

LenMus architecture

LenMus was conceived as a reader for interactive eBooks, a kind of reader, such as Adobe Acrobat Reader but oriented to interactive music books. So its functionality is basically a main frame with a tabbed interface, a tab for each book/document that the user opens. Later I added the possibility to open and edit music scores, but the interface doesn't change: a tab for each score the user opens.

To understand LenMus let's start with the eBooks format. Music eBook are zip files. Please, go to folder `lenmus/locale/en/books/` and open the zip file `L2_MusicReading.lmb`. You will see the following content:

Name	Size	Type
content	367,6 kB	Folder
fonts	0 bytes	Folder
images	27,9 kB	Folder
META-INF	3,9 kB	Folder
mimetype	27 bytes	unknown

Folder META-INF contains only the TOC for the eBook. And folder 'content' contains the pages for the eBook. Open it and you will see the pages:

Name	Size
L2_MusicReading_cover.lmd	5,2 kB
L2_MusicReading_thm205.lmd	5,9 kB
L2_MusicReading_thm205_E1.lmd	5,2 kB
L2_MusicReading_thm205_E2.lmd	6,4 kB
L2_MusicReading_thm206.lmd	5,4 kB
L2_MusicReading_thm206_E1.lmd	5,3 kB
L2_MusicReading_thm206_E2.lmd	5,5 kB
L2_MusicReading_thm207.lmd	5,9 kB
L2_MusicReading_thm207_E1.lmd	5,2 kB
L2_MusicReading_thm207_E2.lmd	6,1 kB
L2_MusicReading_thm208.lmd	6,1 kB
L2_MusicReading_thm208_E1.lmd	5,4 kB
L2_MusicReading_thm208_E2.lmd	5,7 kB
L2_MusicReading_thm209.lmd	5,8 kB
L2_MusicReading_thm209_E1.lmd	5,8 kB
L2_MusicReading_thm209_E2.lmd	5,3 kB

If you open now one of the pages you will find an xml document with the content for that page. Please, open and examine in detail one document, for instance, L2_MusicReading_thm205.lmd. You will see a content similar to that of an HTML page but using different tags. You will notice also tags `<ldpmusic>` that contain scores in LenMus format (LDP) (<https://lenmus.github.io/ldp/>). And a tag `<scorePlayer>`: it is for inserting a 'play' control, for now just a play button, but in future it could be improved.

Now let's examine file L2_MusicReading_thm205_E1.lmd. At start you will see the style tags but if you advance, its content is practically nothing, a tag `<dynamic classid='TheoMusicReading'>` with some params.

Tag `<dynamic>` is equivalent to the html `<object>` tag. It is a placeholder for content injected dynamically by the browser. In this case, tag `<dynamic>` is the place at which LenMus will insert dynamically generated content, and the children tags are the parameters to determine what content to be inserted. In this case, attribute `classid='TheoMusicReading'` informs that the content to insert is an exercise of music reading, and the parameters determine the complexity of the score to generate.

For instance, this document: (I move the image to the next page to facilitate comparison)

```

<lenmusdoc vers='0.0' language='es'>
<styles> ... </styles>
<content>
  <section level='1' style='eBook_heading_1'>Subdivisión ternaria: corchea
    ligada a semicorcheas</section>
  <para style='eBook_para'>Pulsa 'Tocar' y escucha el ritmo. Entrena tu oído y
    practica la lectura de los nuevos patrones.</para>
  <dynamic classid='TheoMusicReading'>
    <param name='control_go_back'>L2_MusicReading_thm208.lmd</param>
    <param name='control_play'></param>
    <param name='fragment'> ... </param>
    <param name='clef'>G;a3;a5</param>
    <param name='clef'>F4;a2;f4</param>
    <param name='time'>128</param>
    <param name='key'>C</param>
  </dynamic>
</content>
</lenmusdoc>

```

Will display the following:

<section level='1' style='eBook_heading_1'>Subdivisión ternaria: corchea ligada a semicorcheas</section>

<dynamic classid='TheoMusicReading'>

Subdivisión ternaria: corchea ligada a semicorcheas

Pulsa 'Tocar' y escucha el ritmo. Entrena tu oído y practica la lectura de los nuevos patrones.

☐ Comenzar marcando un compas ☐ Tocar con metrónomo

[Volver a la teoría](#) [Nuevo problema](#) [Tocar](#)


<param name='control_go_back'>

<param name='control_play'>

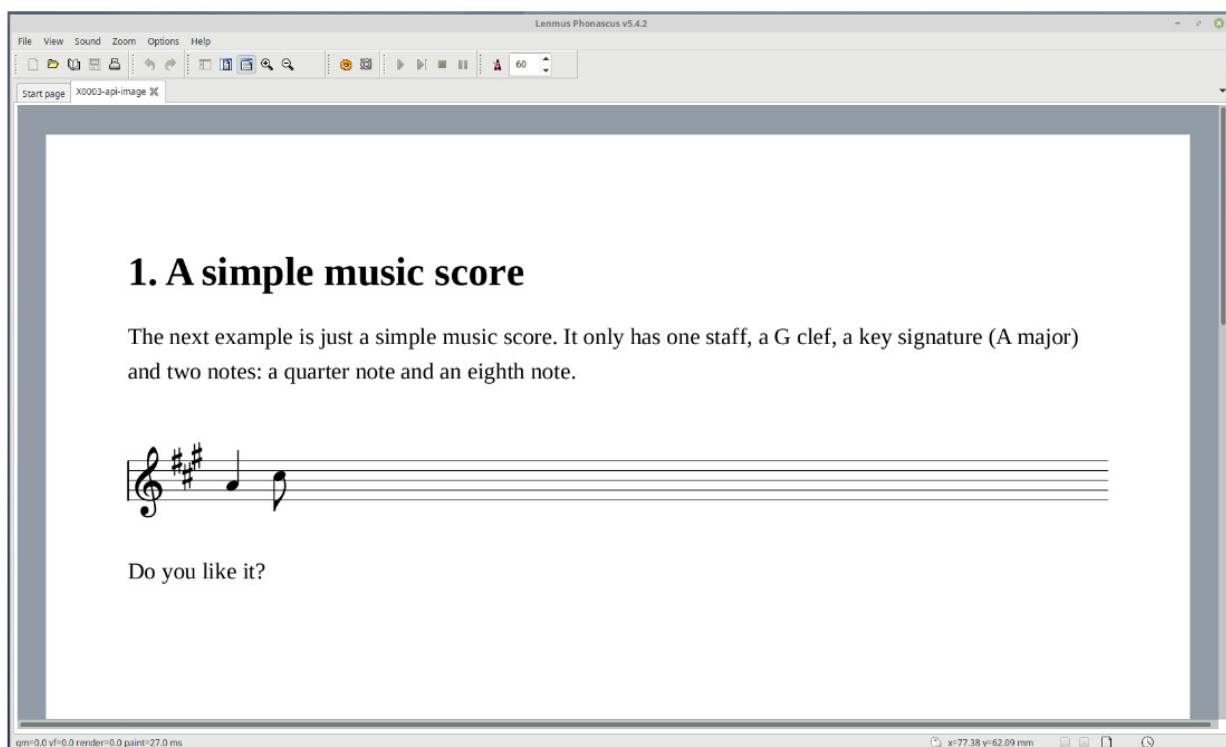
<para style='eBook_para'>Pulsa 'Tocar' y escucha el ritmo. Entrena tu oído y practica la lectura de los nuevos patrones.</para>

Here is another example, this time a document with one score:

```
<?xml version="1.0" encoding="UTF-8"?>
<lenmusdoc vers="0.0" language="en">
  <content>
    <section level="1">1. A simple music score</section>
    <para>
      The next example is just a simple music score.
      It only has one staff, a G clef, a key signature (A major)
      and two notes: a quarter note and an eighth note.
    </para>
    <ldpmusic>
      (score (vers 2.0)
        (instrument
          (musicData
            (clef G)
            (key A)
            (n a4 q)
            (n c5 e)
          )
        )
      )
    </ldpmusic>
    <para>Do you like it?</para>
  </content>
</lenmusdoc>
```

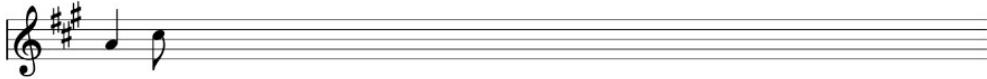


And the result:



1. A simple music score

The next example is just a simple music score. It only has one staff, a G clef, a key signature (A major) and two notes: a quarter note and an eighth note.

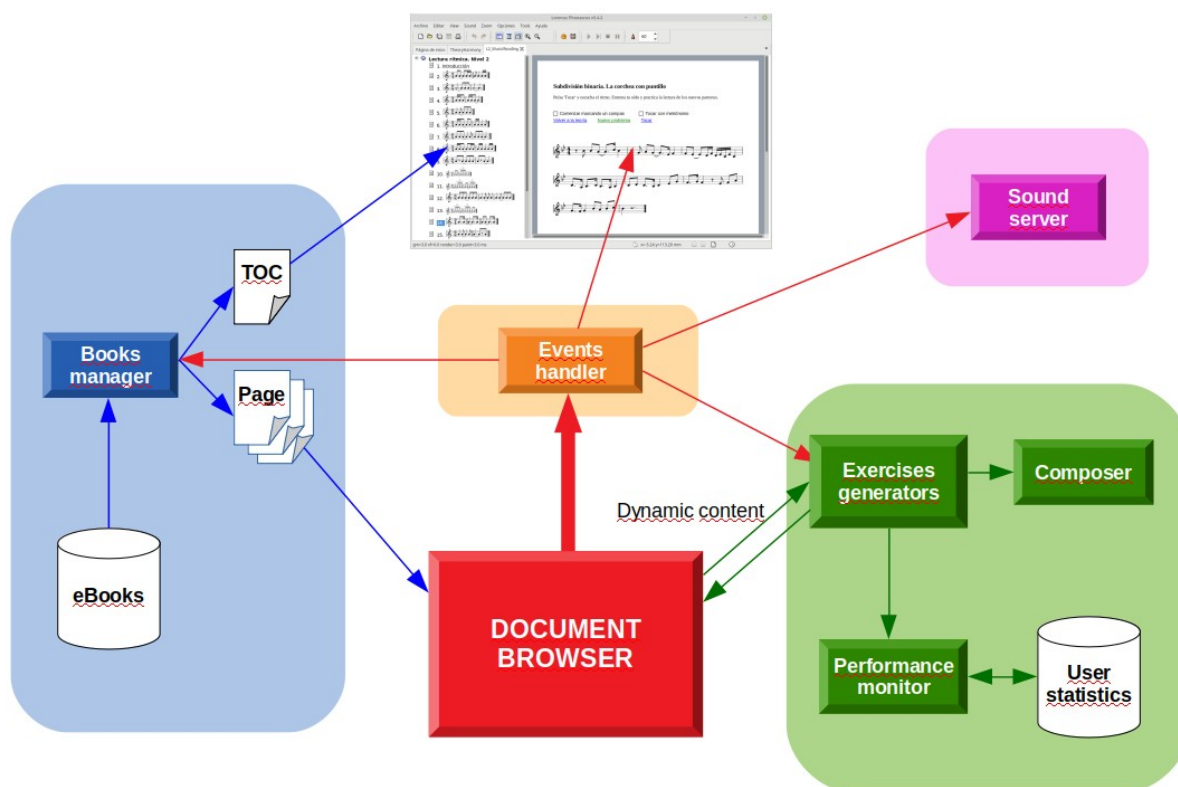


Do you like it?

So, in conclusion, the **important points to learn**:

1. Lenmus documents are similar to html documents, with texts, controls and dynamic content, and also have a tag for embedding scores in the content.
2. Lomse is a rendering engine for lenmus documents. Also it manages interaction with the document and generates events when the user interacts with the document.

The following picture shows the LenMus program architecture:



There are six main blocks:

1. **The GUI.** The associated code is mainly in folder 'app'. The most important files are `lomse_main_frame.cpp` and `lomse_document_canvas.cpp`. This code is responsible for initializing/terminating the app, for managing the main window with the tabs for the open documents, and for generating actions in response to toolbar / menubar user actions.
2. **The Books Manager.** It is responsible for unpackaging the eBooks and extracting the TOC and the pages. The code is in file `app/lenmus_book_reader.cpp`. This code is very old and I don't like it, but works and there is no need to deal with it, so probably you will never have to review it or to patch it.

3. **The Events system.** Well, this is just the wxWindows events system. The only thing to note is that LenMus is event driven. So, any action generates an event and the different component handle it. The lomse library generates events.
4. **The Sound Server.** For scores playback, lomse does not generate sounds but generate *sound-events*. These events are handled by the user application (LenMus app. in this case) and generates the requested sound. Lomse is platform independent and know nothing about how to generate sounds. So, responsibility is transferred to the user application that must generate sounds using whatever it likes: midi, wave synthesis, or other. The main file is `sound/lenmus_midi_server.cpp` but, probably, you will never have to patch it.
5. **The Exercises Generator.** When a `<dynamic>` tag is found in the document, lomse generates an event requesting to inject the content in the document. This event is handled in `MainFrame::on_lomse_request()` [`lenmus_main_frame.cpp`, line 1470] and invokes `generate_dynamic_content()` method. And this triggers the injection of content. The injected content is always an exercise. The code for generating exercises is in folder 'exercises' and is, perhaps, the most important part of the application and the one we will have to touch for creating new exercises.
6. **The Document Browser.** It is responsible for loading document content into memory, and rendering it. It is also responsible for generating events when the user modifies the document or interacts with the controls (links, buttons, play score control, etc.) in the document. As you have deduced, this is the lomse library.

In addition, there are other helper code:

- Folder '*auxmusic*' is code for representing and managing music related concepts: intervals, scales, chords, etc.
- Folder '*dialogs*' contains the auxiliary dialogs displayed by the application.
- Folder '*globals*' contains singletons and other global objects. The most important are:
 - `globals/lenmus_paths.cpp`, that manages the paths to access LenMus resources: icons, eBooks, etc.
 - And `globals/lenmus_injectors.cpp`, that is responsible for all global objects, such as the Path instance.
- Folder '*options*' contains the code for managing user options, such as language to use.
- Folder '*properties*' contains code related to editing the properties of music notation symbols. This code is not yet fully developed. The actual code are just first attempts and tests for developing the score editor. You can ignore this code.
- Folder '*tests*' contain unit tests. But I never developed LenMus using TDD, so there are very few tests and are reduced to specific issues I needed to tests.
- Folder '*toolbox*' contains the code related to the UI for the score editor. You can ignore it for now. Moreover, the plan is to remove the score editor from the LenMus application so, probably, all this code will be removed.

- Folder *'updater'* contains the code for checking if there are new LenMus releases and in that case, inform the user for updates. This is how things were done in 2004. Probably we should reconsider this code.
- Folder *'widgets'* contains a few wxWidgets derived classes, such as a message box dialog used in LenMus. Probably you will never have to deal with this code.
- Folder *'xml_parser'* contains just some wrapper code to deal with xml.

Navigating the code

To start understanding the code let's go to the application initialization and the code in folder *'app'*. Method `TheApp::OnInit()` in file `lenmus.app.cpp` is the starting point, the equivalent to the `main()` function in C. This method just creates all the needed resources, creates the main frame and returns. And wxWidgets passes control to the main frame (file `lenmus_main_frame.cpp`), that remains idle waiting for user events.

The main frame is just a manager for documents. When the user asks to open a book or an score, control arrives to `MainFrame::load_file()`. It asks `DocumentLoader` to open and load the content and to create a canvas window that will be displayed in a new tab.

`DocumentLoader` (in file `lenmus_document_frame.cpp`) in method `DocumentLoader::create_canvas()` creates a canvas appropriate for the type of document: a normal window for scores or a splitted window for eBooks. And asks the canvas to display the document. Control arrives then to `DocumentFrame::display_document()` where, depending on file type, just loads the content in the window (scores) or loads the eBook and displays the TOC in the left pane and the first page in the right pane.

If it is an eBook, method `BooksCollection::add_book()` is invoked:

```
291:    if (!m_pBooksData->add_book(document))
```

The book manager reads the book, extracts the TOC and the pages, and creates the list of pages. But this code is not important. Let's move to how the eBook page is displayed. This takes place when execution arrives to `DocumentWindow::display_document()` (in file `lenmus_document_canvas.cpp`, line 563). Here the important point is the line

```
581:    m_pPresenter = m_lomse.open_document(viewType, filename, reporter);
```

In this line the page to be displayed is just passed to the lomse library, and lomse takes care of loading its content in memory. A few lines down control arrives to:

```
590:    do_display(reporter);
```

and from there to method `DocumentWindow::do_display()` [line 646]. In this method:

- LenMus provides lomse with a buffer for rendering the document:

```
653:    spInteractor->set_rendering_buffer(&m_rbuf_window);
```

- Sets some event handlers and ... that's all, because after creating a pane and loading content on it, wxAuiNotebook / wxFrame will issue an *resize* event followed by an *repaint* event.

The *repaint* event arrives to `DocumentWindow::on_paint()` [line 727] and there, basically, two important things are done:

1. Method `update_rendering_buffer()` is invoked:
752: `update_rendering_buffer();`
This generates a request to lomse to render the document in the passed bitmap:
966: `pInteractor->redraw_bitmap();`
2. Method `copy_buffer_on_dc()` is invoked:
753: `copy_buffer_on_dc(dc);`
and the bitmap buffer passed to lomse is copied onto the window screen buffer.

And the main frame gets idle waiting for new events.

As you can see, until now LenMus code is only for managing user events and managing tabs and windows. But the real work of loading document content and rendering it is performed by the lomse library.

In summary, the main files you will probably manage will be:

- `lenmus_main_frame.cpp`
- `lenmus_document_canvas.cpp`
- and the files in *exercises* folder.

How exercises work

Main files

All code related to exercises is in folder 'exercises'.

In first versions of LenMus each exercise was an independent class, but as exercises share a lot of behaviour and code, with the time exercises evolved, and nowadays they are implemented as a hierarchy of classes, and all exercises derive from base class `EbookControl` (files `exercises/ctrls/lenmus_exercise_ctrl.cpp` & `.h`). It is an abstract class for any kind of control that is included in an eBook. From it, the following classes are derived:

- **ExerciseCtrl**. An abstract class for any kind of exercise included in an eBook.
- **CompareCtrl**: Abstract class to create exercises to compare scores/sounds
- **CompareScoresCtrl**: Abstract class to create exercises to compare two scores
- **OneScoreCtrl**: Abstract class to create exercises with one problem score. The score is presented to the user (either displaying the score – theory mode – or playing back the score – aural training –). And the student must say something about the score content. For instance, an exercise to identify an scale.

- **FullEditorCtrl**: Abstract class for any kind of exercise using the score editor. It was first created for music dictation exercises. These exercises are coded but were never published because we need to write the eBook and improve the scores generator.

And some examples of specific exercises (classes derived from the above abstract classes):

- **EarCompareIntvCtrl** derived from **CompareScoresCtrl** (in file `lenmus_ear_compare_intv_ctrl.cpp & .h`). An aural training exercise to compare two intervals. The student must say which one is greater.
- **EarIntervalsCtrl** derived from **OneScoreCtrl** (in file `lenmus_ear_intervals_ctrl.cpp & .h`). An aural training exercise to listen an interval and identify it.
- **IdfyCadencesCtrl** derived from **OneScoreCtrl** (in file `lenmus_idfy_cadences_ctrl.cpp & .h`). A class to generate theory and aural training exercises for identifying cadences. In aural training mode, the student will hear a cadence and must identify its type. In theory mode, the student is presented an score with a chord progression and must identify the cadence type. As you can see, the exercise is just generating an score with a chord cadence and to present it to the student. In aural training the score is not displayed but it is played back. In theory mode the score is displayed but it is not played back.

The naming convention for exercises is:

- **Ear** prefix, for exercises valid only for aural training
- **Theo** prefix, for exercises valid only for theory.
- **Idfy** prefix for, exercises supporting for both modes: aural training and theory.

I'm not going to review here every existing exercise. All are coded following the same patterns, so it is enough to understand one of them for grasping the general idea.

Therefore, I'm going to review the **IdfyScalesCtrl**, derived from **OneScoreCtrl** (in file `lenmus_idfy_scales_ctrl.cpp & .h`). It is a class to generate theory and aural training exercises for identifying scales. In aural training mode, the student will hear an scale and must identify its type. In theory mode, the student is presented an score with an scale and must identify it.

Exercise parameters

Any exercise needs some configuration parameters, such as key signatures to use, intervals to generate, etc. They are provided in the eBook file, in the children of the `<dynamic>` tag. For instance, for the score identification in theory mode:

```

<dynamic classid='IdfyScales'>
  <param name='height'>35</param>
  <param name='control_settings'>TheoIdfyScale</param>
  <param name='mode'>theory</param>
</dynamic>

```

classid attribute specifies the exercise type. When lomse request to inject the content for the <dynamic> tag, execution arrives to `MainFrame::generate_dynamic_content()` and there, a `DynControl` object is created by invoking a factory method:

```

1505: DynControl* pControl
      = DynControlFactory::create_dyncontrol(m_appScope, classid, pWnd);

```

In this example, the generated `DynControl` object will be an `IdfyScalesCtrol` object. See `DynControlFactory::create_dyncontrol()` in file `exercises/ctrls/lenmus_dyncontrol.cpp`.

After control is created its `generate_content()` method is invoked:

```

1507: pControl->generate_content(dyn, doc);

```

As the layout of all exercises is similar, `generate_content()` method is not a method implemented in each exercise, but a method handled in any of the abstract classes, in this case in `EbookCtrol::generate_content()`. There, two specific implementations in derived class are invoked:

```

94:   get_ctrol_options_from_params();
95:   initialize_ctrol();

```

And execution arrives, first, to `IdfyScalesCtrol::get_ctrol_options_from_params()` (file `lenmus_idfy_scales_ctrl.cpp`, line 82). This method is responsible for parsing the parameters in <dynamic> tag and preparing a `Constrains` object containing all the settings for the exercise. For each exercise, there it is necessary to create a `Constrains` object (files `exercises/constrains/lenmus_xxxxxxx_constrains.cpp`, in this example the applicable file is `lenmus_scales_constrains.cpp`). And also it is necessary to create a parser for the params. These parsers are a base class `EbookCtrolParams` (implemented in `exercises/params/lenmus_exercise_params.cpp`) and a derived class, specific for each exercise, currently implemented in a header file (`include/lenmus_xxxxxxx_ctrol_params.h`, in this example `lenmus_idfy_scales_ctrol_params.h`).

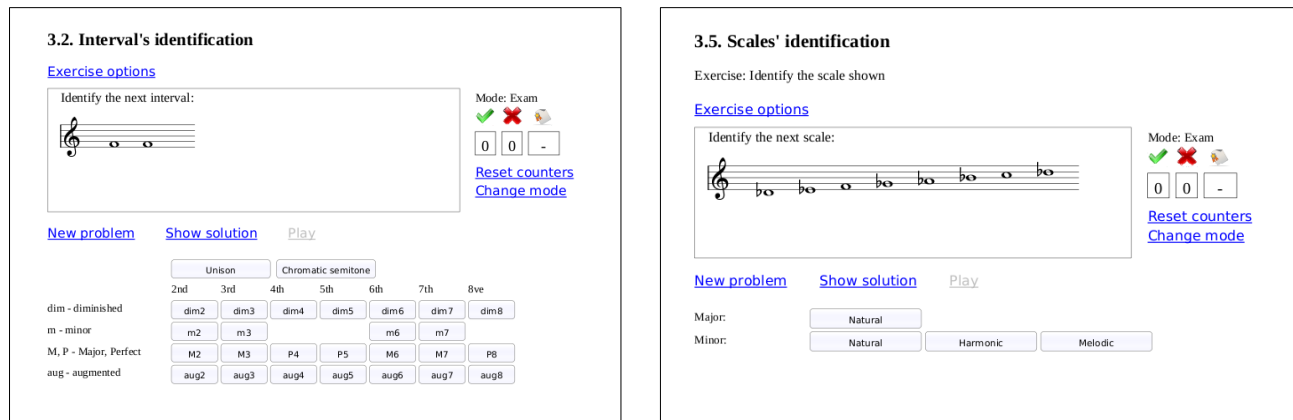
Once the parameters are parsed, method `initialize_ctrol()` is invoked. It is also an specific method that must be implemented in each exercise. In this example, it is `IdfyScalesCtrol::initialize_ctrol()` (file `lenmus_idfy_scales_ctrl.cpp`, line 69). Normally this method just creates and injects the content for the exercise by invoking `create_controls()` method.

How is content injected?

The content is not injected as source code with xml tags but modifying the document internal model by creating and adding more nodes to the tree. You can consider the

internal model as a DOM like structure: the root object represents the document and the children of this root object represent the blocks composing its content: headers, paragraphs, music scores, lists, tables, images, etc. For music scores the DOM model analogy is not truly feasible as many music notation markings (like slurs or beams) require additional structures for correctly representing a music score.

The exercise we are analyzing, `IdfyScalesCtrl`, derives from `OneScoreCtrl`. All exercises derived from this base class have a similar layout (see the following images):



`OneScoreCtrl` layout is as follows:

1. Paragraph with the "Exercises options" link.
2. A '*multicolumn*' object. It is like a table with a single row, and many columns, in this case two columns: the left one for the score, and the right one for the counters.
3. A paragraph with links for actions: "New problem", "Show solution" and "Play".
4. Six paragraphs for the answer buttons. The six paragraphs with the buttons always exist but, depending on the exercise options, some buttons are disabled and hidden.

And for the `IdfyScalesCtrl` this is the injected objects structure:

```
(dynamic
  (paragraph (control)) //The link "Exercise options"
  (multicolumn (content
    (content //left column: a box with the problem
      (paragraph (text)) //Text: "Identify the next scale"
      (score) //The score with the scale
    )
    (content //right column: the counters control
      (paragraph (control))
    )
  ))
  (paragraph
    (control) // "New problem" link
    (wrapper) // to add space
    (control) // "Show solution" link
    (wrapper) // to add space
    (control) // "Play" button
  )
)
```

```

(paragraph //Answer buttons: first line
  (wrapper (text)) "Major" label
  (wrapper (button)) "Natural" button
  (wrapper (button)) hidden, used for advanced exercises
  (wrapper (button)) hidden
  (wrapper (button)) hidden
)
(paragraph //Answer buttons: second line
  (wrapper (text)) "Minor" label
  (wrapper (button)) "Natural" button
  (wrapper (button)) "Harmonic" button
  (wrapper (button)) "Melodic" button
  (wrapper (button)) hidden
)
(paragraph //Answer buttons: third line
  (wrapper (text)) hidden
  (wrapper (button)) hidden
  (wrapper (button)) hidden
  (wrapper (button)) hidden
  (wrapper (button)) hidden
)
(paragraph //Answer buttons: fourth line
  (wrapper (text)) hidden
  (wrapper (button)) hidden
  (wrapper (button)) hidden
  (wrapper (button)) hidden
  (wrapper (button)) hidden
)
(paragraph //Answer buttons: fiveth line
  (wrapper (text)) hidden
  (wrapper (button)) hidden
  (wrapper (button)) hidden
  (wrapper (button)) hidden
  (wrapper (button)) hidden
)
(paragraph //Answer buttons: sixth line
  (wrapper (text)) hidden
  (wrapper (button)) hidden
  (wrapper (button)) hidden
  (wrapper (button)) hidden
  (wrapper (button)) hidden
)
)
)

```

As this objects structure is common to all OneScoreCtrl exercises, these objects are created in base classes, as follows:

Base class method `ExerciseCtrl::create_controls()` (file `lenmus_exercise_ctrl.cpp`, line 219 – **please review this code in detail** →) is the most important. It first translates the strings:

```
226: initialize_strings();
```

Next, creates the '*problems manager*' (I will explain this later). For now, just to know that it is responsible for generating problems:

```
230: create_problem_manager();
```

And next, the different objects are created and injected in the document. You will see that *ImoParagraph* objects are created (the equivalent to an HTML element node for a

paragraph) as well as other *ImoXXXXX* objects (internal model objects, equivalent to HTML element nodes). Please, review the code; it should be easy to understand, more easy than trying to explain it!).

To customize the answer buttons, an specific method in derived class is invoked:

```
342: create_answer_buttons();
```

and derived class method `IdfyScalesCtrol::create_answer_buttons()` is invoked. There, the labels for columns, rows and buttons are set, and also the number of buttons is defined. Also (line 131) even handlers are defined, for receiving a click event when the user clicks a button.

And later, in `ExerciseCtrol` base class, the next line:

```
355: display_first_time_content();
```

jumps to line 776:

```

775 //-----
776 void ExerciseCtrol::display_first_time_content()
777 {
778     if (is_theory_mode())
779         new_problem();
780     else
781         display_initial_msge();
782 }
783
```

If the exercise is in theory' mode, method `new_problem()` is invoked, to generate and display the score with the problem. Otherwise, method `display_initial_message()` is invoked to generate the problem score, display a message introducing the problem (e.g. "Identify the scale that will sound"), and start playback of the score.

Let's see method `ExerciseCtrol::new_problem()` (in line 738). This method is invoked now and also every time the user clicks the "New problem" link, so first thing to do is to clear data from previos problem:

```
741: reset_exercise();
```

update counters and reset answer buttons:

```
749: m_pCounters->OnNewQuestion();
```

```
750: enable_buttons(true);
```

And generate a new problem. For this, derived class method is invoked:

```
753: wxString sProblemMessage = set_new_problem();
```

Control arrives then to `IdfyScalesCtrol::set_new_problem()`. A new score is generated (see `IdfyScalesCtrol::prepare_score()`) and saved in member variable `m_pProblemScore`. The generated score is not yet injected into the internal model, it is just a detached subtree, to be attached to the internal model when it is necessary to display the score. For theory problems, the score is attached to the model when finishes `IdfyScalesCtrol::set_new_problem()` and control returns to `ExerciseCtrol::new_problem()` when it invokes:

```
758: display_problem_score();
```

And execution arrives to `OneScoreCtrol::display_problem_score()`. For ear training exercises the score is not attached to the model. Instead it is played back and will be attached to the model when the solution must be displayed.

The Problem Manager: exercises operation modes

The Leitner method

Usually, all learning takes place through repetition. When a new piece of information is presented, human memory tends to forget it if it is not recalled in a certain time period. But if that piece of information is presented again, before being totally forgotten, the memory strengthens and the retention period became greater. I read somewhere that 48 hours after a study session, we have generally forgotten 75% of the presented material. This is one of the key points of the *spaced repetition* method: each time we review a piece of information, our memory becomes stronger and we remember it for longer.

In the days before computers, it was a common practice to write the facts to learn on a set of cards (called *flashcards*), look at each card in turn, think of the answer, then turn the card over, and take the next card. But, constantly reviewing everything is not an optimal method, and there were no guidelines for deciding when to next review the cards: the next day? Every day for a week? Once a week every month?. Another problem of this method is that easy questions end up being repeated just as often as difficult ones, which means either you're not reviewing the difficult questions enough, or you're reviewing the easy ones too often. In any case, your time or memory suffers.

In 1972, a German science journalist named Sebastian Leitner wrote the book "*How to learn to learn*", a practical manual on the psychology of learning, that became a bestseller and popularized a new and simple method of studying flashcards. In the *Leitner method* (http://en.wikipedia.org/wiki/Leitner_system) also known as *spaced repetition* (http://en.wikipedia.org/wiki/Spaced_repetition) learning technique or *flashcards method*), a box is divided up into a bunch of compartments. Each compartment represents a different level of knowledge. Initially, all cards are in compartment 1. When you remember a card correctly you move it to the next compartment. If you forget, you move it back to the start.

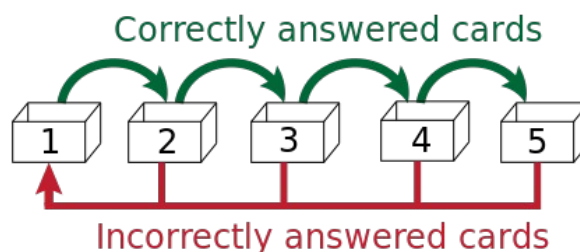


Figure: In the Leitner system, correctly answered cards are advanced to the next, less frequent box, while incorrectly answered cards return to the first box (picture taken from Wikipedia at http://en.wikipedia.org/wiki/Leitner_system. Available under the Creative Commons CC0 1.0 Universal Public Domain Dedication).

The advantage of this method is that you can focus on the flashcards that you have problems to remember, which remain in the first few compartments or boxes, and from

time to time review the questions in the other boxes. The result is a reduction in the amount of time needed to study a subject.

LenMus adaptation for music exercises

Originally, LenMus exercises were not based on any particular learning methodology. Questions were selected just at random and easy questions were repeated annoyingly. Therefore, in version 4.1 I started to experiment with the idea of adding support for the Leitner methodology in a couple of exercises.

The first tested algorithm was a direct implementation of Leitner method. I tested it with the intervals identification exercise (theory). The results were quite bad because the Leitner method is suited for problem spaces where you have to memorize the answer to a question. But in most music theory exercises, the objective is not to memorize an answer but to learn a concept (i.e. "3rd major interval") and quickly identify examples of the concept that can have different representations in a music score.

In the original Leitner method for learning facts, a correct answer means that you know the fact to learn. But when you are studying concepts, a correct answer means that the student always succeed in recognizing an example of the concept. So, presenting just one example of the concept and recognizing it is not enough to conclude that the concept is learn. In these cases, the direct implementation of Leitner method leads to a lot of irrelevant repeated questions: for instance all possible 3rd major intervals!

By considering how a human teacher could determine if a student has learn a concept (i.e. 3rd major interval), I concluded that the program should consider that the concept is learn not when all possible 3rd major intervals has been displayed and the student has successfully recognized all them but when a certain number of 3rd major intervals has been presented and successfully recognized without failures.

Therefore, to adapt the Leitner method for learning concepts instead of facts, I kept the idea of demoting a question if it is failed, but I introduced a new criteria for promoting a question: the consecutive number of samples of a concept that was successfully recognized by the student (RT, *Repetition Threshold*). The idea is to assume that a concept has been learn when the student has successfully recognized RT samples of the concept without a failure. This modified algorithm is presented in Annex 1. It is the current algorithm used in LenMus.

Exercises: operation modes

All LenMus exercises have at least two operation modes: 'exam' and 'quiz'.

- In 'exam' mode questions are selected just at random. Neither student performance data nor the answers to previous questions are taken into account to formulate the next question. At any moment, all possible questions have the same probability of being asked. This mode is useful for testing the student knowledge before taking an examination.

- The '*quiz*' mode is similar to the '*exam*' mode but two answer counters are displayed and questions are accounted in both counters: one in first counter and the next one in the second counter. This mode is useful to work in pairs or in teams at classroom.

In version 4.1, as a proof of concept, I started to implement the Leitner methodology. As a consequence, in the exercises adapted for using the Leitner methodology two additional operation modes were available: '*learning*' and '*practising*'.

- In '*learning*' mode the program analyzes the student answers and schedule questions to systematically cover all the subject, focusing on those questions that are troubling the student. This mode is based on the Leitner method and is the most systematic one. The student performance data is saved and the next time he/she returns again to the exercise, the program takes care of asking questions to ensure an optimal learning path. The result is, ideally, a reduction in the amount of time needed to study a subject and the assurance that the subject has been systematically reviewed.
- In '*practising*' mode the program uses the data, saved in learning mode about the student performance, to choose questions. It selects questions at random but giving more probability to those that are troubling the student. Performance data is not saved in this mode. This mode is useful when the student has finished his/her daily assignment in 'learning' mode but he/she would like to practise more.

Implementation details

Organizing concepts into cards and decks

A concept is studied along several course grades. In each grade, the concept is studied with more detail. Also, for each grade, there can be difficulty levels.

Therefore, the first step is to analyze and define the course grades. This has to be done in any case, even if Leitner method is not used, and it is a consequence of the chosen syllabus. It is also a requirement for defining exercise programming specifications.

The additional task to perform, when Leitner method is going to be used, is to group exercise settings to define the difficulty levels for each grade. And then, the flashcard decks are defined, by grouping questions by grade and difficulty level. As an example, in the following section the process for defining the question decks for the interval exercises is examined in detail.

Terminology

The following terminology will be used:

- *Problem space*: The set of questions for a level of an exercise.
- *Deck*: A fixed set of questions forming a coherent and logical group of cards to learn an aspect of a subject. A deck corresponds to the group of questions

added/removed when in the exercise settings dialog, an specific value is chosen/removed.

I decided to try an abstract implementation of the modified Leitner method so that the same code could be used by all exercises. A problem space is defined by a vector of *flashcards* (question items) containing all data about questions, user performance and current box. So each question is characterized by the following information:

- question index (0..n)
- current box
- number of times this question was asked in current session
- number of times this question was correctly answered in current session
- number of times this question was asked (total)
- number of times this question was correctly answered (total)

Thus, each question can be represented by its question index and the Leitner method implementation needs to know nothing about each question content: it just deals with question indexes. It is responsibility of each exercise to define the real questions and to assign indices to them. The exercise object will create and pass the problem space vector to the Leitner manager. And it will be responsibility of each exercise to serialize data and relate it to a specific user.

Application for the intervals exercises

The first step is to analyses and define the requirements for course grades. The interval identification exercise will organized in four levels to match the requirements of most common syllabus:

Level 0:

Learn just the interval numbers.

Level 1:

Learn perfect, major and minor intervals, in any key signature (the intervals present in a natural scale).

Level 2:

Learn augmented and diminished intervals (i.e. one accidental introduced in C major/ A minor key signatures).

Level 3:

Learn double augmented / diminished (i.e. two accidentals are introduced in C major/ A minor key signatures).

Therefore, for each level there must be at least one deck of questions.

The second step is to define the exercise setting options that will be available for each level, and group these settings to define 'difficulty levels'. The exercise has settings to choose clefs, key signatures, and maximum number of ledger lines.

For level 0 the available settings doesn't modify the possible questions to ask. Therefore, for level 0 there will be only one deck, D_0 , containing all possible questions for these level: the name (just the number) of all intervals.

For the other levels, the selection of a clef doesn't change the set of questions, but the selection of a key signature significantly changes the set of possible questions. Therefore, it was decided to create a deck for each key signature. This leads to the need to define the following decks:

Deck D_{1k} : No accidentals. Perfect, major and minor intervals in k key signature.

Deck D_{2k} : One accidental. Augmented and diminished intervals in k key signature.

Deck D_{3k} : Two accidentals. Double augmented / diminished intervals in k key signature.

Now, the set of questions to use for each exercise level will be created by using the following decks:

Set for level 0: Deck D_0

Set for level 1: Set 0 + decks D_{1k} , for k = all selected key signatures

Set for level 2: Set 1 + decks D_{2k} , k = all selected key signatures

Set for level 3: Set 2 + decks D_{3k} , k = all selected key signatures

Repetition intervals

The repetition intervals table is taken from the one proposed by K.Biedalak, J.Murakowski and P.Wozniak in article "*Using SuperMemo without a computer*", available at <http://www.supermemo.com/articles/paper.htm> . According to this article, this table is based on studies on how human mind 'forgets' factual information.

Box	Repetition interval
0	1 day
1	4 days
2	7 days
3	12 days
4	20 days
5	1 month
6	2 months
7	3 months
8	5 months
9	9 months
10	16 months
11	2 years
12	4 years
13	6 years
14	11 years
15	18 years

Except for box 1, this table use a factor of 1.7 to increase subsequent intervals.

Progress estimation indicators

In order to provide the student with some indicators of his/her long term acquired knowledge it was necessary to study and define a suitable way of doing it. Many indicators could be proposed but the approach followed has been pragmatic, oriented to the main issue that usually worry the student: his/her probabilities to pass an examination. For this purpose, three 'achieved retention level' indicators has been defined: short, medium and long term indicators. They attempt to provide a quantitative evaluation for the student preparation for not failing any question in an examination to be taken at three time points: in the short term (i.e., in 20 days), at the end of the academic year (i.e., in 9 months) or in long term (in a few years).

For defining these 'achieved retention level' indicators it was taken into account that each box B_i represents a more consolidated level of knowledge than the preceding box B_{i-1} . Therefore, a weighting factor w_i has been assigned to each box, and the boxes have been grouped into three sets:

- Short: answers that could be forgotten in less than 20 days
- Medium: answers that could be forgotten in less than 9 months
- Long: answers that will be remembered during some years

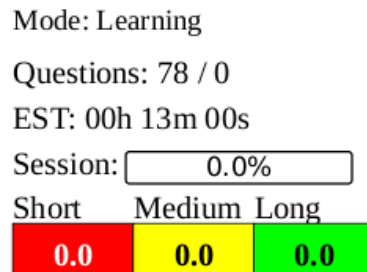
The split points are shown in following table:

Box	Repetition interval	w	Retention level
0	1 day	0.0	Short: need more work
1	4 days	0.1	
2	7 days	0.2	
3	12 days	0.3	
4	20 days	0.4	
5	1 month	1.0	Medium: need some repetitions
6	2 months	1.1	
7	3 months	1.2	
8	5 months	1.3	
9	9 months	1.4	
10	16 months	2.0	Long: known questions
11	2 years	2.1	
12	4 years	2.2	
13	6 years	2.3	
14	11 years	2.4	
15	18 years	2.5	

To have great chances of passing an examination to be taken in 20 days it should be ensured that all questions are in box B_4 or above it. Therefore, the indicator for 'short term achievement' should display 100% when all questions are in box B_4 or above it; it should mark 0% when all questions are in box B_0 ; and should advance smoothly as questions are promoted between boxes B_0 to B_4 . To satisfy these requirements I have tried different formulas. After some simulations, I finally choose the formulae implemented in the

program. *[I have to describe here but I need to learn how to include mathematical formulae in the text]*

Performance display for learning mode



Questions: Two numbers:

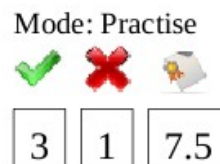
- The first one is the number of unlearned questions: those that are in group 0.
- The second one is the number of expired questions: those in higher groups whose repetition interval has arrived.

EST (Estimated Session Time): The estimated remaining time to review all questions in today assignment (unlearned + expired) at current answering pace.

Session progress: It is a *progress bar* indicator of the student achievement in current session. It is computed as the ratio (percentage) between learned today and total for today

Short, Medium and long term achievement indicators: Three global indicators of your chances of not failing any question in an examination to be taken, respectively, in 20 days (red), 9 months (yellow), or in some years (green).

Performance display for practicing mode



As no performance data is saved, there is no use in displaying the same information than in 'learning' mode. The only useful information for the student is the same than in 'exam' mode:

- number of success / failures
- total score in this session
- practice time
- mean time to answer a question

Information saved

- Problem space data

General information about a problem space. One entry per problem space:

SpaceKey

Id of this problem space

NumQuestions

Total number of questions in the problem space

Repetitions

Number of times a question has to be answered correctly before being promoted

LastUsed

Last DateTime when this space was used

Creation

DateTime when this space was created

TotalRespTime

TimeSpan. Accumulated total time to answer questions (response time)

TotalAsked

Total number of questions asked. To compute average response time

- Problem space sessions

Log of problem space usage. One entry per session:

SpaceKey

Id of the problem space logged

DateUsed

DateTime when this space was used

Duration

TimeSpan session duration

- Questions data

Information about each question. One entry per question.

Hacking details: involved objects

ProblemSpace: It is an object for containing the set of questions for an exercise and level

CountersCtrl: It is a control created by injecting several *ImoXXXXX* objects, to create an area to display user performance statistics

CountersCtrl

|

+--- LeitnerCounters (for 'learning' mode)

|

+--- PractiseCounters (for 'practise' mode)

|

+--- QuizCounters (for 'exam' and 'quiz' modes)

ProblemManager: It is an object with the following responsibilities:

- Chooses a question and takes note of right/wrong user answer
- Load/Saves/Updates the problem space.
- Keep statistics about right/wrong answers
- Owns: the ProblemSpace object

There are two derived objects:

- *LeitnerManager*: It is a problem manager that chooses questions based on the Leitner system, that is, it adapts questions priorities to user needs based on success/failures. It is used for '*learning*' and '*practise*' modes.
- QuizManager: It is a problem manager that generates questions at random. It is used for '*exam*' and '*quiz*' modes.

```

ProblemManager
+--- LeitnerManager
+--- QuizManager
    
```

The different objects involving in managing questions and answers are set in ExerciseCtrl object:

- It chooses a ProblemManager suitable for the exercise mode when invoking create_problem_manager() method. See details in next paragraphs.
- Chooses a CountersCtrl suitable for the exercise mode
- It owns the ProblemManager and the CountersCtrl objects.

ProblemManager creation

ExerciseCtrl owns the ProblemManager and ProblemManager owns the ProblemSpace.

ProblemManager is created in ExerciseCtrl::create_problem_manager(). This method is invoked from:

```

ExerciseCtrl::change_generation_mode()
ExerciseCtrl::create_controls()
    
```

ProblemSpace creation

Once the ProblemManager is created ExerciseCtrl::create_problem_manager() invokes SpecificExercise::set_problem_space() to initialize and load data into the ProblemSpace object.

This method is also indirectly invoked from ExerciseCtrl::create_controls(), near the end, when invoking OnSettingsChanged().

It can be subsequently invoked at:

```

SpecificExercise::OnSettingsChanged()
    
```

SpecificExercise::set_problem_space() invokes one of the following methods:

```
m_pProblemManager->LoadProblemSpace(sKey)  
m_pProblemManager->SetNewSpace(nNumQuestions, sKey);
```

And these methods invoke:

```
m_pProblemManager->OnProblemSpaceChanged()
```

to prepare problem manager to use the new problem space.

CountersCtrol creation

It is created in method `ExerciseCtrol::create_counters_ctrol()`, first invoked from `ExerciseCtrol::CreateCtrols()`. But it can be changed at any moment by invoking `ExerciseCtrol::ChangeCountersCtrol()`. This method is only invoked from `ExerciseCtrol::change_generation_mode()` when the user clicks on the button to select the exercise mode. And from `LeitnerManager`, to change to practise mode

Annexes

Annex 1. Learning mode algorithm

1. Prepare set of questions that need repetition (Set0):

- Explore all questions and move to Set0 all those questions whose scheduled time is \leq Today or are in box 0

2. Start exercise:

- Shuffle Set0 (random ordering).
- While questions in Set0:
 - For $iQ=1$ until $iQ=\text{num questions in Set0}$
 - Take question iQ and ask it
 - If Success, increment the question repetitions counter.
Else reset the question repetitions counter and move question to box 0.
 - If the question repetitions counter is greater than the Repetition Threshold (RT) mark the question to be promoted, and schedule it for repetition at Current date plus Interval Repetition for the box in which the question is classified.
 - End of For loop
- At this point a full round of scheduled questions has taken place: Remove from Set0 all questions marked as 'to be promoted'.
- End of While loop

3. At this point, a successful round of all questions has been made:

- Display message informing about exercise completion for today and about next repetition schedule. If student would like to continue practicing must move to 'Practicing Mode'.