Technical Specification
OpenLogbook

OpenLogbook devteam
developer@openlogbook.org

15th May 2003

Abstract

This specification describes the technical solutions of OpenLogbook software. It provides an overview of the system and describes the basic architecture, different modules, save file structures and other technically oriented aspects of the program. This specification guided the development during the initial project and now serves as an architectural document.

Contents

Version	Date	Author	Description
1.0	2.12.2002	Tuomas	First release
1.1	8.2.2003	Henri, Juho, Niklas, Tuomas	Updated module and architecture descriptions.
1.2	21.3.2003	Tuomas	Updated to describe the current situation
1.3	14.4.2003	Tuomas	Updated for final review

Table 1: Changelog

1 Introduction

1.1 Purpose and Scope

This specification describes the structure of the OpenLogbook program. It describes modules, their public interfaces and the overall communication between them. It also defines the file formats to be used with saved projects.

This specification was meant for the internal use of the project group during the project. It describes the essential features (e.g. interfaces) in such a detail that developers could use it for implementing modules. Technically oriented customer(s) could also obtain information about the basic structure of the program from this specification. After the ending of the project this document serves as an architectural document for those who might need it.

This document describes the features that were implemented during the initial project in 2002 - 2003. For example the networking modules are mentioned but neither described nor even properly designed because they were not to be implemented during the project. The structure is however designed so that those modules can be added easily. In the future this document should be updated as necessary.

1.2 Product and Environment

The name of the product is OpenLogbook. For more detailed information about the product see OpenLogbook Project Plan [2] and OpenLogbook Requirements Specification [4].

OpenLogbook must runs on PC and at least on Windows and Linux platforms. Other platforms may also work if they have proper components i.e. JVM and JMF installed.

1.3 Definitions, Terms and Abbreviations

Definitions, Terms and Abbreviations are described in a separate document [5].

1.4 Overview of this Document

The rest of this document describes elaborately the technical solutions of OpenLogbook. Section 2 presents an overview of the System. Section 3 describes the large scale architecture and communication between the modules. Section 4 gives more detailed descriptions of the modules and their public interfaces. Section 5 describes possible error conditions and how they are handled inside the software and when and how to inform the user about them. Section 6 presents some future scenarios for OpenLogbook.

2 Overview of the System

2.1 Application domain

The application domain is described in OpenLogbook Project Plan [2].

2.2 Hardware and Software Environment

These have been fully explained in the User Requirements Document [4].

2.3 Development Environment

Development was done mainly in Linux environment. However, some testing was carried out on Windows to ensure that OpenLogbook will run on Windows as well. If a developer had wanted he might have used Windows as a development environment, too.

The programming language used is Java 1.4. The development environment must therefore include Java 1.4 compiler and bytecode interpreter i.e. Sun Java 1.4 SDK. For the streaming media content JMF 2.1.1 is required.

Other tools needed in the development environment include CVS, Ant and JUnit. CVS is a program for version controlling [6]. Ant is a tool for automating the compilation of the source code [7]. JUnit is a library that helps in writing unit tests [8].

The usage of editors or other development tools was not restricted and the choice was up to the developers. During the initial project the most popular tool was Emacs.

2.4 Limitations

The customer required that all the documentation had to be written in English during the initial project. This applied to the code and comments as well so that the code would be easily understood by an english speaker (i.e. names of variables and methods were derived from English words). English was a natural choice because the future development group was unknown.

No other limitations were set by the customer or by any other participant.

2.5 Agreements and Standards

The development was done under the directions of the OpenLogbook Quality Manual [3].

3 Description of the Architecture

3.1 Design Principles

The leading design principles are modularity and clarity. The system is divided to explicit modules, each of which handles a specific job. We are trying to keep modules as simple as possible and to make them perform their tasks as well as possible. Using modules also enables us to develop them individually.

The interfaces of the modules are designed to be clear and as stable as possible. This enables us to make changes to modules without having any influence to other modules. It is probably impossible to keep the interfaces totally unchanged all the time, but we have tried to avoid changes by giving attention also to features that were not implemented during the project.

Simple modules and clear interfaces should result in clarity of the whole design. Clarity is especially important after the ending of the project to help the developers to come. Clarity makes it possible for new developers to internalise the essentials of the system fast and get them to work quickly.

Since we are using Java the object orientation comes automatically with it. We do not actually consider it as a design principle but rather as a necessity which naturally affects every aspect of the software - design, implementation and so on.

3.2 Save Project Structure

In OpenLogbook program a project means a structure that can contain audio, video and different log files. A project can then be played or edited (add files, remove files, change synchronization etc.) with OpenLogbook.

A saved project is a kind of a database. The files of the project are handled as individual files and can be stored anywhere (e.g. hard disk, web server or a dedicated OpenLogbook server) but the contents of each project is structured. The structure of the project is described in a project.xml [1] file. Another important save file is the hotspot file. Hotspots are saved individually even though they are an important part of every project. This makes it possible to handle the hotspots more flexibly which is necessary because hotspots do change a lot more often than other project data. Saving the whole project would be waste of resources if for example only one hotspot has changed.

3.2.1 Project XML File Structure

The heart of a save file is the project.xml file that contains information about every other piece of data contained in the project. The XML file tells where each data can be found and how it should be handled. The information about hotspots is saved on another file which has a file name related to a project.xml. More detailed description can be found in the section 4.3.

In the beginning of the project.xml file there is an experiment ID which tells the exact time when the OpenLogbook project was created. This ID is considered as a unique identifier. After that there is a description of the project and the creator of the project.

After these compulsory fields there can be a varying set of different kinds of other data files. There may be video, audio and text log files. Format of the audio and video files is not important as long as JMF knows how to handle them. The text logs' format can also vary and it is the OpenLogbook's data module that has to know how to read them. Each node tells the location and the time synchronization offset. Offset tells when the presenting of the file should be started relative to the beginning of the project. For text logs there is also a compulsory attribute that tells which wrapper should be used for that data.

As mentionded before, in another file there are hotspots, which are time stamps of interesting parts of the project. They have their own file and file format, which also uses XML. Hotspot file also has ID and after this there are hotspots. Each hotspot contains the relative time to the start of the project and a description of the phenomenon.

Appendix 7.2 shows the element declarations and an example project.xml file. Appendix 7.3 shows the element declarations and an example file for hotspots.

3.3 Time model

The time model of OpenLogbook is derived from the time model of JMF. JMF handles the time in nanoseconds and so does OpenLogbook. Thus, internally every time must be in nanoseconds. User interface has to do necessary time conversions to represent the time in reasonable accuracy to users.

Inside a project the time starts from zero. Time zero is the time from which the OpenLogbook will start to play projects. The time offsets of medias and different log files are relative to the project's time. For example if an offset of a video is 1 000 000

[1] In reality not necessarily this exact name, however project.xml is used for clarity in this specification.

000 (= 1 second), it means that the video will start to play one seconds after the playing of the whole project started. Also negative offsets are allowed.

Inside the differet log files time also starts from zero and the times inside the file are relative to the time of that particular file. The offset, which is defined in the project.xml, is then used to synchronize the log file and the events to the project time.

All medias, except hotspots, can have the offset. Hotspots are concidered as an inseparable part of each project. This means that the times of individual hotspots are directly relative to project time.

3.4 Architecture, Modules and Communication

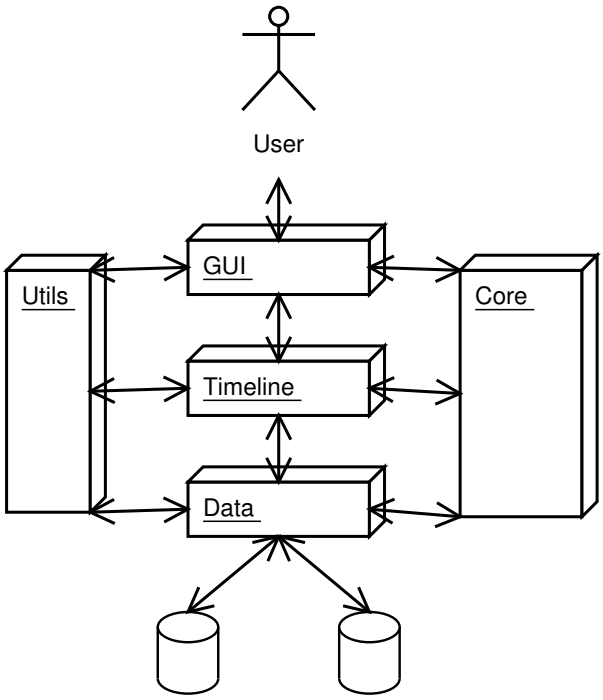

Figure 1: Module overview and communications

Main modules of the OpenLogbook are Core, Timeline, Data, Util and GUI. The architecture can be seen as a three layer structure shown in figure 1. This section presents an overview of the modules and the communication between them. A more detailed module descriptions are available in section 4.

The GUI module functions as an interface to the user. It handles the data representation and interprets the user's commands to the program. It doesn't do any actual data processing - just the representation.

Timeline functions as the middle layer that processes the data and handles the synchronization. It communicates constantly with the GUI which presents the data. It also has a full access to the data on the underlying Data module.

Data module handles the communication with the hardware. It does the XML-parsing and takes care of actual saving and loading actions. Data can be located in multiple places. In figure 1 there are only 2 data storages but there may be more and those storages can be different kinds of devices as described in section 3.2. Data can even be read through network.

Core and Util are supporting modules to the others. Core contains classes and methods that are essential for the program like logging and main Exception classes. It also handles the starting of the program including parsing arguments, loading properties and opening the GUI. Util on the other hand contains a set of common classes and methods that may be needed in different classes or do not fit very well to anywhere else.

Communication between the modules is mainly handled by calling public interface classes. Figure 1 shows which modules communicate with each other. Communication between the GUI and the timeline is done with different kinds of events that tell the GUI what to do. For example the GUI components that show the text data are registered to a specific controller which then sends events to control the GUI component.

4 Module Descriptions

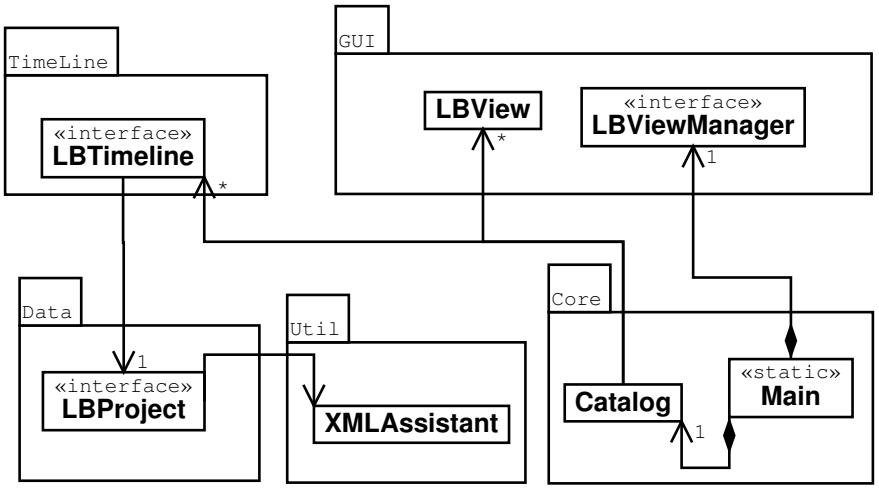

Figure 2: Modules and their relationships

4.1 Core

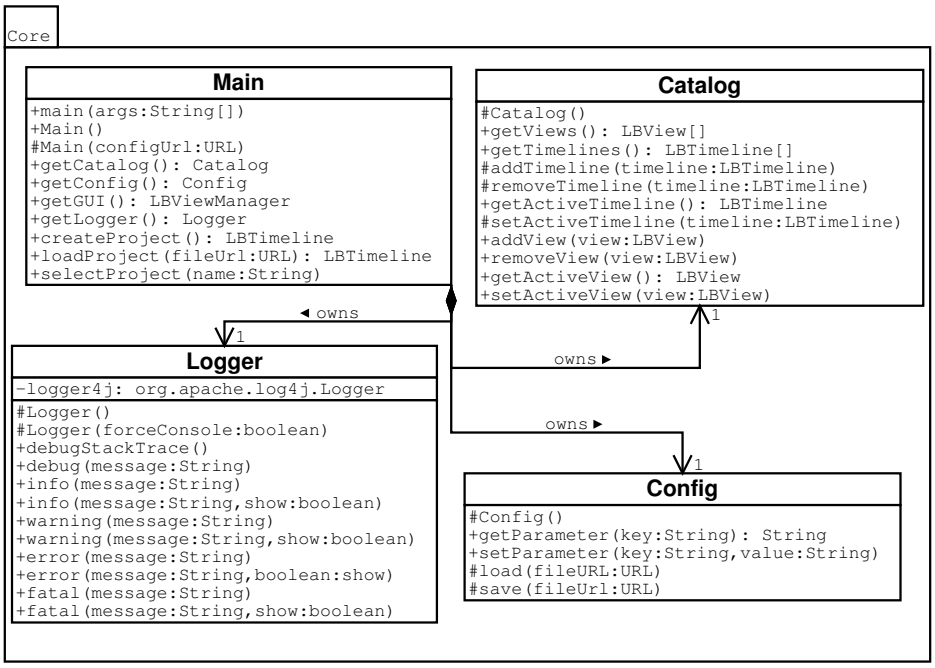

Figure 3: Core module class diagram

4.1.1 Description

Core module is the center of the whole program. Everything begins from here when the user calls the main function of the program. Command line arguments are parsed and basic services like logging and configuration management are started. These services are then provided to the other modules through public interfaces. Core module also calls initialization of other modules, e.g. GUI, and stores instances of them. Core's class diagram is shown in figure 3.

4.1.2 Interfaces

Core stands between the other modules. Every module can ask for an instance of some other module through Main class's corresponding methods. There are methods for accessing GUI's ViewManager and other public system wide instances like Logger, Config and Catalog.

Creation or loading of projects goes through the Core module. The request is forwarded straight to the Timeline module which returns an instance of the Timeline after all related files are loaded and initialized from the disk. This class is returned to the user interface which shows the content of the project for the user. After initialization, user interface and Timeline-module can communicate directly but this conversation always starts within the control of the Core module.

Main class contains public get methods for program wide instances. Project handling methods are also present. The class and it's methods are static so that other

modules can call them direcly. It contains the main-function of the program.

Logger class has public methods for adding different kinds of log and error messages. These methods must be called from other modules when there is a need for logging. No *System.out.println()* -methods are allowed for logging purposes. It is also possible to show all log messages to the user inside pop-up windows.

Config class has an interface for accessing configurations. Values can be retrieved or inserted using key names found from *openlogbook.properties*-file. Null value is returned if no matching key name was found. Configuration information is read from the disk when the program starts and written back before quitting the software.

Catalog class keeps track of all open components. It has an interface for accessing all Timelines and their graphics components. Currently active items can be requested too. Adding of Timelines is handled only by the Main class and the Views are handled by the ViewManager, so they are not intended for public usage.

Command line parameters

- -config location of the configuration file in URL syntax.

4.1.3 Structure and Implementation

The Main class is static so that it is accessible from anywhere in the program.

Logging is implemented using the assistance of the Log4j external package. Our implementation of Logger class is only a wrapper to that package, so the logger implementation can be easily replaced at a later time if needed.

Configurations are saved using a syntax compatible with Java's Properties class.

4.1.4 Exception handling

The module has to be resistant to incorrect arguments and configurations provided by the user. There have to be some functional default values in a case of missing configurations.

The module has to handle all possible errors inside the module so that the main program never throws any exceptions to the underlying Java virtual machine.

4.2 Timeline

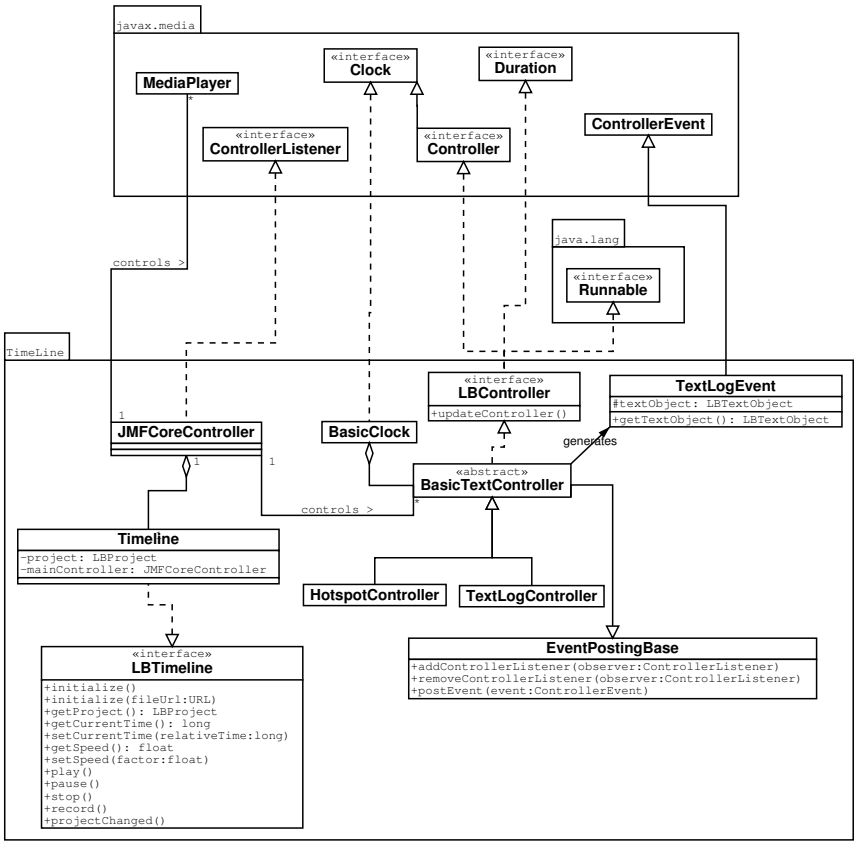

Figure 4: Timeline module class diagram

4.2.1 Description

Timeline module controls the time and media content of the OpenLogbook software. It stands between the data and the presentation layers. It depends heavily on JMF for synchronizing media files. It extends the functionality of JMF to represent the textlog files and to control all media representations. The content of the timeline is played to the user using the presentation layer, GUI module. This playing is done by controlling of medias and sending action messages to the GUI.

4.2.2 Interfaces

LBTimeline is a public interface for controlling the entire project. There are methods for creating a new project either by loading it from a project file or by starting from scratch. In case of an existing project, all data are loaded using assistance of the Data module and then the created instance of the Timeline is returned.

After creation of the timeline and initialization of the GUI views, the project is ready to be played. There are several control methods for different types of operations

like play, record, pause and stop. These control methods don't return anything, they just fire the event.

The project instance which contains all the data of the project can be returned using *getProject()* method. Timeline itself doesn't handle the data, it uses the Data module for that. See section 4.3 for more details about data storage.

LBController is a public interface that extends JMS's **Controller** interface. A separate interface is needed for the data module so that the controller can be updated when the data changes.

4.2.3 Structure and Implementation

Implementation of the timeline is based heavily on the JMF. Time controlling is done by JMF which takes care of all media files, too. The timeline consist of two parts: the components controlling the whole project and the components that handle the text playing. There are also several classes imoprtant to both of these parts, for example the *BasicClock* that takes care the low level time handling.

Backbone implementation of timeline consists of *BasicClock* and *BasicTextController* classes. They implement the basic functionality of Clock and Controller interfaces. The *BasicTextController* is extended by other controllers to get proper functionality for different types of text files. Also other kinds of medias can extend the *BasicTextController* if the data can be represtented as text and if there are necessary gui components availabe to present the data. For example a contoller for pictures could be created easily.

JMFCoreController is the controlling component for all other contorllers and players. It does not have any media of its own to play, it just controls all the others. The synchronization of different media happens automatically when the other players are set as controlled by the *JMFCoreController*. The functionality of *JMFCoreController* is hidden behind the public interface of the timeline. *LBTimeline* interface can handle all the necessary actions pointed to all of the medias.

The synchronization of text logs is done with different controller classes inherited from the *BasicTextController*. These controllers implement the *Controller* interface of JMF and act just as any other controller, even though the data is not traditional streaming data like audio or video. The contollers keeps track of text logs' time and fires an event when it's time to change the data that is shown to user. The data is sent as a *TextLogEvent* to the user interface which shows it on the screen.

Each text log needs an individual controller instance. These instances are connected to each other and when the data changes the conrtoller is updated automatically.

4.2.4 Exception handling

The module handles unexpected exceptions caused by time handling and/or user's actions itself. In case of severe errors which abort the execution, the cause of the error is shown to the user. For logging and user messages the module uses the *Logger* from the Core module.

4.3 Data

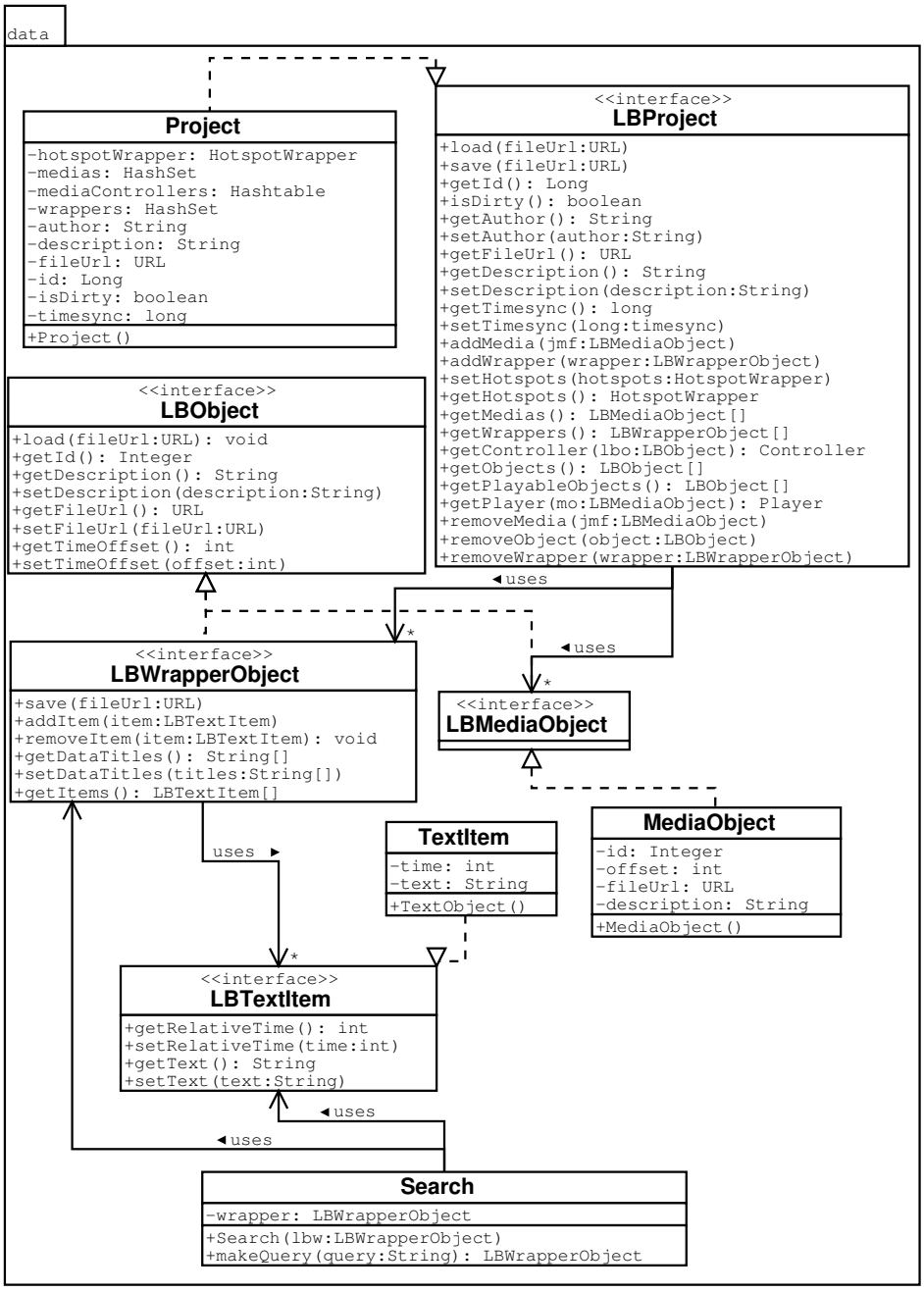

Figure 5: Data module class diagram

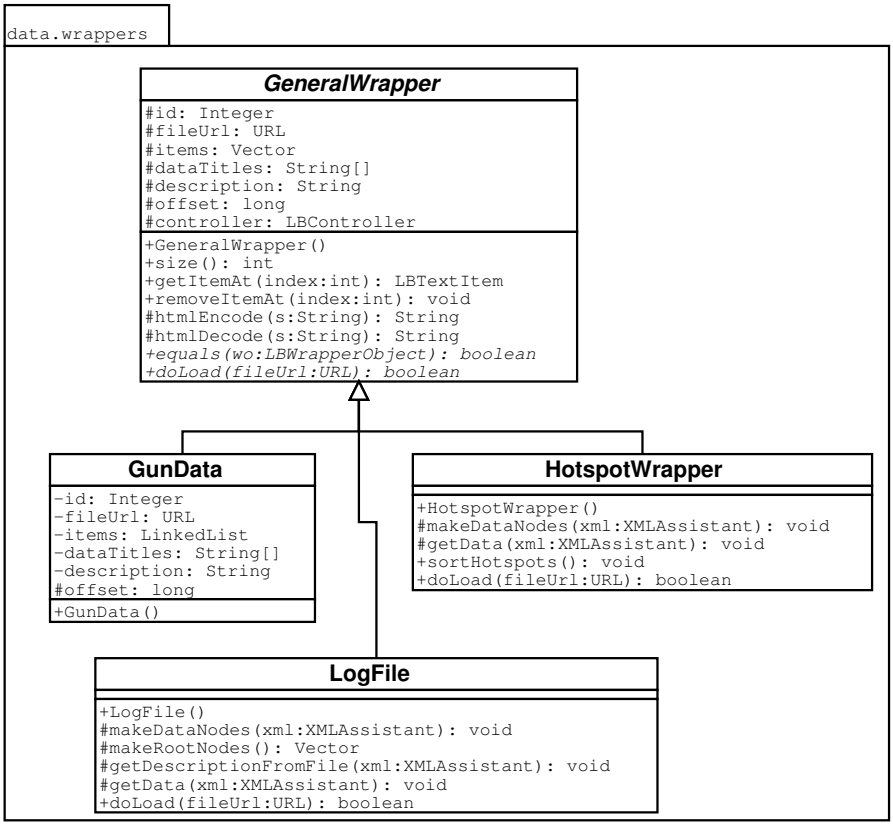

Figure 6: Wrappers package class diagram

4.3.1 Description

Data module handles all storage needs for the OpenLogbook. There is a Project class which can be saved into an XML-file. It contains information about all other data files in the project. These data can be either media files or text/xml based log files. OpenLogbook supports various kinds of text logs by having one interface (LBWrapperObject) to all of them. There can be many different implementations but they are all the same from the software point of view. Thus only one implementation of Search-class is needed for making search-queries to all wrappers. The different wrapper-implementations are located in data's sub-package called wrappers (fi.hip.openlogbook.data.wrappers).

Writing third party wrappers is simple. The developer has to implement the LB-WrapperObject interface and add the wrapper to the load dialog. There is an abstract class called GeneralWrapper that can be extended if a straight implementation of the interface is not escpecially necessary. When extending the GeneralWrapper the developer has to take care only about loading, saving and comparing this type of wrappers with each other. Thus this is recomended instead of implementing the wrapper interface directly.

Data module also creates the controllers that the timeline module needs and keeps them uptodate when data changes.

4.3.2 Interfaces

LBProject contains all information related to the project. A project can be saved and loaded from disk, where it's in an XML file (see 7.2 for the file structure). It contains methods for getting a list of included media files and log files, adding new items to the project and setting and getting the hotspots. Also methods for getting the necessary JMF components for timeline are included.

When a project is loaded, an instance of LBProject is created using the information in the XML file. Then included files are loaded using the assistance of LBWrapperObject and LBMediaObject interfaces. Hotspots are loaded using the HotspotWrapper-class. They are saved in a xml-file called *<projectfilename>.<hotspotpostfix>.xml* (e.g. *project.hotspots.xml*). The postfix can be changed from the *openlogbook.properties* -file. The structure of the xml-file can be seen at appendices (7.3).

LBObject is an interface that is extended by all the other OpenLogbook's Object-interfaces. It includes methods for setting and getting the unique Id for an object, it's description, file url and time offset.

LBWrapperObject is a wrapper interface to different text formats. Wrappers can contain unlimited amount of timestamped messages, stored using the LBTextItem interface. There are methods for returning, setting and adding items to the wrapper.

When a new log file is to be loaded, the Project class creates a new instance of the wrapper using the class name mentioned in the project file. Then *load(fileUrl)* method is executed with the URL to the logfile. After this the wrapper is on its own and must implement all necessary routines to read the content of the file and fill up the data fields.

LBWrapperObject contains two-dimensional tabular data. There is a set of data titles (columns) and an unlimited amount of values for them (rows). Every row has a timestamp. Using this kind of syntax it is possible to have a general log viewer for showing any kind of log files. There is no need to have specialized viewers for every log type.

getDataTitles() is an interface for retrieving the array of titles and *getItems()* is used for getting the array actual data items (LBTextItem).

Search is a class that is used for searching information from a wrapper. It includes one public method *makeQuery(String query)* for making a search to a wrapper given in a constructor. The method returns a new wrapper with a same type but only with the data that passed the query. The query can include both keyword- and comparing-searchs (greater than, lower than, etc..) with AND/OR/NOT operators.

Currently, OpenLogbook supports 3 different types of wrappers: HotspotWrapper, LogFile and GunData. HotspotWrapper is used for saving the project's hotspots, LogFile is a wrapper for OpenLogbook logfiles (see 7.4 for the structure of the file) and GunData is used for reading Gun microscope data.

LBTextItem is a container for time and data. Time is a relative time in nanoseconds and the data is an array of plain Strings, one string for each column. The interface contains *getRelativeTime()* and *getTexts()* methods for accessing these values, as well as corresponding setter methods.

There is a same kind of an interface for media files, a **LBMediaObject**. Instead of a text message it contains a URL to the location of the media file. The relative time is replaced with an offset to when the media should start playing on timeline.

4.3.3 Structure and Implementation

Data package is only a bunch of interfaces and as many implementations for them as needed. Project is not dependent on any other wrapper than HotspotWrapper, which ensures the scalability of the data module in the future.

Wrapper-implementations are located in their own package, data.wrappers. This is the place where the third party wrapper should be located, too.

4.3.4 Exception handling

There can be many kinds of errors during reading, parsing and transforming different data types. It is important that these errors don't prohibit the usage of the entire software. They must give clean error messages to the caller explaining what went wrong and why. The review of the entire project must not be aborted on errors, only problematic data items are left out.

4.4 GUI

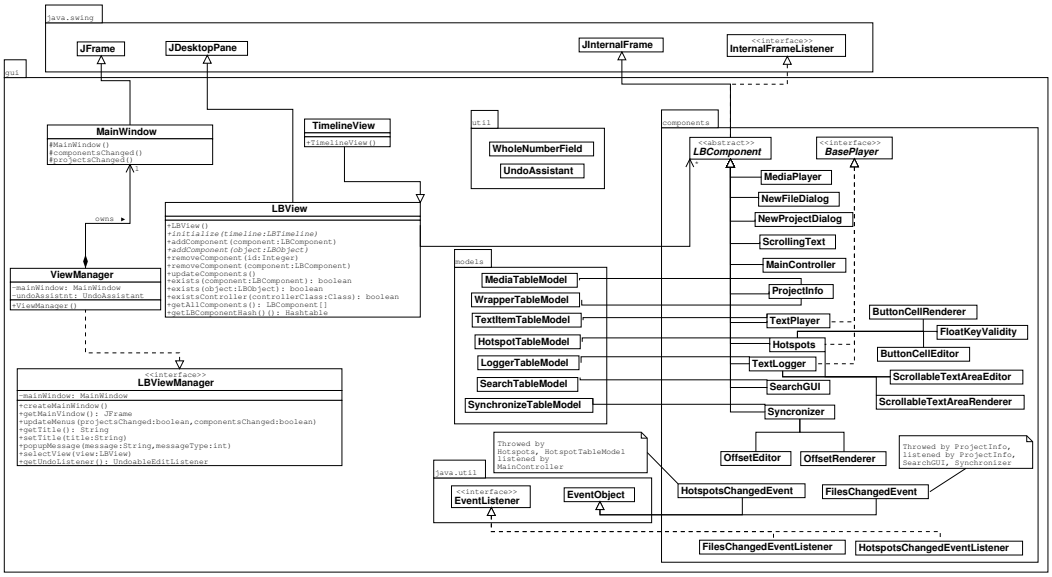

Figure 7: GUI module class diagram

4.4.1 Description

Graphical user interface module is the presentation layer of the software. There should not be any external logic nor data handling inside this module. Timeline module controls the time flow and informs GUI with events when something has to be done.

GUI is constructed from one ViewManager, several views and multiple components inside every view. Only one view is visible at the same time. Each instance of Project class has its own view, currently implemented by TimelineView class. Core module's Catalog class takes care of that the right Project's TimelineView is visible at the right time.

4.4.2 Interfaces

The main interface to the GUI is **LBViewManager**. It handles the main window of the software. There are public methods for creating and getting instances of MainWindow, updating menus inside MainWindows, selecting views and getting the undo listener common for all components. All information messages to the user are sent through the *popupMessage(String message, int messageType)* method. The title of the main window can also be set using *setTitle(String title)* method.

MainWindow is the main frame of the GUI. It interacts with the user through common menus and toolbars, and forwards users actions to the right modules. The main window itself is accessed only by the ViewManager.

LBView is an abstract class that extends JDeskTopPane. LBView offers methods for adding and removing components shown in the view. LBView also takes care of the singleton controllers(Hotspots, MainController, Synchronizer and ProjectInfo). LBView is extended by **TimelineView** which knows how to create visual components out of LBObjects.

LBComponent (gui.components.LBComponent) presents a component inside view. There can be many different types of derived components for different data presentations and visualizations. All subcomponents implement LBComponent and therefore extend JInternalFrame and implement InternalFrameListener. Interfaces for components are simple, only one method for getting the ID of the component.

If component is used to present JMF related data, it must implement *javax.media.ControllerListener* interface. After that JMF communicates with the component sending events to *controllerUpdate(ControllerEvent)* method.

4.4.3 Structure and Implementation

Creating new components must be easy, without need to recompile the whole program. This means having either a list of classes in the config file or autonomous listing based on classes found in the classpath.

All views are stored inside Core package's Catalog class. Views are always bound only to one Timeline object. This is done by the Catalog class. Every view has to store in itself all the components it owns. Components are thus bound only to one view.

4.4.4 Exception handling

Errors in the user interface may not affect other parts of the software. If very severe faults occur and the whole UI crashes, the cause must be printed on the console and to the log file.

4.5 Util

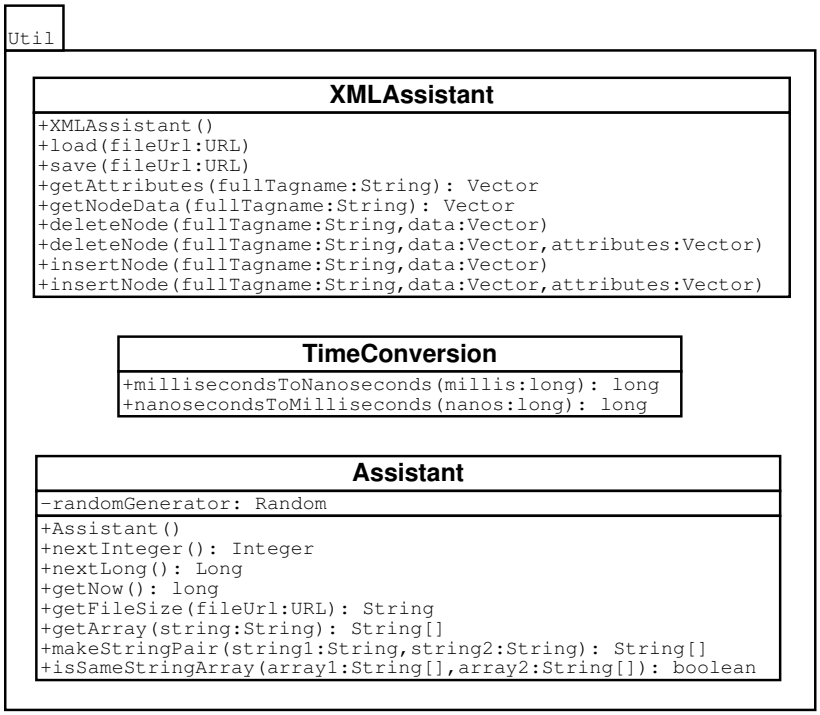

Figure 8: Util module class diagram

4.5.1 Description

Util module has common helper classes used by other modules.

4.5.2 Interfaces

XMLAssistant provides an easy way for reading and writing XML files. After loading an XML-file to the class it can be inspected using methods for getting node value and it's attributes in a Vector. The class also provides methods for inserting and deleting nodes and finally saving the XML-tree back to a file.

TimeConversion provides simple methods for making time conversions.

Assistant includes all kind of general methods. It provides methods for retrieving random numbers as integers or longs and getting the size of a file behing an URL-address. It also includes some methods for making and comparing of String-arrays.

4.5.3 Structure and Implementation

Util module is only a bunch of invidual classes. They are not dependent on each others.

5 General Error Handling

All errors and exceptions must be caught inside the module where the fault happened. There is a much better possibility of identifying and solving the problem if it is caught near the problem. This also keeps modules as autonomous as possible without throwing exceptions to each other and confusing the design and implementation.

If the exception affects or prevents actions that user wanted to accomplish, a pop-up window may be used to inform the user. Core's Logger class has methods for showing log messages on the screen.

5.1 Logging

For logging the *Logger* class from Core module is used. The logger has five levels of messages and each of them should be used as follows:

- **fatal** level should be used when the program cannot run anymore or is likely to crash.
- **error** level should be used when important actions cannot be performed.
- **warning** level should be used when something happens that is not expected and/or may (but not yet has) cause an error.
- **info** level should be used for informational messages.
- **debug** level should be used for debug messages.

All these messages except debug level messages can be showed to the user with a pop-up window.

Each of these log messages are generated using a corresponding method in Logger. See Javadoc documentation generated from source code files for detailed information about instantiating and using the Logger.

In general there should rather be too much logging than too little (of course on an apporopriate level).

6 Future Development Plans

The plans of the future are represented in the Status Report of the last project phase [10].

References

[1] OpenLogbook project webpages for the course T-76.115
http://motha.tky.hut.fi/openlogbook/
Cited: 13.4.2003.

[2] OpenLogbook Project Plan
http://motha.tky.hut.fi/openlogbook/delivery/project_plan.pdf
Cited: 13.4.2003.

[3] OpenLogbook Quality Manual
http://motha.tky.hut.fi/openlogbook/delivery/quality_manual.pdf
Cited: 13.4.2003.

[4] OpenLogbook Requirements Specification
http://motha.tky.hut.fi/openlogbook/delivery/requirements.pdf
Cited: 13.4.2003.

[5] OpenLogbook Terminology document
http://motha.tky.hut.fi/openlogbook/delivery/terminology_doc.pdf
Cited: 13.4.2003.

[6] OpenLogbook method description: Concurrent Versions System and Software Configuration Management
http://motha.tky.hut.fi/openlogbook/methods/cvs_and_scm/
Cited: 13.4.2003.

[7] Apache Ant homepage
http://jakarta.apache.org/ant/
Cited: 13.4.2003.

[8] OpenLogbook method description: Automated unit testing with JUnit
http://motha.tky.hut.fi/openlogbook/methods/unit_testing/
Cited: 13.4.2003.

[9] JMF: Supported Media Formats and Capture Devices
http://java.sun.com/products/java-media/jmf/2.1.1/formats.html
Cited: 13.4.2003.

[10] OpenLogbook - Status report
http://motha.tky.hut.fi/openlogbook/delivery/status_report.pdf
Cited: 13.4.2003.

7 Appendices

7.1 File formats supported by JMF

According to [9] JMF supports the following file formats:

- `D` indicates the format can be decoded and presented.
- `E` indicates the media stream can be encoded in the format.
- `read` indicates the media type can be used as input (read from a file)
- `write` indicates the media type can be generated as output (written to a file)

Media Type	JMF 2.1.1 Cross Platform Version	JMF 2.1.1 Solaris Performance Pack	JMF 2.1.1 Windows Performance Pack
AIFF (.aiff)	read/write	read/write	read/write
8-bit mono/stereo linear	D,E	D,E	D,E
16-bit mono/stereo linear	D,E	D,E	D,E
G.711 (U-law)	D,E	D,E	D,E
A-law	D	D	D
IMA4 ADPCM	D,E	D,E	D,E
AVI (.avi)	read/write	read/write	read/write
Audio: 8-bit mono/stereo linear	D,E	D,E	D,E
Audio: 16-bit mono/stereo linear	D,E	D,E	D,E
Audio: DVI ADPCM compressed	D,E	D,E	D,E
Audio: G.711 (U-law)	D,E	D,E	D,E
Audio: A-law	D	D	D
Audio: GSM mono	D,E	D,E	D,E
Audio: ACM	-	-	D,E
Video: Cinepak	D	D,E	D
Video: JPEG (411, 422, 111)	D	D,E	D,E
Video: RGB	D,E	D,E	D,E
Video: YUV	D,E	D,E	D,E
Video: VCM	-	-	D,E
Flash (.swf, .spl)	read only	read only	read only
Macromedia Flash 2	D	D	D
GSM (.gsm)	read/write	read/write	read/write
GSM mono audio	D,E	D,E	D,E
HotMedia (.mvr)	read only	read only	read only
IBM HotMedia	D	D	D
MIDI (.mid)	read only	read only	read only
Type 1 and 2 MIDI	-	D	D

Media Type	JMF 2.1.1 Cross Platform Version	JMF 2.1.1 Solaris Performance Pack	JMF 2.1.1 Windows Performance Pack
MPEG-1 Video (.mpg)	-	read only	read only
Multiplexed System stream	-	D	D
Video-only stream	-	D	D
MPEG Layer II Audio (.mp2)	read only	read/write	read/write
MPEG layer 1, 2 audio	D	D,E	D,E
MPEG Layer III Audio (.mp3)	-	-	read only
MPEG layer 1, 2 or 3 audio	-	-	D (through DirectX)
QuickTime (.mov)	read/write	read/write	read/write
Audio: 8 bits mono/stereo linear	D,E	D,E	D,E
Audio: 16 bits mono/stereo linear	D,E	D,E	D,E
Audio: G.711 (U-law)	D,E	D,E	D,E
Audio: A-law	D	D	D
Audio: GSM mono	D,E	D,E	D,E
Audio: IMA4 ADPCM	D,E	D,E	D,E
Video: Cinepak	D	D,E	D
Video: H.261	-	D	D
Video: H.263	D	D,E	D,E
Video: JPEG (411, 422, 111)	D	D,E	D,E
Video: RGB	D,E	D,E	D,E
Sun Audio (.au)	read/write	read/write	read/write
8 bits mono/stereo linear	D,E	D,E	D,E
16 bits mono/stereo linear	D,E	D,E	D,E
G.711 (U-law)	D,E	D,E	D,E
A-law	D	D	D
Wave (.wav)	read/write	read/write	read/write
8-bit mono/stereo linear	D,E	D,E	D,E
16-bit mono/stereo linear	D,E	D,E	D,E
G.711 (U-law)	D,E	D,E	D,E
A-law	D	D	D
GSM mono	D,E	D,E	D,E
DVI ADPCM	D,E	D,E	D,E
MS ADPCM	D	D	D
ACM	-	-	D,E

Table 2: File formats supported by JMF

7.2 XML representation of project data

```
<?xml version=''1.0'' encoding=''ISO-8859-1'' ?>
<!DOCTYPE project [
 <!ELEMENT project (description?,creator?,media*,wrapper*)>
 <!ELEMENT timesync (#PCDATA)>
 <!ELEMENT description (#PCDATA)>
 <!ELEMENT creator (#PCDATA)>
 <!ELEMENT media (file,timesync?,description?)>
 <!ELEMENT wrapper (file,timesync?,description?)>
 <!ELEMENT file (#PCDATA)>
 <!ELEMENT timesync (#PCDATA)>
 <!ATTLIST project id CDATA #REQUIRED>
 <!ATTLIST wrapper class CDATA #REQUIRED>
]>

<project id=''1044605170157''>
  <description>This is an example project</description>
  <creator>Someone@somewhere</creator>
  <media>
    <file>file:video.avi</file>
    <timesync>1200</timesync>
    <description>Laboratory view</description>
  </media>
  <media>
    <file>file:sem.avi</file>
    <timesync>-500</timesync>
  </media>
  <log wrapper='fi.hip.openlogbook.data.wrappers.GunData'>
    <file>file:/tmp/gundata.txt</file>
    <timesync>-500</timesync>
    <description>Comments written during the experiment</description>
  </log>
  <log wrapper='fi.hip.openlogbook.data.wrappers.LogFile'>
    <file>http://www.hip.fi/experiment/test.txt</file>
  </log>
</project>
```

7.3 XML representation of hotspot data

```
<?xml version=''1.0'' encoding=''ISO-8859-1'' ?>
<!DOCTYPE hotspots [
 <!ELEMENT hotspots (hotspot*)>
 <!ELEMENT hotspot (#PCDATA)>
 <!ATTLIST hotspots id CDATA #REQUIRED>
 <!ATTLIST hotspot timesync CDATA #REQUIRED>
]>
```

```
<hotspots id=''176941689''>
  <hotspot timesync=''3212''>the first interesting thing</hotspot>
  <hotspot timesync=''55344''>look at the temperature!</hotspot>
</hotspots>
```

7.4 XML representation of logfile data

```
<?xml version=''1.0'' encoding=''ISO-8859-1'' ?>
<!DOCTYPE logfile [
 <!ELEMENT logfile (description?,data?)>
 <!ELEMENT description (#PCDATA)>
 <!ELEMENT data (logline*)>
 <!ELEMENT logline (#PCDATA)>
 <!ATTLIST logfile id CDATA #REQUIRED>
 <!ATTLIST logline timesync CDATA #REQUIRED>
]>

<logfile id=''107959875''>
  <description>OpenLogbook logfile for experiment #112</description>
  <data>
    <logline timesync=''1333''>First thing</logline>
    <logline timesync=''44423''>The other</logline>
  </data>
</logfile>
```