displacement is computed normally from parent object position, from its anchor point. This point is normally the lower left corner of its bounding rectangle, that is, the smallest rectangle that will completely enclose the object. In case of notes, the anchor point is the lower left corner of the rectangle enclosing the notehead.

8.2.15.3.1 Units

Two type of units:

relative: tenths.

one tenth of staff interline space. These units are relative, as a score can have staves of different sizes. So 'tenths' units can only be used to specify location points for objects referred to a staff.

absolute: millimetres, inches, centimetres.

When no relative units are possible (for example, to specify page size or page margins) absolute units must be used.

The unit name is always mandatory for absolute units:

- mm - millimetres
- cm - centimetres
- in - inches

if no unit name is specified it will be considered as tenths.

8.2.15.4 The **Font** element

```
[35     Font :: (font      [FontName]      [FontSize]  
]           = [FontStyle]))
```

```
[36 FontStyle :: { normal | bold | italic | italic-bold }  
]           =
```

Examples:

```
(font bold 14pt)  
(font "Arial" 12pt)  
(font 10pt)  
(font italic)
```

8.2.15.5 The **Title** element

```
[37     Title :: (title Alignment TextString [Font]  
]           = [Location]))
```

```
[38 Alignment :: { left | center | right }
```

```
]
      =
```

Examples:

```
(title center "Ave Maria (Ellen's Gesang III), D. 839" (font bold 14pt))
(title left "Words by Walter Scott" (font italic 12pt))
(title right "Franz Schubert")
(title left "Translation by F.Green")
```

center alignment forces to center the string in current line, without taking into account the space consumed by any possible existing left title. That is, **center** always means 'centered in the line'.

The font settings are stored as new default settings for coming **title** elements.

The title text string will be printed at current location; therefore, the location tag is not normally required. A new line is forced only if not enough space in current line for the text-string.

If a location is specified, the text will be positioned so that the lower edge of the text is located at the specified y location. And alignment takes place relative to the given x location. For example:

```
(title right "Franz Schubert" x:40mm y:30mm)
```

The y location given is 30mm. Since this is an absolute location the lower edge of the text will be at 30 millimetres from the top margin of the page. As the x location given is 40mm and, again, this is an absolute location, the current horizontal position will be 40 millimetres from the left edge of the page. Since right justification is indicated, the string "Franz Schubert" will be placed such that the right edge of the final "t" will be 40 millimetres from the left margin of the page. If "center" had been specified, the middle of the string "Franz Schubert" would be at 40 millimetres from the left margin of the page.

If a relative location is specified (dx or dy) they will refer always to current location. It, normally, will be the right bottom corner of rectangle bounding the previous title.

As titles are not positioned on staves, relative location units (tenths) are not possible as no staff is involved. You must always use absolute units, such as inches or millimetres.

8.2.15.6 The Text element

Elements of type **text** are used for placing arbitrary strings of text on the score. They must not be used to place neither the lyrics under the notes nor the score titles as for both of them there are specific elements.

```
[39  Text  :: (text  TextString  [Location]  [Font]
]           =  [Alignment] [Origin] [hasWidth] )

[40  Origin  :: { bottom | top }
]           =
```

Examples:

```
(text "Angelus dicit:" (dx -5) (dy 60))
(text "Allegro" dy:-30)
(text "A" dy:0 hasWidth)
(text "Right" dy:40 right (font normal 24))
(text "Example 1" dx:center dy:20 center)
```

Parameter **Origin** specifies the line used as reference in the **Location** parameter: either the top line of the text or its bottom line. By default, location parameter refers to the bottom line of the text.

Parameter **Alignment** refers to how the text will be positioned, relative to the specified location point and origin:

- **left** - text will be placed such that the right edge of the last character will be at the specified point. Depending on the specified or implied default origin, it will be the right top corner if origin is 'top', or the right bottom corner if origin is 'bottom'.
- **right** - text will be placed such that the left edge of the first character and the origin line will be at the specified point.
- **center** - text will be placed such that the middle of the string will be at the specified point.

Location should be normally relative to the parent's anchor point. But in case of **text** elements defined in a measure, they are relative to current x position and fifth line of staff.

The font settings are stored as new default settings for coming **text** elements

By default, **text** elements do not influence the note spacing. To take text width into account, parameter **hasWidth** must be included.

The location admits an special keyword, **dx:center**, which means 'center of current measure'. This moves the current x location to the center of the space occupied by current measure. Using this keyword in conjunction with alignment **center** simplifies placing texts centered in a measure.

8.3 LenMus eMusicBooks' file format

Copyright © 2006-2007 Cecilio Salmeron

Legal Notice

Permission is granted to copy, distribute, and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is available at <http://www.gnu.org/licenses/fdl.html>.

This document may be copied and distributed in any medium, either commercially or noncommercially, provided that the GNU Free Documentation License (FDL), the copyright notices, and the license notice saying the GNU FDL applies to the document are reproduced in all copies, and that you add no other conditions whatsoever to those of the GNU FDL.

8.3.1 eMusicBooks Intro

LenMus is a reader for books about music. These book are not part of the program and anyone can write them. This document is intended to describe the format of the eMusicBooks' files.

eMusicBooks are written in XML format. The reason to choose XML as the format for eMusicBooks is due to its advantages. Among them I would emphasize:

- The files are also readable by humans, and can be created and edited with any text editor.
- XML allows to separate content from presentation. The author can concentrate on the content and the presentation can be later changed or chosen to suit different needs.
- XLM allows for "single-sourcing", that is, it permits to generate multiple delivery formats, such as HTML, PDF, online help, etc., from the same source file.

As an eMusicBook is just a book about music, I have decide to take advantage of the widely used 'DocBook' XML format. The DocBook markup language is maintained by the OASIS consortium and is specifically suited for software documentation, I have added a few specific tags for LenMus purposes and have taken many of the tags from DocBook, so that people having good knowledge of this format will find easy to write eMusicBooks.

The eMusicBook format consists of tags, such as <book>, each with corresponding ending tags such as </book>, and textual content, such as "This is a line of text". Here you have a simple example:

```
<book id="first_book">
  <title>My first eMusicBook</title>

  <theme id="p1">
    <title>Page 1</title>
    <para>Hello world!</para>
    <para>This is my first music eMusicBook!</para>
  </theme>

  <theme id="p2">
    <title>Page 2</title>
    <para>Have a good day!</para>
  </theme>

</book>
```

The complete content of the document (a "book") has been structured into two "themes", each with a "title" and consisting of one or more "para(graphs)".

eMusicBooks are structured as conventional printed books, in chapter, sections, sub-sections, etc.. But the final structuring chunk is the **theme**, that is, the information that is going to be displayed in a page. This is the main difference with books in paper: paper has finite physical dimensions so text flows from one page to the next one. But the 'page' in a computer display can have an infinite dimension, so you must decide how you would like to chunk the document to present it to the reader. Each chunk is named a **theme**.

Therefore, a **book** is a collection of **themes**, organized in **chapters** and **sections**, and sections can also have sub-sections. For example:

```
<book id="first_book">
  <title>My first eMusicBook</title>

  <chapter id="ch1">
    <title>The first chapter</title>

    <section id="s1">
```

```
<title>Section one</title>
<theme id="p1">
    <title>Page 1</title>
    <para>Hello world!</para>
    <para>This is my first music eMusicBook!</para>
</theme>
</section>

<section id="s2">
<title>Section two</title>
<theme id="p2">
    <title>Page 2</title>
    <para>Have a good day!</para>
</theme>
</section>

</book>
```

Themes can have also internal structure, but sections inside a theme will be named **parts**.

Indenting and all whitespace is ignored except in some verbatim elements.

Notice how the tags indicate the structure and meaning of the content, but not its appearance. There is nothing to say "put this paragraph in bold-face type" or "center this on the page". This is by design. The same eMusicBook file can be processed into many different output formats, each with a completely different appearance and even a different arrangement of content elements. So authors must put the emphasis on content not on appearance.

Sections and parts can be nested to create sub-sections and sub-parts. For example:

```
<section>
    <section>
        <section>...</section>
        <section>
            <section>
                <section>...</section>
                <section>...</section>
            </section>
            <section>...</section>
        </section>
    </section>
    <section>...</section>
```

```
</section>
<section>...</section>
<section>...</section>
```

No theoretical limit on depth of nesting.

8.3.2 Tags to use: the eMusicBook DTD

To write an eMusicBook you need to know the set of tags that can be used. The formal way of defining this is by using a **Document Type Definition** or DTD. The DTD is a special file defining the legal XML tags and attributes that an author can use, and how they relate to each other. For example, a **book** element can contain a **title** element, any number of **para** elements for paragraphs, and any number of **chapter** elements. It also serves to automatically checking your document against the valid element names and rules, so to verify that your XML document is correct. Using the DTD and XML syntax, authors mark up their text content with tag names enclosed in angle brackets like `<chapter>`. The markup is similar to HTML, but with more tags and tighter rules.

8.3.3 Starting a document

All eMusicBook documents always start with the xml declaration file type, followed by a reference to the eMusicBook DTD. And then the eMusicBook content enclosed by a `<book>` tag:

```
<?xml version="1.0"?>
<!DOCTYPE book PUBLIC "-//LenMus//DTD eMusicBook 1.0//EN"
"http://www.lenmus.org/dtd/eMusicBook.dtd">

<book>
<bookinfo>
    <title>Music Reading exercises</title>
    <abstract>General exercises, customizable to suit your needs at any
        moment.</abstract>
    <copyright>
        <year>2002-2006</year>
        <holder>Cecilio Salmeron</holder>
    </copyright>
    <legalnotice>
        <para>Permission is granted to copy, distribute and/or modify this
document
            under the terms of the GNU Free Documentation License, Version 1.2 or
any
    
```

later version published by the Free Software Foundation; with no
Invariant

Sections, with no Front-Cover Texts, and with no Back-Cover Texts.

A copy of the license is included in the documentation received with
this program, and is also available at

<ulink

url="http://www.gnu.org/licenses/fdl.html">http://www.gnu.org/licenses/fdl.html</ulink>.

</para>

<para>

This document may be copied and distributed in any medium, either
commercially or noncommercially, provided that the GNU Free
Documentation

License (FDL), the copyright notices, and the license notice saying
the GNU FDL applies to the document are reproduced in all copies,
and that you add no other conditions whatsoever to those of the GNU FDL.

</para>

</legalnotice>

</bookinfo>

... here the book content

</book>

The <bookinfo> tag contains all meta-information about the book: A brief summary paragraph, to be included in the cover page (abstract) and copyright and legal information.

8.3.4 Creating a Table of Content (TOC)

When processing an eMusicBook, LenMus automatically compiles a table of content for the book. The TOC is formed by all chapters, sections and themes unless they have the attribute 'toc="no"'. Default value, if nothing specified, is toc="yes".

The TOC is structured following the structure of the included chapters, sections and themes. Notice that parts inside a theme are not included in the TOC.

When displaying the TOC of an eMusicBook in the navigation panel of the LenMus program it is possible to include images instead of (or beside of) titles. These images must be specified by

using tag <tocimage> when defining chapters, sections and themes. <tocimage> tag must go before the <title> tag. For example:

```
<theme id="ch205">
<tocimage>img205.png</tocimage>
<title>8th note tied to group of one 8th and two 16th notes</title>

<para>In this lesson .....
```

The image must not contain path information as all images for an eMusicBook must be stored in the same folder than the book XML files.

8.3.5 Indexes

When processing an eMusicBook, LenMus automatically compiles an index. To establish that a term is placed in the index, we have to mark it as being indexed. Doing so is simple, we just use the <indexterm> tag and give the term a name. Example:

```
This note is also known in British English as 'crochet'
<indexterm>crochet</indexterm>, and its usage...
```

8.3.6 Tags for content

Most tags for content are taken from DocBook (see appendix [Tags cross-reference](#)). The most used tag is the one to write paragraphs: <para>.

Example.

```
<para>
This is a very short paragraph
</para>
```

All usable tags are going to be described in the following sections.

8.3.7 Lists

An 'itemized' list is a list in which each entry is marked with a bullet or other dingbat. Example:

Source	Output
<pre><para>This is my shopping list:</para> <itemizedlist> <listitem>Apples</listitem></pre>	<pre>This is my shopping list:</pre>

Source	Output
<pre><listitem>Tomatos (but red)</listitem> <listitem>Carrots and beans</listitem> </itemizedlist></pre>	<ul style="list-style-type: none"> ● Apples ● Tomatos (but red) ● Carrots and beans

An 'ordered' list is a list in which each entry is marked with a sequentially incremented label, usually numerical. Example:

Source	Output
<pre><para>This is my shopping list:</para> <orderedlist> <listitem>Apples</listitem> <listitem>Tomatos (but red)</listitem> <listitem>Carrots and beans</listitem> </orderedlist></pre>	<p>This is my shopping list:</p> <ol style="list-style-type: none"> 1. Apples 2. Tomatos (but red) 3. Carrots and beans

8.3.8 Links

It is possible to include two types of links: external, to an URL, or internal, to other themes of the eMusicBook or to another eMusicBook. The tags to use are different:

External links (links to an URL):

```
<ulink url="http://www.lenmus.org/">visit our website</ulink>
```

```
<ulink url="#LenMusPage/SingleExercises.lmb"> a link to cover page of book 'SingleExercises'
```

```
<ulink url="#LenMusPage/Exercises for ear training"> a link to a theme whose title is 'Exercises for ear training'
```

Internal links are based on the eMusicBook title and on the id attributes added to the themes:

```
<link linkend="idname">Internal link</link> a link to theme 'idname' of this book
```

Example:

```
<section id="introduction">
```

```
...
```

```
</section>

<section id="another">
<para>
    For more info see the <link linkend="introduction">intro
    section</link>.
</para>
```

8.3.9 Including scores and interactive exercises

Scores and interactive exercises are included by using the `<score>` and `<exercise>` tags, respectively. These tags and their content is explained in detail in [chapter 2](#).

8.3.10 Splitting a book into several files

Usually, for large eMusicBooks, you may want to break it up into multiple files, as a book in a single file may not be the easiest thing to work with. Luckily this is easy to do. For example, if you create a two files, `part1.xml` and `part2.xml`, with the following content:

File '`part1.xml`':

```
<theme>
<title>On Foo's</title>
<para>
    Stuff about Foo's goes here.
</para>
</theme>
```

File '`part2.xml`':

```
<theme>
<title>On Bars's</title>
<para>
    Stuff about Bars's goes here.
</para>
</theme>
```

And creates a main file like:

```
<?xml version="1.0"?>
<!DOCTYPE book PUBLIC "-//LenMus//DTD eMusicBook 1.0//EN"
"http://www.lenmus.org/dtd/eMusicBook.dtd" [
```

```
<!ENTITY chapter1 SYSTEM "part1.xml">
<!ENTITY chapter2 SYSTEM "part2.xml">
]>

<book>
  <title>Splitted book</title>

  <theme>
    <title>Preface</title>
    <para>
      This is a book splitted into three files
    </para>
  </theme>

  &chapter1;
  &chapter2;

</book>
```

The use of the <!ENTITY> tags creates entities named 'chapter1' and 'chapter2'. Using them automatically includes their contents in to the main.xml file, so be careful to not put the <?xml?> preprocessor tags in the included files (part1.xml and part2.xml respectively).

8.3.11 Translation

Translating an eMusicBook requires the translation of the contents but a lot of care have to be put to not translate the tags or the attribute values. So, to minimize problems and to simplify translators life and translations coordination, the following schema will be used: all eBooks are written, initially, in English. The source XML eBook is then compiled using lenmus utility 'LangTool', that generates the LMB compiled eBook format used by LenMus. For translation, LangTool relies on PO files containing the translations for all strings in the eBook. LangTool is also used to generate a PO file from the source XML eBook.

The steps to translate an eMusicBook are, basically:

1. Process the eMusicBook with LangTool to generate the PO file for the required language.
2. Send the PO file to the translators.
3. When the translation is received, generate the compiled MO file.

4. Using LangTool, generate the translated eMusicBook in LenMus compiled format (.LMB).

These steps are performed by the LenMus Translations Coordinator.

8.3.12 Including scores and interactive exercises

Scores and interactive exercises are included by using the <score> and <exercise> tags, respectively. The content of these tags are a group of parameters for configuring the exercise or for defining the score and the controls. The possible tags to use to set up an score or an exercise are described in the following sections.

8.3.13 Including scores

Scores can be included by using the <score> tag. The <score> tag admits the following parameters:

<score_type>

Defines the type of score being included. Possible values: "short | pattern | full | XMLFile | LDPFile". If this param is not specified, default assumed value is: "full"

<music>

Mandatory. Defines the score content. Depending on the previous parameter it will be described using the LDP language or the MusicXML language.

<music_border>

Defines the type of score being included. Possible values: "0 | 1". If this param is not specified it is assumed: "0" (no border around score)

<control_play>

Include 'play' link. Default: do not include it. Value="play label|stop playing label". i.e.: "Play|Stop" Stop label is optional. Default labels: "Play|Stop"

<control_solfra>

Include 'solfa' link. Default: do not include it. Value="music read label|stop music reading label". i.e.: "Play|Stop". Stop label is optional. Default labels: "Read|Stop"

<control_measures>

Include 'play measure #' links, one per measure. Default: do not include them. Value="play label|stop label". i.e.: "Play|Stop". Stop label is optional. Default labels: "Measure %d|Stop %d"

Example:

```
<score classid="Score" width="600" height="100" border="0">
    <score_type>short</score_type>
    <music_border>0</music_border>
    <control_play>1</control_play>
    <music>
        (Metrica 2 4) (n a4 c g+) (n a4 c l g-) (n a4 c g+) (n a4 s) (n a4 s g-)
        (Barra Doble))
        (c (n a4 c) (n a4 n) (n a4 s g+) (n a4 s g-) (Barra Final))
    </music>
</score>
```

8.3.14 Including exercises

Interactive exercises can be included by using the <exercise> tag. All exercises available in LenMus program can be embedded in an eMusicBook. Attribute 'type' specifies the type of exercise:

- type="TheoIntervals"
- type="TheoScales"
- type="TheoKeySignatures"
- type="EarIntervals"
- type="EarCompareIntervals"
- type="IdfyChord"
- type="IdfyScales"

8.3.14.1 *Exercises on key signatures*

<max_accidentals>

Defines the maximum number of accidentals to use in the exercise. Possible values: A number from 0..7. If this param is not specified, default assumed value is: "5".

<problem_type>

Defines the type of exercise. Possible values: "DeduceKey | WriteKey | Both". If this param is not specified, default assumed value is: "Both".

<clef>

Possible values: "Sol | Fa4 | Fa3 | Do4 | Do3 | Do2 | Do1". If this param is not specified, default assumed value is: "Sol".

<mode>

Possible values: "Major | Minor | Both". If this param is not specified, default assumed value is: "Both".

Example

```
<exercise type="TheoKeySignatures" width="100%" height="300" border="0>
<max_accidentals>7</max_accidentals>
<problem_type>both</problem_type>
<clef>sol</clef>
<mode>both</mode>
</exercise>
```

8.3.14.2 Exercises on intervals (theory)

<accidentals>

Possible values: "none | simple | double". If this param is not specified, default assumed value is: "none".

<problem_type>

DPossible values: "DeduceInterval | BuildInterval | Both". If this param is not specified, default assumed value is: "Both".

<clef>

Possible values: "Sol | Fa4 | Fa3 | Do4 | Do3 | Do2 | Do1". If this param is not specified, default assumed value is: "Sol".

8.3.14.3 Exercises on chords (theory and aural training)

<keys>

Defines the allowed key signatures for the exercise. Possible values: Keyword "all" or a list of allowed key signatures, i.e.: "Do,Fas". If this param is not specified, default assumed value is: "all"

<chords>

Defines the allowed chords for the exercise. Possible values: Keyword "all" or a list of allowed chords:

m-minor, M-major, a-augmented, d-diminished, s-suspended

T-triad, dom-dominant, hd-half diminished

Examples:

triads: mT, MT, aT, dT, s4, s2

sevenths: m7, M7, a7, d7, mM7, aM7 dom7, hd7

sixths: m6, M6, a6

If this param is not specified, default assumed value is: ""mT,MT,aT,dT,m7,M7"

<mode>

Possible values: "'theory' | 'earTraining'" Keyword indicating type of exercise".

<play_mode>

Possible values: "'chord' | 'ascending' | 'descending'" allowed play modes. Default: chord".

<show_key>

Possible values: "'0' | '1'" Default: 0 (do not display key signature)".

<inversions>

Possible values: "'0 | 1' Default: 0 (do not allow inversions)".

<control_settings>

Possible values: "[key for storing the settings]" By coding this param it is forced the inclusion of the 'settings' link. Its value will be used as the key for saving the user settings.".

Example

```
<exercise type="IdfyChord" width="100%" height="300" border="0">
  <mode>earTraining</mode>
  <chords>mT,MT,aT,dT,m7,M7,dom7</chords>
  <keys>all</keys>
</exercise>
```

8.3.14.4 Exercises on scales (theory and aural training)

<keys>

Possible values: Keyword "all" or a list of allowed key signatures, i.e.: "Do,Fas" Default: all.

<scales>

Possible values: Keyword "all" or a list of allowed scales: m-minor, M-major, a-augmented, d-diminished, s-suspended T-triad, dom-dominant, hd-half diminished triads: mT, MT, aT, dT, s4, s2 sevenths: m7, M7, a7, d7, mM7, aM7 dom7, hd7 sixths: m6, M6, a6 Default: "mT,MT,aT,dT,m7,M7".

<mode>

Possible values: "'theory' | 'earTraining'" Keyword indicating type of exercise.

<play_mode>

Possible values: "'ascending' | 'descending' | 'both'" allowed play modes. Default: ascending.

<show_key>

Possible values: "'0 | 1' Default: 0 (do not display key signature).

<control_settings>

Possible values: "Value="[key for storing the settings]" By coding this param it is forced the inclusion of the 'settings' link. Its value will be used as the key for saving the user settings.

Example

```
<exercise type="IdfyScale" width="100%" height="300" border="0">
  <mode>earTraining</mode>
  <scales>mT,MT,aT,dT,m7,M7,dom7</scales>
  <keys>all</keys>
</exercise>
```

8.3.14.5 Exercises on music reading

optional tags for including controls:

<control_play>

Include 'play' link. Default: do not include it. Value="play label|stop playing label". i.e.: "Play|Stop" Stop label is optional. Default labels: "Play|Stop".

<control_solfa>

Include 'solfa' link. Default: do not include it. Value="music read label|stop music reading label". i.e.: "Play|Stop". Stop label is optional. Default labels: "Read|Stop".

<control_settings>

Value="[key for storing the settings]" This param forces to include the 'settings' link. The key will be used both as the key for saving the user settings and as a tag to select the Setting Dialog options to allow.

<control_go_back>

id of chapter, section or theme, i.e.: "ch205".

mandatory tags to set up the score composer:

<fragment>

one tag for each fragment to use.

<clef>

one tag for each allowed clef. It includes the pitch scope.

<time>

a list of allowed time signatures, i.e.: "68,98,128".

<key>

keyword "all" or a list of allowed key signatures, i.e.: "Do,Fas".

<max_interval>

a number indicating the maximum allowed interval for two consecutive notes. Default: 4

Example

```
<exercise type="TheoMusicReading" width="100%" height="300" border="0">
  <control_go_back>ch205</control_go_back>
  <control_play>1</control_play>
  <control_solfra>1</control_solfra>
  <fragment>24; (n * c g+) (n * c l g-) , (n * c g+) (n * s) (n * s g-)</fragment>
  <clef>Sol;a3;a5</clef>
  <clef>Fa4;a2;f4</clef>
  <time>24</time>
  <key>Do</key>
</lmobject>
```

8.3.15 Params cross-reference

In following table, columns labelled as 1, 2, .. 8 indicate the applicability of the described tag to each type of exercise, as listed here:

1. score
2. exercise type="TheoIntervals"
3. exercise type="TheoKeySignatures"
4. exercise type="EarIntervals"
5. exercise type="EarCompareIntervals"

6. exercise type="IdfyChord"
7. exercise type="IdfyScales"
8. exercise type="TheoMusicReading"

Table A.1. Supported params for exercises. Cross-reference table.

parameter	1	2	3	4	5	6	7	8	Obs.
accidentals	-	x	-	-	-	-	-	-	Defines if accidentals should be used. Possible values: "none simple double"
chords	-	-	-	-	-	x	-	-	Defines the allowed chords for the exercise. Possible values: Keyword "all" or a list of allowed chords.
clef	-	x	x	-	-	-	-	x	The clefs to use in the exercise. Possible values: "Sol Fa4 Fa3 Do4 Do3 Do2 Do1".
control_go_back	-	-	-	-	-	-	-	x	Include the 'go back to theory' link. Value: id of chapter, section or theme (i.e.: "ch205").
control_measures	x	-	-	-	-	-	-	-	Include 'play measure #' links, one per measure.
control_play	x	-	-	-	-	-	-	x	Include 'play' link.
control_settings	-	-	-	-	-	x	x	x	Include the 'settings' link to allow the user to configure the exercise.
control_solfa	x	-	-	-	-	-	-	x	Include the 'solfa' link
fragment	-	-	-	-	-	-	-	x	Information to set up the score composer. One tag for each fragment to use.
inversions	-	-	-	-	-	x	-	-	Defines if inversions are allowed. Possible values: "0 1".
key	-	-	-	-	-	-	-	x	Information to set up the score composer. Keyword "all" or a list of allowed key signatures, i.e.: "Do,Fas".
keys	-	-	-	-	-	x	x	-	Possible keys to use in the exercise. Possible values:

parameter	1	2	3	4	5	6	7	8	Obs.
									Keyword "all" or a list of allowed key signatures, i.e.: "Do,Fas" Default: all.
max_accidentals	-	-	-	-	-	-	-	-	Defines the maximum number of accidentals to use in the exercise. Possible values: A number from 0..7.
max_interval	-	-	x	x	x	-	-	x	Information to set up the score composer. A number indicating the maximum allowed interval for two consecutive notes. Default: 4
mode	-	-	x	-	-	x	x	-	The meaning of this parameter depends on the exercise. Usually it is to select the exercise type (theory or aural training) or the tonality (Major, Minor)
music	x	-	-	-	-	-	-	-	The score to show. Either LDP language or MusicXML.
music_border	x	-	-	-	-	-	-	-	To set up a border around the control (the score plus the optional links).
problem_type	-	x	x	-	-	-	-	-	Defines the type of exercise: direct (i.e. "DeduceKey"), inverse (i.e. "WriteKey") or both.
play_mode	-	-	-	-	-	x	x	-	Defines how to play the intervals or chords: melodic, harmonic, ascending, descending, etc.
scales	-	-	-	-	-	-	x	-	Defines the allowed scales for the exercise. Possible values: Keyword "all" or a list of allowed scales
show_key	-	-	-	-	-	x	x	-	Defines if the key signature must be displayed or, instead, accidentals should be placed on individual notes. Possible values: "0 1". Default: 0 (do not display key signature)
time	-	-	-	-	-	-	-	x	Information to set up the score composer. A list of allowed time signatures, i.e.: "68,98,128".

parameter	1	2	3	4	5	6	7	8	Obs.
score_type	x	-	-	-	-	-	-	-	Defines the type of score being included. Possible values: "short pattern full XMLFile LDPFile".

8.3.16 Tags cross-reference

The present appendix list alphabetically all tags that are defined and supported in current version of the LangTool processor program, used to compile the eMusicBooks:

- abstract - Used inside <bookinfo> for the text to include in the cover page after the book title
- book - An eMusicBook. It is the main tag.
- bookinfo - Meta-information for an eMusicBook
- chapter - A chapter, as of a book
- content - There is a restriction of XML the syntax: an XML file must content just one main tag, and all content must be children of this main tag. In eMusicBooks this main tag is the <book> tag. When a book is splitted in two or more files, for your convenience, I have defined tag <content> as a wrapper for all the content of an XML splitted file. So tag <content> is a way to by-pass the XML restriction before mentioned.
- copyright - Copyright information about a document. It is used inside a <bookinfo> tag for the text to include in the cover page after the abstract
- emphasis - Emphasized text
- exercise - An interactive music exercise
- holder - The name of the individual or organization that holds a copyright. It is used inside <copyright> tag.
- itemizedlist - A list in which each entry is marked with a bullet or other dingbat
- leaflet - An eMusicBook leaflet. It is the main tag, used instead of <book> for single page books.
- leafletcontent - Used instead of <theme> tag for leaflets. A wrapper for the leaflet content.
- legalnotice - A statement of legal obligations or requirements
- link - A hypertext link
- listitem - A wrapper for the elements of a list item

- orderedlist - A list in which each entry is marked with a sequentially incremented label
- para - A paragraph
- part - A section inside a theme
- score - An interactive music score
- section - A section inside a chapter. Can be used recursively, that is, a section can also have sections.
- simplelist - An undecorated list of single words or short phrases
- theme -
- title - The text of the title of a book, chapter, section, theme or part of a document.
- titleabbrev - Holds an abbreviated version of a <title>. It is used for the text in running headers or footers, when the proper title is too long to be used conveniently. It must be defined before the <title> tag.
- tocimage - A reference to an image file (.png, jpg, etc.) to be used in the TOC
- ulink - A link that addresses its target by means of a URL (Uniform Resource Locator)
- year - The year of publication of a document; used as child of <copyright> tag.

8.4 La licencia libre de LenMus

LenMus es software libre: puede distribuirse y modificarse según los términos de la licencia GNU General Public License publicados por la Free Software Foundation, en la versión 3 o posterior (a elegir por el usuario).

LenMus es copyright © 2002-2009 de Cecilio Salmerón.

Para versiones antiguas, las licencias aplicables son las siguientes:

- lenmus 1.x - freeware. Sin código fuente.
- lenmus 2.x & 3.x - GPL v2+
- lenmus 4.x - GPL v3+

El texto íntegro y literal que recoge los términos de la licencia GPL versión 3 de GNU es el que sigue. Se recoge la versión oficial en inglés. No hay traducción oficial al castellano aunque es fácil encontrar traducciones no-oficiales en <http://www.gnu.org/licenses/translations.es.html>.

GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc.
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

Preamble

The GNU General Public License is a free, copyleft license for
software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program--to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

Analizador tonal en software libre

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

- a) The work must carry prominent notices stating that you modified it, and giving a relevant date.
- b) The work must carry prominent notices stating that it is released under this License and any conditions added under section

Analizador tonal en software libre

7. This requirement modifies the requirement in section 4 to "keep intact all notices".

c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and CorrespondingSource of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

Analizador tonal en software libre

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work. You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

- a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
- b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate LegalNotices displayed by works containing it; or
- c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or
- d) Limiting the use for publicity purposes of names of licensors or authors of the material; or
- e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
- f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose onthose licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction doesnot survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions;the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally

terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version,

but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to

whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

8.5 Teoría de armonía a cuatro voces de Marcelo Gálvez

Este material es una recopilación de textos sobre teoría de la armonía a cuatro voces realizado por Marcelo Gálvez para LenMus.

8.5.1 Consonancia y disonancia

De manera preliminar, se recomienda practicar los ejercicios de reconocimiento e identificación de intervalos y acordes.

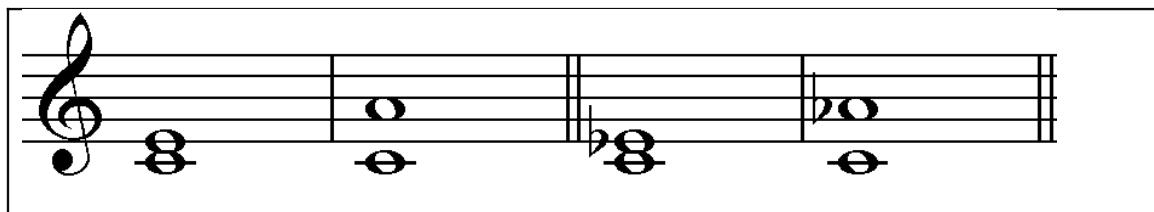
Existe una creencia popular que afirma que la consonancia ‘suena bien’ mientras que la disonancia ‘suena mal’. Ésta es una visión en exceso simplista y reduccionista, además de altamente subjetiva. Una música que huya de la disonancia es frecuentemente una música floja y falta de interés puesto que el elemento disonante proporciona vitalidad y necesidad de movimiento. Aunque también sea subjetiva, esta clasificación es la más aceptada:

Un intervalo consonante suena estable y completo, por lo que produce sensación de reposo sonoro al oído. Son consonancias los intervalos de 3^a y 6^a tanto Mayor como menor y los intervalos Justos (4^a, 5^a y 8^a).

Por el contrario, un intervalo disonante suena inestable, crea tensión sonora al oído por lo que sugiere una continuación (resolución) en uno consonante. Son intervalos disonantes la 2^a y la 7^a tanto Mayor como menor y todos los intervalos aumentados y disminuidos. El tratamiento de la disonancia es parte del estudio de la armonía y a lo largo de la historia tanto los teóricos como los compositores lo han abordado de distintas maneras.

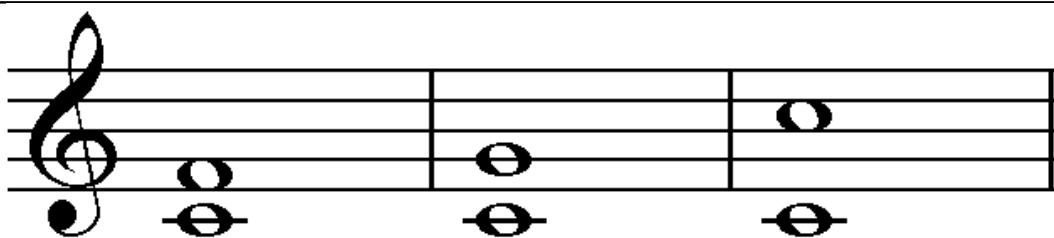
Dentro de los intervalos consonantes encontramos dos tipos:

- Consonancia imperfecta, variable o modal: Se considera tradicionalmente así a los intervalos de 3^a y 6^a tanto Mayores como menores. Precisamente es esta variabilidad entre Mayor y menor la que la hace imperfecta o variable, mientras que es modal porque los grados III y VI de la escala son los que definen con mayor fuerza la modalidad Mayor o menor.



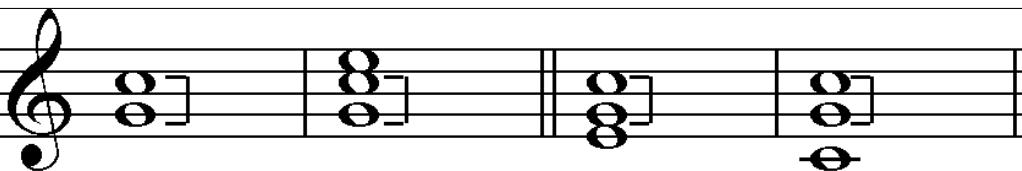
Ejemplo de consonancias imperfectas: 3^a y 6^a Mayores; 3^a y 6^a menores.

- Consonancia perfecta, invariable o tonal: Se considera tradicionalmente así a los intervalos de 4^a, 5^a y 8^a Justas. En el hecho de que sólo admite la posibilidad de intervalo justo radica su carácter perfecto e invariable. Su carácter tonal viene dado por el hecho de que los grados I, IV y V de la escala son los grados tonales, es decir, los que definen la tonalidad.



Ejemplo de consonancias perfectas: 4^a, 5^a y 8^a Justas.

No obstante, cabe una excepción: El intervalo de 4^a Justa se considera disonante cuando está sola, mientras que se considera consonante cuando por debajo de él hay una 3^a mayor o menor o una 5^a Justa. Por ello, en algunos manuales podemos encontrar que se refieren a él como un intervalo ambiguo o anfibio.

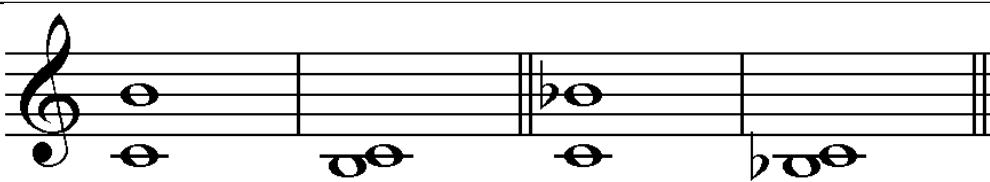


Ejemplos de 4^a Justa disonante y consonante. Los dos primeros casos muestran una 4^a Justa empleada como disonancia, mientras que los dos casos siguientes muestran la misma 4^a Justa empleada como consonancia.

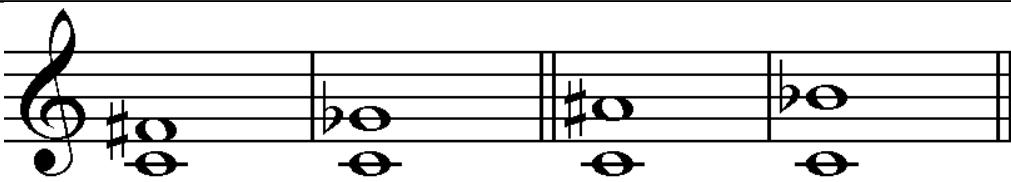
Cualquiera de las tres denominaciones de cada tipo es perfectamente definitoria e independiente de las demás, por lo que no es necesario aludir a las tres.

Dentro de los intervalos disonantes encontramos igualmente dos tipos:

- Disonancia absoluta: Se considera así a los intervalos de 2^a y 7^a tanto mayores como menores, y los intervalos aumentados o disminuidos en los que aun enarmonizando se sigue manteniendo el carácter de disonancia. Por tanto, una disonancia para que sea absoluta debe ‘resistir’ a la inversión y a la enarmonización.

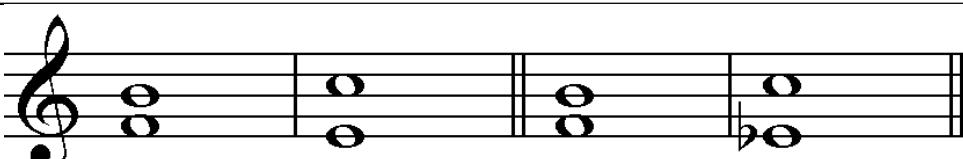


Ejemplos de disonancias absolutas. En primer término 7^a y 2^a Mayores. En segundo término 7^a y 2^a menores.

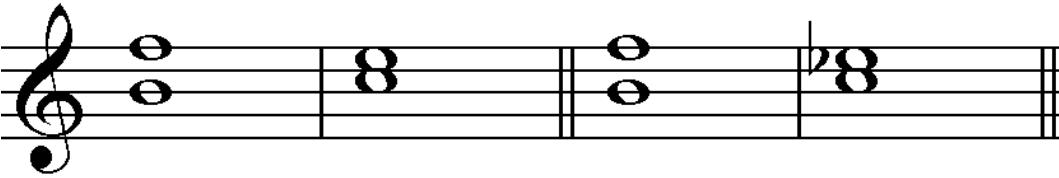


Dos ejemplos de disonancias absolutas con intervalos aumentados y disminuidos, en los que tanto la forma original como la enarmonizada mantienen su carácter disonante.

Un caso especialmente interesante de disonancia es la cuarta aumentada que se forma de manera natural en la escala entre los grados IV y VII de la escala, que también recibe el nombre de ‘cuarta tritono’. Este intervalo fue llamado durante la Edad Media ‘Diabolus in Musica’ y prohibido por su ‘carácter siniestro’ y dificultad de entonación. De manera general, este intervalo se resuelve abriendolo en una sexta (tanto Mayor como menor); y su inversión (5^a disminuida) cerrándolo en una tercera (tanto Mayor como menor).

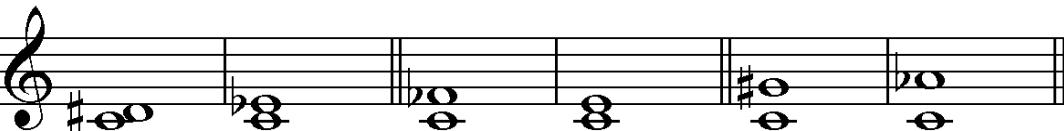


Ejemplos de resolución de la cuarta tritono sobre una sexta tanto Mayor como menor.



Ejemplos de resolución de la cuarta tritono invertida (5^{a} disminuida) sobre una tercera tanto Mayor como menor.

- Disonancia relativa o condicional: Son aquellos intervalos aumentados o disminuidos en los que enarmonizando alguno de sus sonidos, el intervalo se convierte en consonante. Es, por tanto, una disonancia más aparente que real condicionada al contexto tonal en el que se inserta.



Tres ejemplos de disonancia relativa: Junto a la disonancia relativa aparece su correspondiente enarmonización por lo que queda como consonancia.

8.5.2 Enarmonía

Enarmonía es el nombre que se aplica a la relación entre sonidos que, a pesar de poseer distintos nombres, son iguales en entonación. La acción de convertir un sonido cualquiera en su correspondiente enarmónico recibe el nombre de enarmonizar.

Todos los sonidos tienen dos correspondientes enarmónicos, excepto el *Sol #* que sólo tiene uno. Este fenómeno puede apreciarse claramente sobre el teclado de un piano, donde se puede observar la correspondencia de una misma tecla para distintas notas. No obstante, en algunos instrumentos de cuerda, por cuestiones de afinación, no existe este fenómeno, que tampoco se daba en la Música Antigua.

Para realizar esta relación es necesario tener en cuenta la distancia entre sonidos contiguos dentro de la escala, que es, de un tono, excepto entre los sonidos *Mi* y *Fa*. A la vez que debemos tener en cuenta el efecto de las alteraciones: sostenido, bemol, doble sostenido y doble bemol.

Ésta es una tabla de los sonidos enarmónicos.

Cuando hablamos de intervalos, podemos enarmonizar uno solo de sus sonidos o ambos sonidos. En el primer caso hablamos de enarmonización parcial, mientras que en el segundo hablamos de enarmonización total o completa.

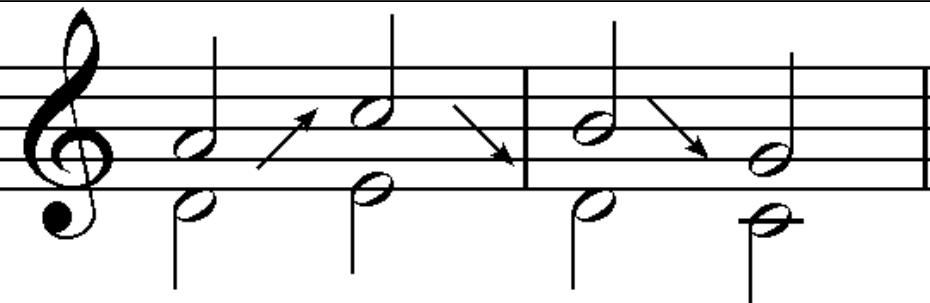
Ejemplos de enarmonización: En primer término, junto al intervalo original aparece

una enarmonización parcial (sólo enarmonizamos un sonido); mientras que en segundo término aparece el mismo intervalo original junto a una enarmonización total del mismo (se enarmonizan los dos sonidos)

8.5.3 Tipos de movimiento relativo de voces

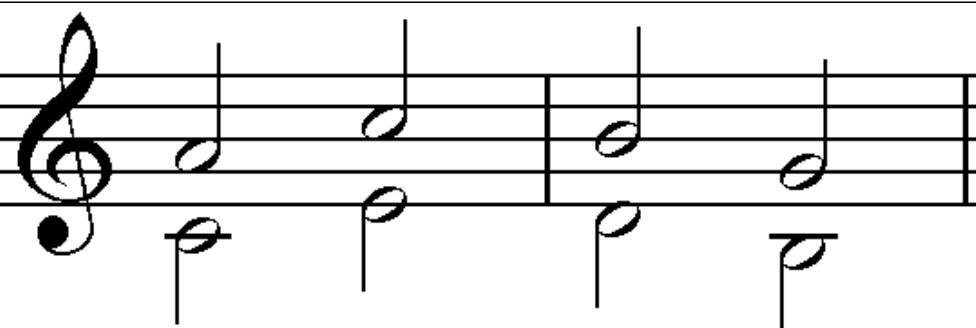
Dos voces pueden moverse de tres maneras básicas, en relación la una con la otra:

- **Movimiento directo:** Las dos voces se mueven en la misma dirección, es decir, ascendente o descendente. En la armonía académica hay que tener especial cuidado con él.



Ejemplo de movimiento directo: las dos voces se mueven primero en dirección ascendente y después en dos movimientos descendentes.

El movimiento directo continuado y a una distancia fija se denomina **movimiento paralelo**, que en determinados casos, dentro de la armonía académica, puede ser contraindicado. Esto se debe a que en la práctica armónica (no tanto en la composición) se busca la individualidad de las voces, que con este movimiento se perdería.



Ejemplo de movimiento paralelo: las dos voces se mueven primero en dirección

ascendente y después en dos movimientos descendentes, a un intervalo fijo de sexta entre la voz superior y la inferior.

- **Movimiento Contrario:** Las dos voces se mueven en dirección opuesta, es decir, mientras que una asciende, la otra desciende o viceversa.

Ejemplo de movimiento contrario de dos: cuando una asciende, la otra desciende y viceversa.

Tanto en uno como en otro, no importa si el intervalo de movimiento de cada una de las voces es el mismo o no.

- **Movimiento oblicuo:** Una voz permanece inmóvil mientras la otra se mueve, sin importar si lo hace ascendentemente o descendientemente, o intervalo del movimiento.

Ejemplo de movimiento oblicuo: la voz inferior se mantiene quieta mientras la voz superior se mueve, después la situación cambia, ahora es la voz superior la que se mantiene inmóvil mientras que la inferior se mueve.

Como es lógico, estos tipos de movimiento pueden darse entre cualquier par de voces, es decir, no tienen porqué ser dos voces contiguas; e igualmente, cuando se tienen más de dos voces, pueden darse simultáneamente varios tipos de movimiento. Obvia decir que ningún movimiento es mejor que otro, un buen trabajo incluye una adecuada combinación de todos ellos.



Ejemplo de combinación de movimientos: las cuatro voces producen simultáneamente distintos movimientos entre ellas: la voz superior con la segunda voz produce un movimiento contrario y dos directos; la voz superior con la tercera voz produce dos movimientos contrarios y uno directo; la segunda voz con respecto a la tercera forman un movimiento directo y dos contrarios; mientras que cada una de ellas producen movimientos oblicuos con la voz inferior.

8.5.4 Armonía

8.5.4.1 Realización a cuatro partes

Cuando realizamos un ejercicio de armonía, lo más habitual es que lo escribamos a cuatro partes, es decir, a cuatro voces. Cuando decimos ‘voz’ no es necesariamente que sea cantado, sino que cada una de ellas es una entidad independiente, melódica ya sea para instrumentos o voces humanas.

Estas cuatro voces, de agudo a grave, son: soprano, alto, tenor y bajo. Suelen repartirse en un sistema de dos pentagramas (como el de un piano) y en cada uno escribiremos dos voces: soprano y alto en el superior, y tenor y bajo en el inferior; a su vez la voz superior se escribirá con la plica hacia arriba y la voz inferior con la plica hacia abajo. Igualmente, cada voz tiene unos límites aproximados que se deben respetar en la medida de lo posible. No hay perjuicios en exceder estos límites, pero debe ser de manera puntual y con moderación para obtener una buena sonoridad.

The image shows a musical score example within a rectangular frame. It consists of two staves. The top staff has a treble clef and a bass clef, and it contains three notes: a soprano note (highest), an alto note (middle), and a tenor note (lowest). The bottom staff has a bass clef and contains one note: a bajo note (lowest). The voices are labeled as follows: 'Soprano' above the first note, 'Alto' above the second note, 'Tenor' below the third note, and 'Bajo' below the fourth note. The notes are connected by vertical stems.

Ejemplo de escritura armónica a cuatro voces. Cada voz incluye los límites de extensión aproximada.

No está permitido que entre un par de voces contiguas (soprano-alto, alto-tenor) se exceda la 8^a, salvo entre bajo y tenor donde la distancia puede ser de hasta dos octavas.

<i>Distancia permitida entre tenor y bajo</i> 	<i>Distancia no permitida entre alto y tenor</i>
---	--

Distancia permitida entre tenor y bajo

A musical staff with four lines. The soprano (S) voice is on the top line, alto (A) is on the second line, tenor (T) is on the third line, and basso (B) is on the bottom line. Brackets group the tenor and basso voices together, indicating they are close together.

De esta manera cuando las tres voces están todo lo juntas que es posible (las tres voces dentro de una octava) hablamos de disposición cerrada, por el contrario una separación mayor de una octava entre soprano y tenor recibe el nombre de disposición abierta. Tanto una como otra tienen ventajas e inconvenientes, la elección dependerá del movimiento melódico de las voces y del ámbito en el que se muevan. La posición abierta o cerrada no afecta a la colocación del bajo con respecto a las voces superiores.

Disposición cerrada

A musical staff with four lines. The soprano (S) voice is on the top line, alto (A) is on the second line, tenor (T) is on the third line, and basso (B) is on the bottom line. Brackets group the soprano and alto voices together, indicating they are close together.

Disposición abierta

A musical staff with four lines. The soprano (S) voice is on the top line, alto (A) is on the second line, tenor (T) is on the third line, and basso (B) is on the bottom line. Brackets group the soprano and alto voices together, indicating they are further apart than in the closed disposition.

Los acordes tríadas tienen 3 factores (notas) y tenemos cuatro voces, por lo que lo más habitual es duplicar la fundamental del acorde, aunque también es posible duplicar la tercera. Duplicar la quinta del acorde es posible pero suele dar lugar a fallos, por tanto, de emplearse, debe tenerse especial atención. En el acorde sobre el VII lo habitual es duplicar la tercera, ya que el VII al ser sensible y sentirse atraído por la tónica, puede crear fallos.

8.5.4.2 Principios de conducción de voces en el modo mayor:

La armonía tonal tiene unas reglas que le permiten mantener la coherencia, el control y el orden, por ello, no son posibles cualquier enlace de acordes. Según Walter Piston, estos son los enlaces de acordes en fundamental más habituales en el modo mayor (extracto):

Al I le siguen el IV o el V y con menos frecuencia el VI.

Al II le siguen el V o el IV y con menos frecuencia el VI.

Al III le siguen el VI y con menos frecuencia el IV.

Al IV le siguen el V, el I o el II.

Al V le siguen el I, el IV o el VI.

Al VI le siguen el I o el V.

Al VII le sigue el I y con menos frecuencia el VI o el V.

Con estos enlaces, tenemos tres movimientos de acordes en fundamental:

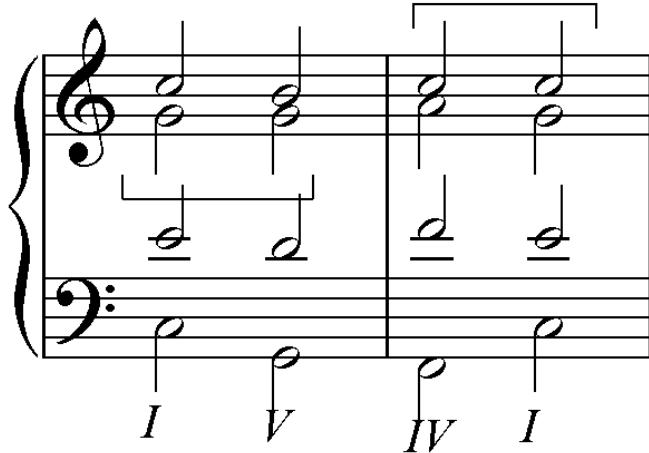
- 1) Acordes cuyas fundamentales están separadas a distancia de cuarta o quinta: Es la relación armónica de mayor fuerza e importancia en la música tonal. Tienen una nota en común.
- 2) Acordes cuyas fundamentales están separadas a distancia de tercera (o sexta): Son las progresiones más débiles, teniendo en común dos notas, por lo que tan sólo hay una nota de diferencia. Debido a este parecido, este tipo de relaciones es muy empleada en muchos tipos de música como en el jazz y música reciente, donde se recurre a la sustitución de acordes.
- 3) Acordes cuyas fundamentales están separadas a distancia de segunda (o séptima): No hay ningún factor en común, por lo que los dos acordes son totalmente distintos.

Normalmente, los saltos entre fundamentales mayores de la 5^a no suelen emplearse, a excepción de la 8^a que confiere una nueva disposición.

8.5.4.3 Primeras reglas prácticas

A fin de mantener el valor musical de cada voz independiente, así como de la coherencia y sonoridad del conjunto, existen unas reglas prácticas que deben seguirse más como consejos que como leyes rígidas puesto que están basadas en la experiencia.

1^a regla práctica: Nota común y mínimo movimiento. Cuando dos tríadas consecutivas tengan alguna nota en común, esa nota común se repetirá, mientras que las notas no comunes se moverán lo menos posible, a fin de no crear faltas.



2^a regla práctica: Movimiento contrario al bajo. Cuando movamos las voces intentaremos mover las voces superiores en dirección opuesta al bajo hasta la nota más cercana. Esto es especialmente recomendable cuando pasemos de un acorde a otro en el que no hay ningún factor común.

3^a regla práctica: Sensible a tónica. La sensible se siente atraída por la tónica puesto que se encuentra a medio tono, por ello, lo normal será conducir la sensible a la tónica, evitando duplicarla especialmente cuando forma parte del V o del VII. Hay una excepción que consiste en que la sensible puede no ir a la tónica cuando se encuentre en voz interior (alto o tenor) y otra voz llegue a la tónica por grado conjunto.



4^a regla práctica: Acorde repetido, cambio de disposición. Cuando encontrremos que la fundamental se repite, y, por tanto, se repita el acorde, realizaremos un cambio de disposición a fin

de conseguir variedad e interés. Este cambio de disposición (movimiento de las voces dentro del mismo acorde) no genera faltas.

5^a regla práctica: Elisión de la quinta del acorde. La quinta del acorde es la única nota que realmente podemos elidir: Si elidimos la fundamental, estaremos cambiando de acorde y si elidimos la tercera, tendremos un acorde vacío (no sabemos si es mayor o menor) que en determinados estilos puede tener interés pero no es válido a efectos armónicos. Este recurso es interesante a tener en cuenta puesto que habrá ocasiones en las que no podamos solventar el enlace si no es recurriendo a este recurso. En este caso podemos:

- Triplicar fundamental dejando una única tercera.
- Duplicar fundamental y duplicar tercera.

Generalmente no se suele duplicar la quinta, pero no hay nada que lo impida si nos vemos en la necesidad de hacerlo. De las opciones anteriores, suele aceptarse como más válida la primera.

8.5.4.4 Movimientos prohibidos

Los compositores han intentado mantener la independencia e interés de cada voz evitando las quintas y las octavas directas. Deben evitarse entre cualquier par de voces pero sobre todo entre las voces polares (soprano y bajo) o entre cualquier voz y el bajo. En términos generales podemos clasificar los movimientos prohibidos de más a menos ‘graves’ de la siguiente manera:

- Soprano y bajo.
- Alto / tenor y bajo.
- Alto / tenor y soprano.
- Alto y tenor.

Excepciones:

- Cuando hagamos un cambio de disposición (mover voces superiores dentro de un único acorde) no produce octavas ni quintas
- Cuando una quinta u octava directa es atacada por grado conjunto por una única voz, siempre que sea una voz distinta al bajo, entonces estará permitida. No es válida la excepción cuando se mueva por grado conjunto en las dos voces o cuando la voz que se mueve por grado conjunto es el bajo.

8.6 Curso de teoría de la armonía de Michel Baron

Extracto de <http://membres.multimania.fr/mbaron/esp/e-h-regles.htm>

8.6.1.1.1.1 Reglas armónicas

Las reglas armónicas hacen referencia a los movimientos que se producen (y se modifican en cada acorde) entre dos voces. No importa qué par de voces entre las cuatro.

8.6.1.1.1.2 Cruzamientos y unísonos

Al comienzo de los estudios no hay ninguna razón para efectuar cruzamientos entre las voces: se las utiliza según su orden normal de superposición. Puede utilizarse el unísono, si es inevitable, o si hay una buena razón para hacerlo, debido al contexto.

8.6.1.1.1.3 Movimientos entre las voces

Hay numerosas maneras de espaciar entre ellas las 4 voces. He aquí unas cuantas, comentadas en los números correspondientes:

The image shows a musical score with four voices: bass (bottom), tenor, alto, and soprano (top). The bass voice has a continuous eighth-note pattern. The tenor voice has a pattern of quarter notes and eighth notes. The alto voice has a pattern of eighth notes and sixteenth notes. The soprano voice has a pattern of eighth notes and sixteenth notes. Below the staff, five numbers (1, 2, 3, 4, 5) are connected by dashed lines to specific notes, indicating movement patterns between voices. Number 1 points to a note in the bass line. Number 2 points to a note in the tenor line. Number 3 points to a note in the alto line. Number 4 points to a note in the soprano line. Number 5 points to a note in the bass line.

1. Intervalos casi iguales entre las diferentes partes. A esta disposición se la conoce como posición de cuarteto. Es una posición ideal, que suena muy bien.
2. Se puede, por el contrario, aislar el bajo y agrupar las tres partes superiores. A esta disposición se la llama a veces la posición de piano (el bajo para la mano izquierda, el resto para la mano derecha).
3. No se debe de sobrepasar la octava entre ninguna de las tres partes superiores y su vecina (pero esto está permitido entre el bajo y el tenor).
4. Una disposición particular: si se alcanza la octava entre **las dos partes intermedias**, las dos voces inferiores deben formar un intervalo **que no sobrepase la tercera**, si no el acorde sonará hueco. Hay que retener esta disposición particular: los estudiantes la temen, aun

cuando ésta no suena tan mal (recordar los cursos de dictado a cuatro partes y todos los errores diversos que esta disposición, rica en armonías, generaba).

5. Una advertencia necesaria: se pueden distanciar hasta la octava las dos voces superiores. Más tarde se verá que incluso pueden, excepcionalmente, sobrepasar la octava durante poco tiempo si el equilibrio de los movimientos melódicos lo justifica.

Observad y adoptad cuidadosamente las convenciones de escritura que se refieren al sentido de las plicas y a la colocación de las ligaduras.

8.6.1.1.4 Movimientos armónicos

Estos son los movimientos que hacen cada par de voces. Hay cuatro clases:

Movimiento paralelo: las dos voces suben o bajan juntas, conservando el mismo intervalo. Frecuentemente lo hacen en tercera o en sextas (o sus redoblamientos).

El movimiento paralelo está absolutamente prohibido en dos casos:

Octavas consecutivas (diferentes) por movimiento paralelo.

Quintas consecutivas (diferentes) por movimiento paralelo.

Estos movimientos han sido juzgados demasiado vulgares y demasiado duros, desde el Renacimiento hasta los post-románticos, quienes no los emplean (a 4 voces) más que muy excepcionalmente.

Movimiento contrario: una parte sube, la otra baja, o a la inversa. Es un movimiento extremadamente frecuente, bien equilibrado. No presenta inconvenientes. El ejemplo siguiente muestra dos pares de voces paralelas que, por pares, evolucionan en movimiento contrario:

Beethoven

Las únicas faltas que este movimiento puede generar, casi por mala suerte, son:

Octavas consecutivas por movimiento contrario.

Quintas consecutivas por movimiento contrario.

Se aplican las mismas razones que se vieron precedentemente.

Nota: se llaman también, a manera de resumen "**octavas consecutivas, quintas consecutivas**", pues la prohibición es aplicable tanto para las quintas y las octavas paralelas como para las quintas y las octavas consecutivas por movimiento contrario.

Movimiento oblicuo: una voz queda quieta mientras la otra se aleja o se acerca. Nada a señalar, salvo en el momento de un unísono: no es elegante llegar a él por parte de la voz que se mueve. Por el contrario, es interesante salir del unísono por movimiento oblicuo.

Beethoven

Movimiento directo: las dos partes ascienden o descenden juntas, pero acercándose o alejándose un poco. Imaginaos un movimiento paralelo que no fuese completamente paralelo...

El movimiento paralelo hace resaltar las quintas y las octavas que conduce. Cuando se escribe un movimiento directo que finaliza en una quinta o en una octava (o su redoblamiento), es necesario satisfacer las condiciones siguientes, destinadas simplemente a *limitar el efecto* del movimiento directo.

Quinta u octava directa entre voces extremas:

- La voz más alta (en principio la soprano) debe proceder por movimiento conjunto.

Quinta u octava directa en cualquier otro par de voces (se suele decir: entre cualesquiera partes):

- Una de las voces de proceder por movimientos conjuntos (idealmente, la de arriba).
- A falta de una de las condiciones aquí arriba mencionadas, la nota de la octava o una de las notas de la quinta debe ser oída en el acorde anterior (no necesariamente a la misma altura).

Unísono directo:

- **Siempre prohibido**, también en el contrapunto y en la fuga...

Nota: las reglas precedentes son una simplificación reciente de antiguas prácticas pedagógicas particularmente intimidatorias y desmoralizantes. Para más detalles, los más curiosos pueden solicitar reglas más detalladas. Ver mi **Compendio práctico de armonía**, página 15; el **Tratado de armonía** de Dubois, páginas 14 a 16; el **Tratado de armonía**, I, página 15 de Koechlin; el **Curso de armonía analítica** de Dupré, página 17; el **Compendio de armonía tonal** de Bitsch, páginas 19 y 20; o el **Tratado de armonía** de Yvonne Desportes, página 7.

Sobre este tema de las quintas y octavas directas permitidas o prohibidas basta con consultar estas diversas obras, todas redactadas por grandes profesores y compositores, para constatar que existen diferencias de detalle más o menos importantes en la concepción de las reglas. Las propuestas en el presente curso tienen la ventaja de ser a la vez razonables y bastante simples. Al comienzo de los estudios será necesario, primeramente, aprender a respetar estrictamente las reglas que tienen preferencia para vuestro profesor. Un poco más tarde vuestro oído aprenderá a ser el último juez, según las circunstancias particulares. De hecho, las diferencias de detalle en los principios de base escogidos no tendrán consecuencias reales en la calidad de vuestra escritura una vez que hayáis adquirido experiencia.

© Michel Baron - Utilización con fines comerciales estrictamente prohibida.

Utilización autorizada con fines personales o pedagógicos solamente.

8.7 Enlaces world wide web

Lista de enlaces interesantes con contenidos relativos a la enseñanza de la música.

- Colección de bibliografía del autor de este proyecto, recopilada en la herramienta colaborativa *on-line* "Mendeley", sobre tecnología, música y educación, con particular énfasis en el tratamiento informatizado de la armonía tonal.

<http://www.mendeley.com/research-papers/collections/1464271/eMusicLearning/>

- LenMus: <http://www.lenmus.org/>
- Repositorio Subversion con el código fuente de LenMus, que incluye el Analizador Tonal desarrollado en este proyecto:

Rama congelada con el Analizador Tonal tal como quedó al finalizar:

<https://lenmus.svn.sourceforge.net/svnroot/lenmus/branches/RB-4.2.2>

Rama oficial, en evolución:

<https://lenmus.svn.sourceforge.net/svnroot/lenmus>

- Algoritmos para analizar música, de Daniel Sleator y Davy Temperley. "*The Melisma Music Analyzer*": <http://www.link.cs.cmu.edu/music-analysis/>
- Página dedicada al uso de la tecnología para la enseñanza de la música

Using Technology in Music Education: <http://kellysmusic.ca/education.htm>

Music Theory & Ear Training Software:
http://kellysmusic.ca/music_theory_ear_training_software.asp#Recommendations

- Lecciones de teoría para ayudar a aprobar el examen de la ABRSM
<http://www.mymusictheory.com/>

- Recursos para enseñanza de la música
<http://www.educacionmusical.es/>
<http://musica.rediris.es/>
<http://www.musicareas.com/>
<http://www.xtec.es/rtee/esp/links/direc.htm>
<http://es.wikipedia.org/wiki/Portal:Música>
<http://www.educasites.net/musica.htm>

- *Webblog* sobre educación musical
<http://didacmus.blogspot.com/>

- Actividades y juegos para aprender y practicar el Lenguaje Musical
<http://www.aprendomusica.com/lenguaje.htm>

- Cursos de música *on-line*
<http://www.insidethemusic.co.uk/>
<http://www.musictheory.net>
<http://www.jisc.ac.uk/>
<http://www.teoria.com/indice.htm>
<http://www.musicawareness.com/>

- Portal de "*The MayDay Group*", sobre música y educación.

<http://www.maydaygroup.org/>

- Enseñanza de la armonía:

<http://www.ruedaarmonica.com/>

<http://www.teoria.com/indice.htm>

<http://michelbaron.phpnet.us/armonia.htm>

http://musicareas.com/cursos/quintas/quintas_in.phtml

- Ayudas a los estudiantes de secundaria para aprender música:

<http://www.aprendemusica.es>

- Música y computación:

Annual international conference on Music Information Retrieval: <http://www.ismir.net/>

Music Information Retrieval research <http://music-ir.org/>

A database of Music Information Retrieval systems <http://mirsystems.info/>

SMC: The Sound and Music Computing research community <http://smcnetwork.org/>

Digital Music Research Network <http://www.elec.qmul.ac.uk/dmrn/>

International Symposium on Computer Music Modeling and Retrieval

<http://www.cmmr2010.etsit.uma.es/>

AdMIRE: International Workshop on Advances in Music Information Research

<http://www.cp.jku.at/conferences/AdMIRE2010/>

- Música desde distintas perspectivas:

The Conference on Interdisciplinary Musicology <http://www.uni-graz.at/~parncutt/cim/>

- Proyecto europeo **i-Maestro** sobre música, tecnología y educación:

<http://www.i-maestro.org/>

- El software libre en la educación

http://es.wikibooks.org/wiki/El_software_libre_en_la_educaci%C3%B3n/Ejemplos_Pr%C3%A1cticos

http://es.wikibooks.org/wiki/El_software_libre_en_la_educaci%C3%B3n/Experiencias

<http://www.miescuelayelmundo.org/spip.php?article305>

<http://www.alcancelibre.org/article.php/software-libre-y-la-educacion>

8.8 Glosario musical español-inglés

El código fuente de LenMus está en inglés. Por ello, este glosario puede servir de ayuda para poder seguirlo y entenderlo.

Acento: *accent*

Acorde: *chord*

Acorde de séptima dominante: *dominant seventh chord*

Altura: *pitch*

Alteración: *accidental*

Bajo continuo, bajo cifrado: *figured-bass*

Barra: *bar line*

Bemol: *flat*

Blanca: *half note*

Blanca: *minim*

Clave: *clef*

Clave de fa: *bass clef*

Cifrado de acordes: *chord notation*

Conducción de voces: *voice leading*

Compás: *bar; measure*

Corchea: *quaver; eighth note*

Escala diatónica: *diatonic scale*

Fundamental (nota): *base/root note*

Fundamental (estado): *root position*

Grado: *degree, step*

Grupos: *beamed group*

Grupos de valor especial (tresillo, cinquillo): *tuplet*

Intervalo: *interval*

Intervalo aumentado/disminuido: *augmented/diminished interval*

Intervalo justo: *perfect interval*

Inversión: *inversion*

Ligadura: *tie, slur*

Línea auxiliar: *ledger line*

Marca de tiempo, métrica: *time signature*

Negra: *crotchet, quarter*

Nota: *note*

Octava: *octave*

Partitura: *score*

Pentagrama: *staff* (pentagramas: *staves*)

Puntillo: *dot*

Pulso: *beat*

Redonda: *whole*

Redonda: *semi breve*

Semicorchea: *sixteenth note*

Semicorchea: *semi quaver*

Silencio: *rest*

Sostenido: *sharp*

Tonalidad: *key signature*

Tríada: *triad*

Unísono: *unison*

Voz: *voice*