



Graphical User Interface for Conditions Database

Phillip Hamnett, University of Manchester, United Kingdom

September 8, 2011

Abstract

Here put a short abstract of what one can find in this document

Contents

1	Introduction	4
2	International Linear Collider	5
2.1	Purpose	5
2.2	ILC in Numbers	5
2.3	Basic Design of the ILC	6
3	International Large Detector	7
3.1	Main Components of ILD	7
3.1.1	Time Projection Chamber	7
4	Conditions Data	8
4.1	The Conditions Database	8
4.2	Errors in Conditions Data	9
4.3	Database Design and Object Structure	9
5	Database Viewer	12
5.1	Introduction	12
5.2	Working Example	12
5.3	Code Structure	12
5.3.1	Using Qt	12
5.4	Connections	13
5.5	Class Hierarchy	14
5.5.1	Class: mainWindow	14
5.5.2	Class: menuBar	15
5.5.3	Class: objectControl	17
5.5.4	Class: objectGrid	20
5.5.5	Class: dbObject	23
5.5.6	Class: objectRect	25
5.5.7	Class: tagWindow	27
5.5.8	Class: taggingSystem	28
5.5.9	Class: dbHierarchy	29
5.5.10	Class: dbTreeView	31
5.6	Next Steps for dbViewer	31
5.6.1	Known Bugs	31
5.6.2	Unknown Bugs	32
5.6.3	Missing Functionality	32
5.6.4	Additional Features	32
6	Summary	33

List of Figures

1	International Linear Collider	6
2	International Large Detector	8
3	Time Projection Chamber	9
4	Database Access	10
5	Database Structure	10
6	Structure of m_allObjects	11
7	Layer and Tagged Object Example	11
8	Class Hierarchy	15

List of Tables

1	Table of Connections	34
---	--------------------------------	----

1 Introduction

For 8 weeks, a program was written called dbViewer. This program is designed to allow the user to view a conditions database stored in a MySQL database system. It is a Graphical User Interface (GUI) that was written using the Qt libraries. The rest of the code is written in C++, with the addition of the LCCD (Linear Collider Conditions Database) libraries that were developed by the software group of the ILC. This document will consist of a brief introduction to the International Linear Collider and International Large Detector, followed by a discussion about what conditions data is, and how the conditions database is arranged. Finally, the remainder of the document will focus on the program I have written and describe all of its functionality for a developer. I have not discussed each function line by line, as the code is well commented whenever the need arises. I will try to make the program as easy to visualise as is possible, using Figures and Tables to explain the more difficult concepts appropriately.

2 International Linear Collider

The International Linear Collider (ILC) is a proposed linear collider that will collide particles of electrons and positrons at centre of mass energies between 200-500 GeV, with a upgrade option to reach energies of up to 1 TeV.

2.1 Purpose

The ILC is designed to take precise measurements of new physics and has the potential for discovering new physics. The precision of the ILC is would be much larger than that of a circular hadronic detector such as the LHC because the initial state of the particles is known precisely for an electron positron collider (compared to the LHC where the initial parton states at the moment of collision are unknown). This means that you get much cleaner results and therefore more accuracy in your measurements.

2.2 ILC in Numbers

For fun, here are some figures for the ILC:

- Collisions
 - Electron/Positron bunches of 5 nanometers height.
 - Each containing 20×10^9 particles.
 - Colliding 14×10^3 times a second.
- Energy
 - 250 GeV per electron or positron.
 - Therefore $\sqrt{s} = 500 GeV$
 - Option to Upgrade to $\sqrt{s} = 1 TeV$
- Approximately 31 km in length.
- 2 Detectors.
- 300 Laboratories.
- At least 1600 People working on design.
- Total cost of \$6.62 Billion¹

¹This estimate is taken from 2007, does not include labour costs, and is not incredibly specific about what it does include.

2.3 Basic Design of the ILC

The electrons are created at a source and accelerated into a damping ring. They are accelerated around the detector and are used at 150 GeV to create positrons at the positron source. The electrons and positrons are both stored in a damping ring before being accelerated down the main linear accelerator. They cross in the middle of this accelerator at a small angle (so that the particles don't go down each others beamlines) and then collision is detected by two detectors operating on a push-pull system.

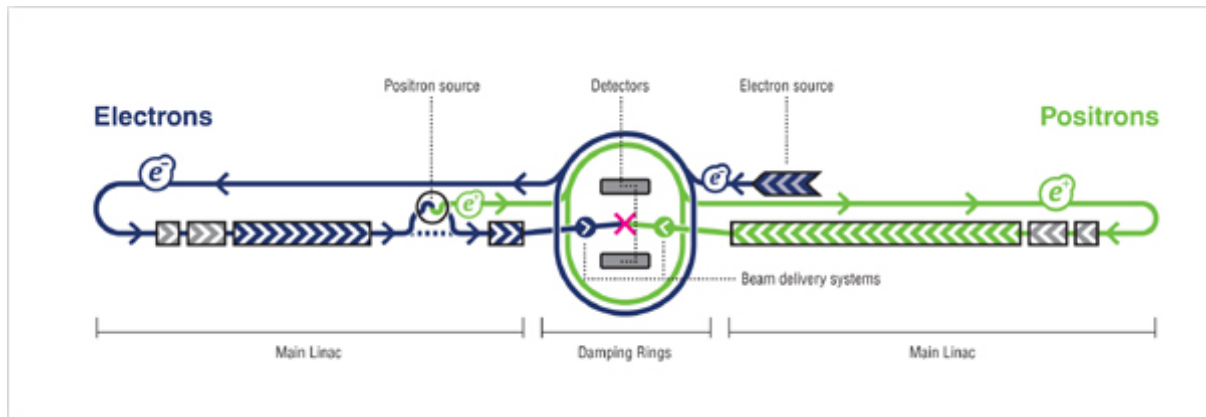


Figure 1: Conceptual design of the ILC

3 International Large Detector

The International Large Detector (ILD) is one of the two proposed detectors that are to be used in the ILC. The ILD is designed to have very accurate tracking capabilities and a highly granular calorimetry. This, coupled with the relatively clean collisions of particles (compared to hadron colliders) should allow for the high accuracy required for the precision measurements of new physics.

3.1 Main Components of ILD

The ILD, like most detectors, has an onion-like structure, which consists of the following layers [1, page 2]:

- A multi-layer pixel-vertex detector (VTX).
- A system of strip and pixel detectors surrounding the VTX detector.
- A time projection chamber.
- A system of Silicon strip detectors.
- A highly segmented electromagnetic calorimeter.
- A highly segmented hadronic calorimeter.
- A superconducting magnet surrounding the calorimeters, giving an axial B-field of 3.5 Tesla.
- An iron yoke to catch the magnetic flux of the solenoid and act as a muon detector.
- A data acquisition system to maximise sensitivity.

These layers are shown in Figure 2.

No discussion will be made of any of the components except the time projection chamber, as this is the area of focus in the summer studentship. For more information on the other sections, see the ILC & ILD website or [1].

3.1.1 Time Projection Chamber

The time projection chamber (TPC) is the main system for tracking particle trajectories. It is a large cylindrical space which is filled with a gas. An electric field runs parallel to the length of the cylinder, and when a charged particle passes through the detector it leaves an ionised track of gas particles. The electric field then pushes the electrons from the ionised path towards the anode (and therefore the positive ions towards the cathode) of the detector, where the electron signal is amplified and collected as a signal. This gives precise measurements of the x and y coordinates. The z coordinate can be measured by knowing the drift velocity of the electrons and knowing when the particle entered and left the detector. This is shown in Figure 3.

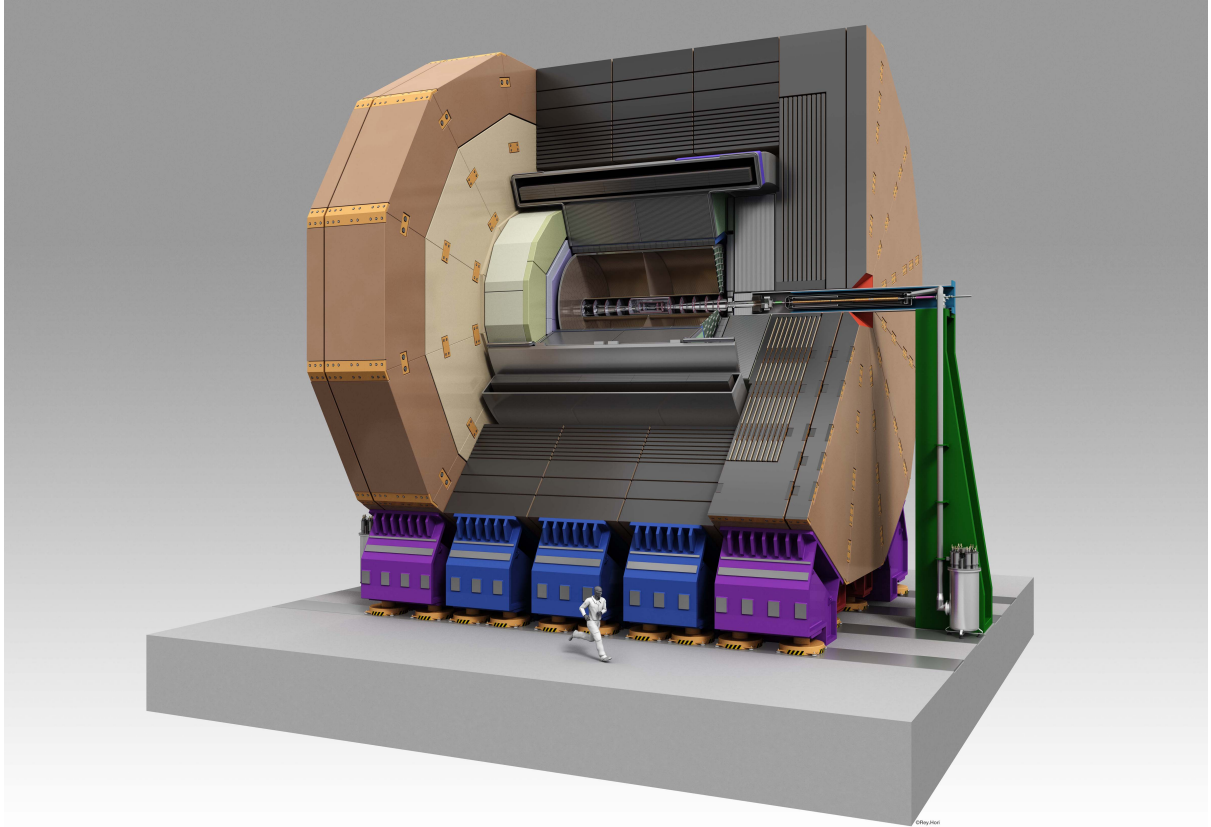


Figure 2: The International Large Detector [2]

4 Conditions Data

Conditions data is anything which isn't a direct measurement of the particles in the detector. It is used in almost all detector components to varying degrees. An example of its use in the TPC would be that in order to get the electron drift velocity, you would need to know the conditions of the gas at that time, such as the pressure, temperature, maybe the concentration etc. Another example would be pedestal values, so when a charge is detected at the endplate, it has to be offset by this pedestal value because it is possible for noise to create negative charge. So if we set the value of the noise to zero then we would lose some information. Therefore the pedestal offset value must be stored and referenced at some point when you want the precise measurement of the charge.

4.1 The Conditions Database

The conditions data is stored in a conditions database. This database is a MySQL database, and originally ATLAS wrote a program called ConditionsDBMySQL which was used to access the information stored in this database. However ATLAS abandoned this program at some stage, so the ILCSoft group created the LCCD class, which is an abstract layer designed to access the functionality of ConditionsDBMySQL. If a

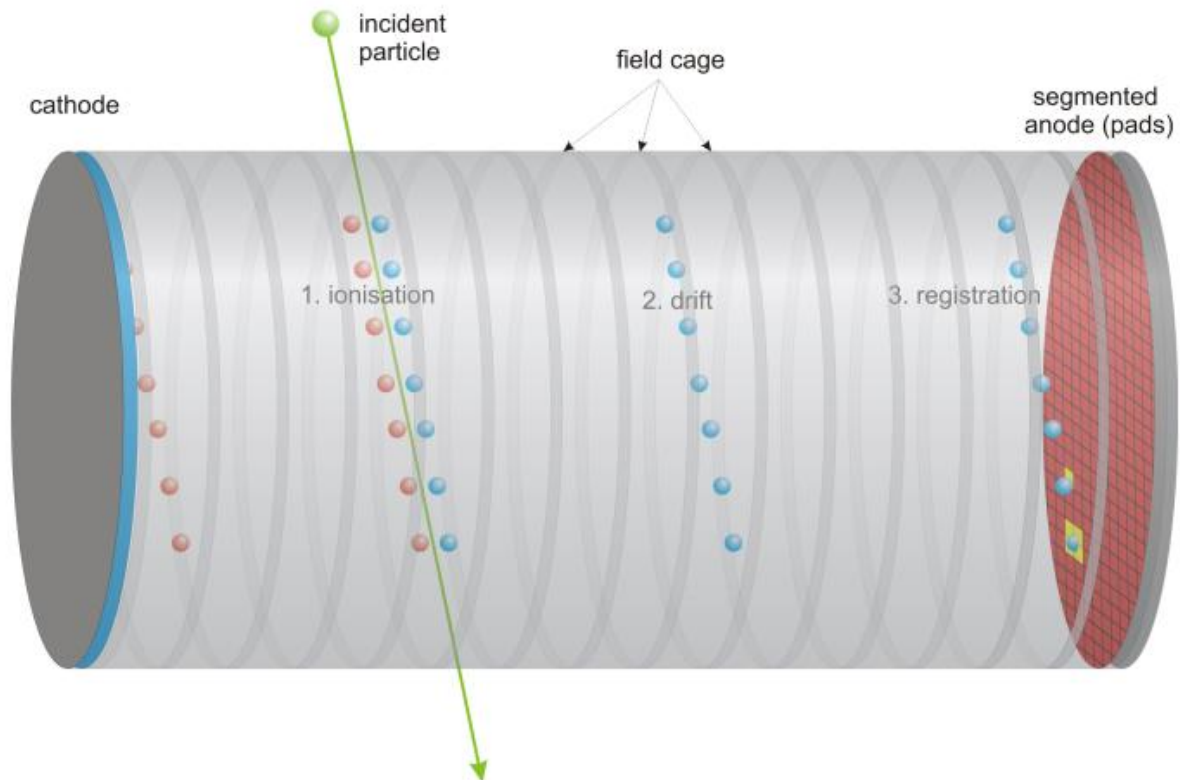


Figure 3: Shows a particle passing through a TPC and the signal being detected at the anode [3].

replacement to ConditionsDBMySQL is ever created, it will be easy for the LCCD class to adapt to this. Figure 4 shows this.

4.2 Errors in Conditions Data

Database administrators need to be able to access the data to do integrity and error checks to ensure that the database is consistent. Plus it is possible that people who write their reconstruction code make mistakes, and if so they will want to check if the mistake they made is in the conditions or not. For both of these things, it is currently very difficult and tiresome to check by writing a script or code that does this for you. A previous GUI existed that was designed to allow this kind of checking, but it was very unstable. This led to the need for a new GUI to be developed that was stable and allows the user to check in an easy and user friendly way whether their code was correct and to physically let them see the data that is stored in the conditions database.

4.3 Database Design and Object Structure

The database consists of folders, each of these folders can have many subfolders. Each subfolder can have many subfolders itself and so on. Eventually, you will reach a folder

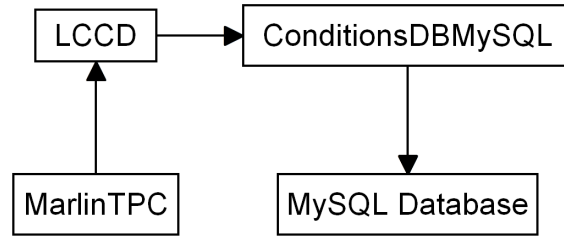


Figure 4: This example shows how the information in the MySQL database is accessed. You might start in MarlinTPC, which uses the LCCD Libraries, which in turn uses ConditionsDBMySQL, this finally accesses the information in the database.

which has no subfolders and this folder will contain some objects. Objects are where all the information related to a set of conditions is stored. Figures 5 and 6 show this.

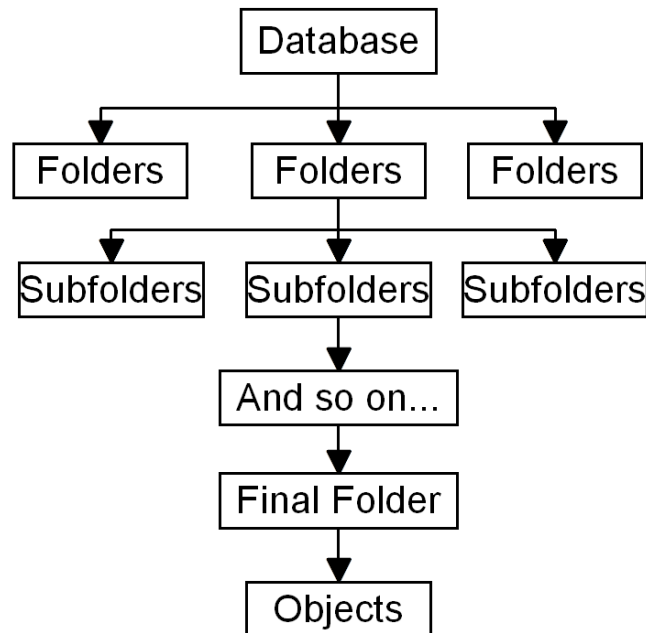


Figure 5: This shows the structure of the MySQL database

For each object, there is specific information which is stored. The information we are interested in is the start time, end time and layer number. The start and end time simply refer to when the records for the data begin and end. But they layer is a slightly more complicated idea: If an object is determined to be incorrect in some sense and the object is modified, the modified object is given a new layer. This stops the original object being deleted if it needs to be referred to later. An example of this is shown in Figure 7. When modifying a object, you should tag the new layer in a way which explains to the user

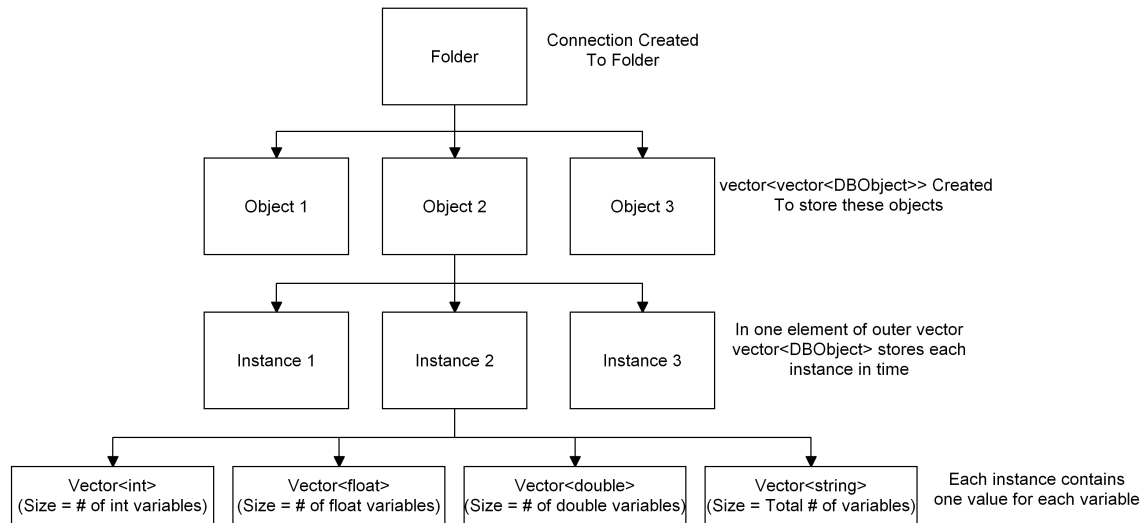


Figure 6: Structure of m.allObjects

what exactly has changed. Doing this tags all the objects with an exposed upper layer so that if you look again at Figure 7 The red object is not highlighted because there is no exposed upper layer, but all the others do have an exposed upper layer and so are tagged.

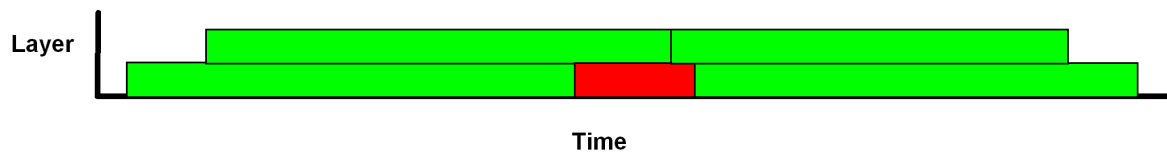


Figure 7: This shows five objects in a folder. Four of these objects have been tagged (green) and two of them are in the 2nd layer because they are modified.

5 Database Viewer

5.1 Introduction

The database viewer (dbViewer) is a graphical user interface (GUI) designed to allow easy access to the conditions database and show in a clear and concise way the raw data. Its use is for finding the mistakes from implementation of any code which accesses the conditions database. It does this by allowing a comparison between the database values and the results the user got (as a visual inspection by eye).

5.2 Working Example

5.3 Code Structure

The code is written in C++ and uses the Qt & LCCD/LCIO libraries. The code is written in a modular way where possible to allow for easy additions to functionality.

5.3.1 Using Qt

Most of the code is written using Qt libraries (Version 4.4). There is a vast amount of information online that will help in understanding how to use what is available [4]. But there are two major points that should be emphasised:

1. Signals and Slots
2. Inheritance.
3. Coordinate Systems.

5.3.1.1 Signals and Slots Qt uses a system called signals and slots in order to connect widgets and do things interactively. Some signals and slots are premade in the widget, however most of them are custom made in a class which is derived from a widget (At the very least, the class must at some level inherit from QObject, and must have the Q_OBJECT macro in the header file). A signal is emitted whenever the “emit” function is used, or if a connection is made to a signal from another signal and this connection is activated. A slot is a function which can be used simply as a normal function, but can also be used to connect to a signal. The connection (using QObject::connect) is a function that can be used to connect signals to other signals or slots. The positioning of the connection is very important, because an incorrect position causes unexpected behavior. If connecting class A and B via class C, then the connection should be in the constructor of class C to insure it gets initialised. However, if you are making a connection within a class, the connection needs to be made in the place just before the connection could be triggered (Although this is uncertain). A good way to debug unusual interactive behavior is simply to move the connection around until it works.

5.3.1.2 Inheritance In Qt, most classes have the option to inherit from others, with most of them leading to the `QObject` class. It is very important that when using the classes within Qt that you give each class the correct ‘parent’. This is because all the functionality of the parent can then be passed to the child class too, and stops issues with memory leaks etc. For instance, if you make a widget and within that widget you make another widget (with the first as its parent), then if the parent widget is closed the child widget is closed with it (as it should be). However if the 2nd widget does not inherit the first then the first being closed will not close the second.

5.3.1.3 Coordinate Systems There are 4 coordinate systems that can be used in Qt, each with varying purposes. Coordinates are important for deciding on the positioning of widgets and for determining where a mouse click occurred. The four systems are:

5.3.1.3.1 Screen Coordinates This is perhaps the least useful of the coordinate systems. (0,0) is located in the top left corner of the monitor, and increases pixel by pixel across your screen. This is particularly useless because it is very rare that two people will have the same monitors or screen resolutions.

5.3.1.3.2 Item Coordinates The item coordinates are used in conjunction with the `QGraphicsItem` class (and its subclasses). All `QGraphicsItems` contain a bounding rectangle which is the smallest rectangle that can be placed around the object which totally encompasses the object. The item coordinates start from the top left of this rectangle and increase in x and y as you go right and down respectively. Namely, this means they are the coordinates of the item itself, which might be useful if you want an item to do different things if you click on different points on it.

5.3.1.3.3 Scene Coordinates The scene coordinates are used in conjunction with the `QGraphicsScene` class. A `QGraphicsScene` is filled with `QGraphicsItems`

5.3.1.3.4 View Coordinates The View comes from the `QGraphicsView` class. It is populated with a Scene of Items and allows the user to scroll across the Scene.

All of the coordinate systems can be converted to each other with the `mapXtoY` functions that can be found throughout the `QGraphics` class range.

5.4 Connections

Table 1 is a table of all the connections that exist within the program.

5.5 Class Hierarchy

The following classes comprise the working sections of the code:

- mainWindow
- menuBar
- objectControl
- objectGrid
- dbObject
- objectRect
- objectTagging
- dbHierarchy
- dbTreeView
- taggingSystem

The class hierarchy is shown in Figure 8.

A detailed description of each class will follow.

5.5.1 Class: mainWindow

The mainWindow class contains the following:

- Public Members:
 - mainWindow(QWidget *parent = 0)

5.5.1.1 Public Members

5.5.1.1.1 mainWindow(QWidget *parent = 0) This is the default constructor and the only function in the mainWindow class. Within this class an instance of the objectControl and menuBar classes are instantiated, with both of the classes inheriting the QWidget that is mainWindow. The class is put into a layout with the menuBar class at (0,0) and the objectControl class at (1,0). This means that the menuBar class goes at the top of the page and the objectControl class goes directly beneath it. Finally, connections are made between the menuBar class and the objectControl class so that what happens in the menuBar class can be transferred to functions in the objectControl class.

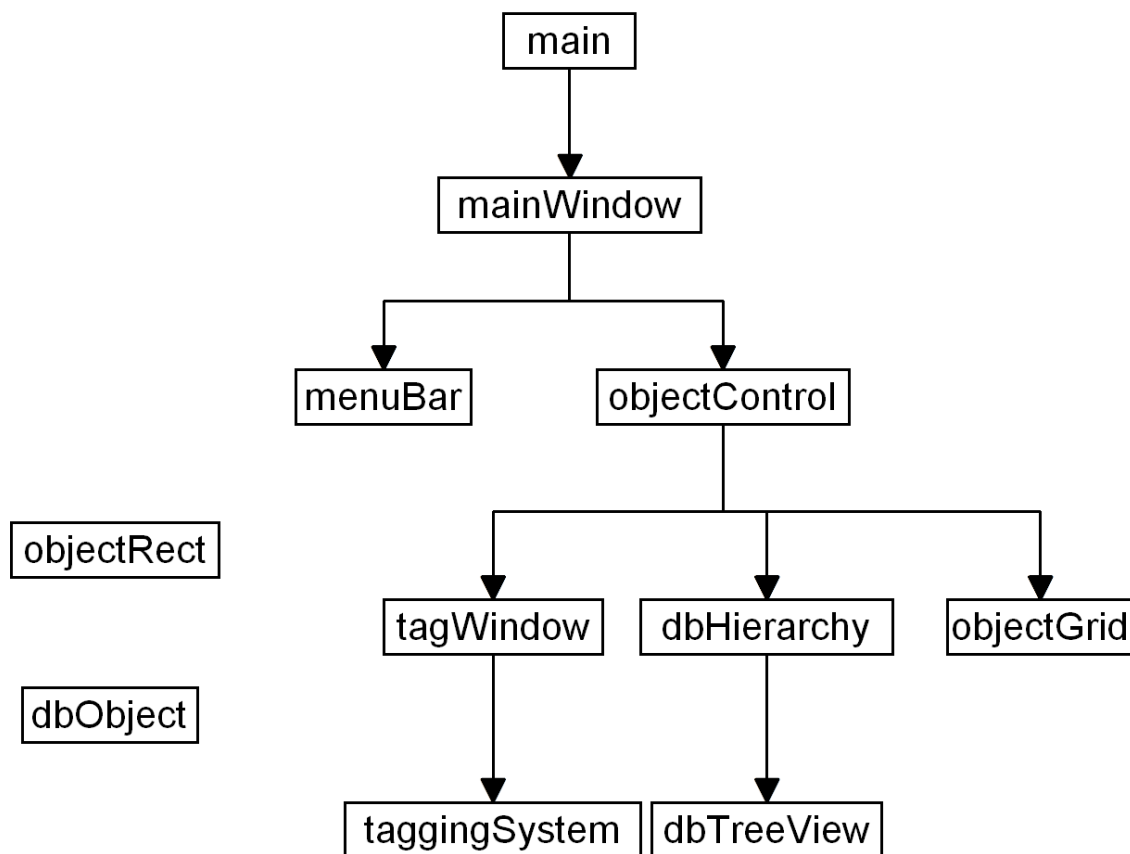


Figure 8: The class hierarchy showing where each class is created and with what features

5.5.2 Class: **menuBar**

The menuBar class contains the following:

- Private Members:
 - QAction *m_quitAct, *m_openDBAct, *m_closeDBAct, *m_resizeAct, *m_clearAct, *m_docAct
 - QMenu *m_fileMenu, *m_viewMenu, *m_helpMenu
- Public Members:
 - menuBar(QWidget *parent)
- Signals:
 - clearView()
 - docView()
 - openFile()

- closeFile()
- resizeGrid()
- Connections:
 - connect(m_quitAct, SIGNAL(triggered()), qApp, SLOT(quit()));
 - connect(m_openDBAct, SIGNAL(triggered()), this, SIGNAL(openFile()));
 - connect(m_closeDBAct, SIGNAL(triggered()), this, SIGNAL(closeFile()));
 - connect(m_clearAct, SIGNAL(triggered()), this, SIGNAL(clearView()));
 - connect(m_resizeAct, SIGNAL(triggered()), this, SIGNAL(resizeGrid()));
 - connect(m_docAct, SIGNAL(triggered()), this, SIGNAL(docView()));

5.5.2.1 Private Members All of the private members of the menu function come in one of two types, they are either a QAction, or a QMenu class.

5.5.2.1.1 QAction *m_ACTIONNAME The QAction class is designed to allow the user to associate an action with a widget (such as clicking on a button, for instance). QAction is derived from QObject directly. Each of the QAction names listed does the action you would associate with its name, but for completeness I will list it here explicitly:

- QAction *m_quitAct - Quits the program.
- QAction *m_openDBAct - Opens a new database.
- QAction *m_closeDBAct - Closes the currently open database.
- QAction *m_resizeAct - Resizes the current grid view. (NOT FULLY IMPLEMENTED)
- QAction *m_clearAct - Clears the current grid view.
- QAction *m_docAct - Opens the in-program help documentation. (NOT FULLY IMPLEMENTED)

5.5.2.1.2 QMenu *m_MENUNAME The QMenu items are the menus in which the actions must be placed. These correspond to the “File”, “View” and “Help” menu options at the top of the program.

5.5.2.2 Public Members

5.5.2.2.1 menuBar(QWidget *parent) The constructor is where everything is put together in the class to make the menu bar. A QMenuBar class is declared. All the private members are initialised. The addAction function is used to put the correct QActions into the correct QMenu lists. The addMenu function puts the QMenus in the QMenuBar. Connections are set at the end of the constructor.

5.5.2.3 Signals The signals are all designed to connect the menuBar class to other classes in the program.

5.5.2.4 Connections These all connect the trigger from the QAction (that is, when a button from the class is clicked, the QAction class has emits the triggered() signal) to the signals that have been created for the menuBar class. These signals that belong to the menuBar class can then be picked up by other classes so that the functionality can be implemented.

5.5.3 Class: objectControl

This is the main control class for the program. Most other classes are either created here or connected through here in some sense.

The objectControl class contains the following:

- Private Members:
 - QList<objectRect*> m_objectList
 - objectGrid *m_grid
 - tagWindow *m_tags
 - dbHierarchy *m_tree
 - QLineEdit *m_hostLine, *m_databasenameLine, *m_loginLine, *m_passwordLine, *m_portLine, *m_directoryLine
 - std::vector< std::vector< dbObject > > m_allObjects
- Public Members:
 - objectControl(QWidget *parent)
 - void sendAllToGridTest()
 - void newTestObject(double xCo, double yCo, double xLength, double yLength, std::vector<dbObject> testObject, QColor colour)
- Signals:
 - clearSignal()
 - sendObjectShapesTest(QList<objectRect*>)
- Private Slots:
 - docView()
 - clear()
 - openFile()
 - openFile(std::string directory)
 - closeFile()
 - resizeGrid()

5.5.3.1 Private Members

5.5.3.1.1 QList<objectRect*> m_objectList A QList is a list of contain from the Qt library that works very similarly to the STL container “List”. This list is filled with pointers to the objectRect class, which is discussed in detail in Section 5.5.6. m_objectList is the entire collection of rectangles that will be displayed in the viewer.

5.5.3.1.2 objectGrid *m_grid This declares an instance of the grid. The grid is described in detail in Section 5.5.4. Briefly, the grid the the QGraphicsView custom class designed to let the user view and interact with the QGraphicsScene (See Section 5.3.1.3).

5.5.3.1.3 tagWindow *m_tags This declares an instance of the tagging window and system. The tagging window and system are described in Section 5.5.7 and 5.5.8 respectively. This creates the widget which can be seen in the bottom left part of the GUI and eventually is used to control the tags and highlighting.

5.5.3.1.4 dbHierarchy *m_tree The dbHierarchy class is very similar to the tagWindow class in that it is the widget which will store the database tree view when a database is opened. Further references to this can be found by looking at Section 5.5.9 and 5.5.10.

5.5.3.1.5 QLineEdit *m_LINEEDITNAME The QLineEdit class is a class which allows the user to input information on a line (normally in a dialog window, see QDialog class in the reference documentation for more information). Each of the QLineEdits corresponds to something which must be input to make a connection to the database.

5.5.3.1.6 std::vector< std::vector< dbObject > > m_allObjects This is a vector of vector of the dbObject class. The dbObject class is designed to contain all the information associated with the conditions at a single instance, detailed documentation for this can be found in Section 5.5.5. The top level vector contains each object that is stored in the folder you are currently in. The vector within that vector stores each instance of the object. This is all shown in Figure 6.

5.5.3.2 Public Members

5.5.3.2.1 objectControl(QWidget *parent) This is the constructor for the objectControl class. In here m_grid, m_tags, and m_tree are initialised. A QGridLayout is created and the 3 widgets are placed in to it with an appropriate layout. Connections between the various classes are also created at the end of the constructor.

5.5.3.2.2 void sendAllToGridTest() This function simply emits the signal sendObjectShapesTest(m_objectList)

5.5.3.2.3 void newTestObject(double xCo, double yCo, double xLength, double yLength, std::vector<dbObject> testObject, QColor colour) This function creates an objectRect class and appends it to the m_objectList member. xCo and yCo are the x and y coordinates in the Scene coordinate system and xLength and yLength are the extensions to the object. For instance, if xCo = yCo = 0 and xLength = yLength = 10, then a 10x10 pixel square is formed at the origin of the QGraphicsScene. testObject contains the information that is associated with the rectangle that is being made and colour simply contains the colour of the rectangle which is necessary for tagging later.

5.5.3.2.4 void clear() This clear function goes through all members which store information related to the database objects and clears them. It then emits the signal clearSignal() (Section 5.5.3.3.1).

5.5.3.3 Signals

5.5.3.3.1 clearSignal() This signal is connected to the objectGrid class to the slot called clear(). Therefore the emittance of this signal clears the grid of any rectangles and gives it a blank background. It does NOT get rid of the objects and information stored in the objectControl class such as m_allObjects.

5.5.3.3.2 sendObjectShapesTest(QList<objectRect*>) This signal is connected to the objectGrid class to the slot called addObjectShapesTest(). The emittance of this signal leads to a function where the rectangles created in the objectList member are physically placed onto the QGraphicsView.

5.5.3.4 Private Slots

5.5.3.4.1 docView() The signal that links to this slot is docView() in menuBar class. This slot currently creates a QMessageBox and informs the user that at some point help documentaiton will be added. This is missing functionality which will be referred to further in Section 5.6.3.1.

5.5.3.4.2 openFile() This is a very large function which begins by calling the clear() function. A dialog window is created using QDialog and it is populated with QLabels and the member functions QLineEdit. You can fill the QLineEdit with the relevant information, and if you click on cancel, nothing will happen, but if you click on OK an attempt will be made to connect to the database (using the connections created inside this function). Beware that if you input the incorrect information into the QLineEdit boxes there is no error handling for dealing with incorrect database names and the program will have a segmentation fault. The LCCD libraries are used to make a connection to the database through CondtionsDBMySQL [5]. This will not be discussed in detail, please see the associated reference for further information. This function is not very modular, and this could be improved in future versions of the code. When the LCCD libraries are opened, the code iterates through all instances of all the objects and stores them in m_allObjects.

By looking at the start and end time of all the objects we can work out a relative scale on which we can draw the rectangles. At the end of the function the `objectRect` class instances are initialised and the signal `sendObjectShapesTest(m_objectList)` is emitted.

5.5.3.4.3 `openFile(std::string directory)` This works in much the same way as the slot above with the main difference being that this way of opening a folder is through the `dbHierarchy` class. Therefore a dialog window is not required because we already know the database and have already opened it. We are only changing the folder, which is the argument “directory”. This function will then close the connection and reopen it with the new directory.

5.5.3.4.4 `closeFile()` This closes the database by clearing all the members of `objectControl` and clearing the `objectGrid` class. It also clears anything which is stored in the tags window.

5.5.3.4.5 `resizeGrid()` The functionality for this slot is not written yet, but its purpose will be to resize the grid so that objects can be seen which are normally not visible.

5.5.4 Class: `objectGrid`

This class inherits from `QGraphicsView`. This is where the objects are shown and where you can interactively click on the objects in order to see the information stored within them.

The class contains the following:

- Private Members:
 - `void addBackground()`
 - `double resizeFactor(double dMaxValue)`
 - `QGraphicsItemGroup *backgroundGroup`
 - `QGraphicsItem *displayObject`
 - `QList<double> dimensionList`
 - `QList<QRectF*> mObjectShapes`
 - `double dMinValue`
 - `double dMaxValue`
 - `QList<objectRect*> m_testGraphicsItems`
- Protected Members:
 - `QPointF CurrentCenterPoint`
 - `QPoint LastPanPoint`
 - `void SetCenter(const QPointF & centerPoint)`

- QPointF GetCenter()
- virtual void mouseDoubleClickEvent(QMouseEvent* event)
- virtual void wheelEvent(QWheelEvent* event)
- virtual void resizeEvent(QResizeEvent* event)
- QGraphicsScene *Scene
- Public Members:
 - objectGrid(QWidget* parent = NULL)
 - void setSceneRectTest(int x, int y, int w, int h)
- Public Slots:
 - void clear()
 - void slot_highlightTaggedObjects(std::string tagName)
 - void addObjectShapesTest(QList<objectRect*> testGraphicsItems)

5.5.4.1 Private Members

5.5.4.1.1 void addBackground() The addBackground function starts with a check to see if objects already exist in the list m_testGraphicsItems. If no object exists then the background simply consists of the x and y axis lines and white space. However, if objects do exist, then as well as an x and y axis line, we get a grid of black lines that are created to make it easy to tell which layer each objectRect is on. Also the x axis is given a timestamp that scales such that 50 timestamps appear in total from the start to end time of the earliest and latest objects respectively. The fact that there are 50 timestamps shown is arbitrary and can be changed easily.

5.5.4.1.2 QGraphicsItemGroup *backgroundGroup This is just a convenience class provided by Qt that allows the programmer to group together lots of items into one single item. Therefore saving time when adding the items to the QGraphicsScene.

5.5.4.1.3 double dMinValue

5.5.4.1.4 double dMaxValue

5.5.4.1.5 QList<objectRect*> m_testGraphicsItems This is a list of the rectangles which contain the information within each folder.

5.5.4.2 Protected Members

5.5.4.2.1 QPointF CurrentCenterPoint

5.5.4.2.2 QPoint LastPanPoint

5.5.4.2.3 void SetCenter(const QPointF & centerPoint)

5.5.4.2.4 QPointF GetCenter()

5.5.4.2.5 virtual void mouseDoubleClickEvent(QMouseEvent* event) The virtual void mouse events are very important because they are what allows a program to be interactive. The mouseDoubleClickEvent function is perhaps the most important, because this is activated when the mouse does a double click anywhere within the QGraphicsView. The position of the mouse at the moment that the event occurs is recorded and the contains() function checks if one of the objectRect items is stored at the position of the mouse event. If it is then a function is called within the objectRect class called openTable(). If the position of the mouse at the moment that the event occurs is not over an objectRect, then nothing happens.

5.5.4.2.6 virtual void wheelEvent(QWheelEvent* event) This makes the mouse wheel into a zooming device, and makes the view zoom in and out when you rotate the mouse wheel. This was written by a previous summer student and works so I have left this function well alone whilst it works.

5.5.4.2.7 virtual void resizeEvent(QResizeEvent* event)

5.5.4.2.8 QGraphicsScene *Scene The QGraphicsScene is where the QGraphicsItems are placed before the scene is uploaded into the view.

5.5.4.3 Public Members

5.5.4.3.1 objectGrid(QWidget* parent = NULL) This is the constructor and this is where the Scene is added to the view. After it is added the addBackground() function is called.

5.5.4.3.2 void setSceneRectTest(int x, int y, int w, int h)

5.5.4.4 Public Slots

5.5.4.4.1 void clear() This function clears the grid of any rectangles stored in it. It does this by removing the items from the Scene. Then it clears the list of m_testGraphicsItems. The Scene is updated and then the addBackground() function is called.

5.5.4.4.2 void slot_highlightTaggedObjects(std::string tagName) This slot is connected to the taggingWindow class via the objectControl class. It is used to make the rectangles in the view change colour if a tag is chosen.

5.5.4.4.3 void addObjectShapesTest(QList<objectRect*> testGraphicsItems) This adds all the items to the scene and then updates the scene and view.

5.5.5 Class: dbObject

- Private Members:
 - int noInts, noFloats, noDoubles
 - std::vector< int > intValue
 - std::vector<float> floatValue
 - std::vector<double> doubleValue
 - std::vector<std::string> stringValue
 - lcio::LCTime startTime, endTime
 - std::string TypeName, argument1, argument2, tag
- Public Members:
 - dbObject()
 - dbObject(int p_noInts, int p_noFloats, int p_noDoubles, lcio::LCTime p_startTime, lcio::LCTime p_endTime)
 - ~dbObject()
 - void clear()
 - void setnoInts(int x)
 - void setnoFloats(int x)
 - void setnoDoubles(int x)
 - void setintValue(int i,int x)
 - void setfloatValue(int i, float x)
 - void setdoubleValue(int i, double x)
 - void setstringValue(int i, std::string x)
 - void setTag(std::string x)
 - void setargument1(std::string x)
 - void setstartTime(lcio::LCTime x)
 - void setendTime(lcio::LCTime x)
 - unsigned int getnoInts()
 - unsigned int getnoFloats()
 - unsigned int getnoDoubles()
 - unsigned int getnoStrings()
 - int getIntValue(int i)

- float getfloatValue(int i)
- double getdoubleValue(int i)
- std::string getstringValue(int i)
- std::string getargument1()
- std::string getTag()
- lcio::LCTime getstartTime()
- lcio::LCTime getendTime()
- dbObject & operator=(const dbObject &rhs)

5.5.5.1 Private Members

5.5.5.1.1 Int noInts, noFloats, noDoubles The variables stored in each instance of an object can be either integers, floats or doubles. This contains the number of each of these which are variables. For instance, if your object contains 2 integer variables, 1 float variable and no double variables. Then those numbers will correspond to the noInts, noFloats and noDoubles respectively.

5.5.5.1.2 std::vector< int > intValue (and floatValue and doubleValue) Each of these vectors contains one instance of each variable. Refer to Figure 6 for more information.

5.5.5.1.3 std::vector<std::string> stringValue The stringValue is a vector with a size equal to the sum of the sizes of the vectors for intValue, floatValue and doubleValue. It contains the names of the variables in the order in which they are read into the int, float and double Values.

5.5.5.1.4 lcio::LCTime startTime, endTime These contain the start and end time for the objects. They are stored in an LCIO class called LCTime.

5.5.5.1.5 std::string TypeName, argument1, argument2, tag TypeName is unused at this point but will contain the value of the typename given for the object from the LCCD class. argument1 and argument2 are the arguments needed to access the database and folder respectively and give identifiers for the objects. tag is the tag associated with the object, this one must be changed as mentioned in Section 5.6.1.2.

5.5.5.2 Public Members

5.5.5.2.1 dbObject() and dbObject(int p_noInts, int p_noFloats, int p_noDoubles, lcio::LCTime p_startTime, lcio::LCTime p_endTime) The empty constructor simply initialises everything to 0 or NULL where possible. The parameterised constructor fills the dbObject with all the currently used information which is necessary. There should probably be more parameterised constructors which, for instance, don't require a knowledge of noInts/Floats/Doubles but can work them out based on a vector of each of the above.

5.5.5.2.2 ~dbObject() The destructor sets all values to zero and clears all dynamically allocated memory.

5.5.5.2.3 void clear() The clear function is very similar to the destructor.

5.5.5.2.4 Set and Get Functions The set and get functions do as you would expect them to do in all cases.

5.5.5.2.5 dbObject & operator=(const dbObject &rhs) The assignment operator is used to set one dbObject equal to another one.

5.5.6 Class: objectRect

This class inherits from QGraphicsItem, which is important because it means that the objectRect class is a custom QGraphicsItem and thus contains all the functionality of the QGraphicsItem class with all the virtual functions being overridable. The class contains the following:

- Private Members:

- double m_xpos, m_ypos, m_xlength, m_ylength
- std::vector<dbObject> m_instancesofObject
- QColor m_colour

- Public Members:

- objectRect(QGraphicsItem *parent = 0)
- objectRect(double xpos, double ypos, double xlength, double ylength, std::vector<dbObject> instancesofObject, QColor colour, QGraphicsItem *parent = 0)
- double x()
- double width()
- double y()
- double height()
- QRectF boundingRect() const

- void paint(QPainter *painter, const QStyleOptionGraphicsItem *option, QWidget *widget)
- void openTable()
- dbObject getObjectInstance()
- void setColour(QColor colour)

5.5.6.1 Private Members

5.5.6.1.1 double m_xpos, m_ypos, m_xlength, m_ylength These correspond to the x and y coordinates and the length and height of each of the rectangles.

5.5.6.1.2 std::vector<dbObject> m_instancesofObject The object must contain all the instances of the variables so that they can be accessed in the table later. So they are stored in this variable.

5.5.6.1.3 QColor m_colour This contains the colour of the rectangle.

5.5.6.2 Public Members

5.5.6.2.1 objectRect(QGraphicsItem *parent = 0) and objectRect(double xpos, double ypos, double xlength, double ylength, std::vector<dbObject> instancesofObject, QColor colour, QGraphicsItem *parent = 0) The uninitialised constructor makes a rectangle with no height or width and therefore basically does not exist. The parameterised constructor gives the rectangle a position, dimensions and fills it with the associated data, as well as giving it a colour.

5.5.6.2.2 double x(), y(), width(), height(), getObjectInstance() These are all instances of get functions, to get the x and y coordinates. The getObjectInstance is slightly different in that it gives you the first instance of an object, so that data can be accessed such as the tags or startTime/endTime etc.

5.5.6.2.3 QRectF boundingRect() const The bounding rectangle is the rectangle in which the object must lie and therefore also sets the items coordinate system. The bounding rectangle for this class is very simple because the shape of the QGraphicsItems are indeed rectangles themselves.

5.5.6.2.4 void paint(QPainter *painter, const QStyleOptionGraphicsItem *option, QWidget *widget) This paints the object and is called by the QGraphicsView.

5.5.6.2.5 void openTable() This is the function which opens QTableWidget and fills it with the information from the instances of the object. The table is sortable, but currently contains no additional functionality.

5.5.6.2.6 void setColour(QColor colour) This sets the colour of the objectRect, this is used to highlight tagged objects.

5.5.7 Class: tagWindow

This class is derived from QWidget. The purpose of this class is to contain the window in which the tagging system will operate. Because it uses signals and slots the Q_OBJECT macro is used. The class contains the following:

- Private Members:
 - taggingSystem *treeWidget
- Public Members:
 - tagWindow(QWidget *parent = 0) : QWidget(parent)
 - void showTagsList(std::vector< std::string > allTags)
 - void clear()
- Signals:
 - signal_showTaggedObjects(std::string)

5.5.7.1 Private Members

5.5.7.1.1 taggingSystem *treeWidget The taggingSystem class is described in detail in Section 5.5.8. But this is where it is declared and used.

5.5.7.2 Public Members

5.5.7.2.1 tagWindow(QWidget *parent = 0) : QWidget(parent) The constructor creates the window, gives it a header, makes a connection to the from the treeWidget to the class we are currently in for the showTags signals.

5.5.7.2.2 void showTagsList(std::vector< std::string > allTags) The argument from this is taken from ConditionsDBMySQL and the function is called in the openFile slot of the objectControl class. This function iterates through the vector of strings and makes a QTreeWidgetItem that only goes one layer deep and has the names of the tags in the text.

5.5.7.2.3 void clear() This clears the treeWidget function using the QTreeWidgetItem::clear() built in function.

5.5.7.3 Signals

5.5.7.3.1 signal_showTaggedObjects(std::string) This signal is connected from this class to the objectControl class and from there the objectRect items are changed to the colour red if they have a tag which matches the string passed as an argument.

5.5.8 Class: taggingSystem

This class is derived from QTreeWidget, and therefore has all its properties. It contains the following:

- Private Members:
 - QAction *m_showObjectsWithTag, *m_deleteTag
 - QMenu *m_menu
 - QPoint *m_item
- Protected Members:
 - virtual void mousePressEvent(QMouseEvent *event)
- Public Members:
 - taggingSystem(QTreeWidget *parent = 0) : QTreeWidget(parent)
- Signals:
 - signal_showTaggedObjects(QPoint)
 - signal_showTaggedObjects(std::string)
- Public Slots:
 - slot_showTaggedObjects()
 - slot_showTaggedObjects1(QPoint itemPos)

5.5.8.1 Private Members

5.5.8.1.1 QAction *m_showObjectsWithTag, *m_deleteTag These actions are both used to do the associated tasks by means of signals and slots. The deleteTag QAction does not yet do anything and is part of the missing functionality that must be included by future modifications. The showObjectsWithTag QAction connects the trigger of this QAction to slot_showTaggedObjects.

5.5.8.1.2 QMenu *m_menu This is the menu that will appear when the user right clicks on an item in the tagWindow class. It will contain the two action members of this class.

5.5.8.1.3 QPoint m_item This point is used to determine if an QTreeWidgetItem is found when the mouse is clicked.

5.5.8.2 Protected Members

5.5.8.2.1 virtual void mousePressEvent(QMouseEvent *event) This functions checks to see if it was the right or left mouse button that was pressed. If it was the left mouse button and the QTreeWidgetItem has children, then the item is expanded so you can see its children. Likewise if it is already expanded you can contract it by left clicking on it. The right mouse button opens the m_menu at the position where the mouse button is clicked and makes a connection of the trigger to the slot for the two QActions.

5.5.8.3 Public Members

5.5.8.3.1 taggingSystem(QTreeWidgetItem *parent = 0) : QTreeWidgetItem(parent) All the constructor does in this class is make the connection between the signal_showTaggedObjects(QPoint) and the slot called slot_showTaggedObjects1(QPoint).

5.5.8.4 Signals

5.5.8.4.1 signal_showTaggedObjects(QPoint) This signal has an argument of a QPoint so corresponds to a position on the QTreeWidgetItem. This is connected to slot_showTaggedObjects1(QPoint itemPos).

5.5.8.4.2 signal_showTaggedObjects(std::string) This is the signal with the tag name in the string as an argument. This is connected to the slot in objectGrid (slot_highlightTaggedObjects(std::string)).

5.5.8.5 Slots

5.5.8.5.1 slot_showTaggedObjects() All this slot does is emit the signal signal_showTaggedObjects(QPoint) with the current value where the mouse event occurred.

5.5.8.5.2 slot_showTaggedObjects1(QPoint itemPos) This slot searches for an item at the position itemPos, from here it takes the string from the QTreeWidgetItem it finds and emits the signal signal_showTaggedObjects(std::string) with this string as the argument.

5.5.9 Class: dbHierarchy

This class is very similar to the tagWindow class. It creates the window in which a QTreeWidgetItem will be placed containing the database folders. The database folders will be layered in a logical way and when you reach the bottom layer you will be able to open the folder from the window. The class contains the following:

- Private Members:
 - dbTreeView *treeWidget

- `std::multimap<std::string, QTreeWidgetItem*> itemList`
- `QAction *m_openFolder, *m_showTags, *m_deleteFolder`
- `std::string m_preamble`
- `std::string findBottomLayer(std::string fullPath)`
- `void addItem(std::string fullPath, std::string layerName, QTreeWidgetItem *childItem)`
- Public Members:
 - `dbHierarchy()`
 - `void addDatabase(std::vector< std::string > dbFolder)`
 - `void clear()`
 - `void setPreamble(std::string preamble)`
 - `std::string getPreamble()`
- Signals:
 - `signal_openFolder(std::string)`

5.5.9.1 Private Members

5.5.9.1.1 `dbTreeView *treeWidget` This contains the class `dbTreeView` which is a `QTreeWidgetItem` class. This class will contain all the information of the `dbHierarchy`.

5.5.9.1.2 `std::multimap<std::string, QTreeWidgetItem*> itemList` This multimap contains a string for a key and the `QTreeWidgetItem` as the value. The string for each of these `QTreeWidgetItem`s is the full path which takes you to this point. For Instance, in the example folder: `“/rdiener/test/pedestal”`, each of the words would be a `QTreeWidgetItem`. The top one contains the text `“rdiener”`, and the key associated with it is also `“rdiener”`. The next one would contain the text `“test”`, but the key would be `“rdiener/test”`. This is necessary and will be explained further later

5.5.9.1.3 `QAction *m_openFolder, *m_showTags, *m_deleteFolder` These `QActions` will be put into a `QMenu` when the user right clicks on a folder. The `openFolder` will open the folder of the currently clicked object. The `showTags`, when implemented, will show all the tags stored in this folder and its subfolders. The `deleteFolder`, when implemented, will delete the current folder and all its subfolders.

5.5.9.1.4 `std::string m_preamble` The preamble is the connection argument to the database, and is necessary for use with opening a new folder from the `dbHierarchy`.

5.5.9.1.5 `std::string findBottomLayer(std::string fullPath)` This is a function which does string magic in order to get the bottom word from a path, for instance with the example above of “/rdiener/test/pedestal”, if that string is entered into this function then the returned string is “pedestal”.

5.5.9.1.6 `void addItem(std::string fullPath, std::string layerName, QTreeWidgetItem *childItem)` This is a recursive function designed to add all the QTreeWidgetItems to the QTreeWidgetItem. The idea is that when we have all the folders (which are obtained in the openFile function of objectControl), we can do a for loop through all the full paths of the folder lists and add apply addItem for each of these. This starts with the bottom layers name and then makes a QTreeWidgetItem with this bottom layer as the text. It then removes this bottom layer from the full path and makes a new item with this new full path. This continues until it reaches the top layer which is where the recursion ends. At each level, the child is related to the parent QTreeWidgetItem. There are checks in place that if a full path is repeated because two bottom layers contain the same upper layer, then one of the upper layers is ignored. This will then automatically make a correct and full QTreeWidgetItem system of folders.

5.5.9.2 Public Members

5.5.9.2.1 `dbHierarchy()`

5.5.9.2.2 `void addDatabase(std::vector< std::string > dbFolder)`

5.5.9.2.3 `void clear()`

5.5.9.3 Signals

5.5.9.3.1 `signal_openFolder(std::string)`

5.5.10 Class: dbTreeView

5.6 Next Steps for dbViewer

5.6.1 Known Bugs

5.6.1.1 Tagging a whole object incorrectly There is an issue in the code currently that if an object has an exposed moment in time for which it is valid, and a tag is created, then the entire object across all moments is tagged. This also means that the entire object is highlighted. There is no simple way to fix this issue currently, except to have some sort of check for a high layer at every instant in time and if this is true to separate the objects into 2 objects at this point.

5.6.1.2 Only one tag per object Currently, only one tag can be applied to a single object. So this must be changed to something like a vector of tags in the dbObject class and then iterated through in order to have multiple tags per object.

5.6.2 Unknown Bugs

5.6.2.1 Crash when trying to reopen a database If you open the database and then choose to close the current database from the File Menu. Then if you try to reopen it or a different database from the File Menu the program will experience a segmentation fault. The cause of this error is currently unknown.

5.6.3 Missing Functionality

5.6.3.1 In-Program Help Documentation

5.6.3.2 Object Table Functionality copy paste sort highlight all get additional info.

5.6.3.3 Resize Grid This allows the user to manually input values for the view, such that you can zoom in on a particular part of the scene.

5.6.3.4 Delete Folders

5.6.3.5 Show All Tags

5.6.3.6 Create New Tags

5.6.3.7 Delete Tags

5.6.3.8 Show Object Specific Information

5.6.3.9 Exception Handling

5.6.3.9.1 Overlapping Objects

5.6.4 Additional Features

6 Summary

References

- [1] ILD Concept Group. The international large detector letter of intent. Technical report, DESY, Fermilab and KEK, 2010.
- [2] International Large Detector Group. Images of the ild. Found at <http://www.ilcild.org/documents/mdi/ild-illustrations/tentative-version-as-of-2011-aug.30>, 2011.
- [3] Ralf Diener. Basic principles of tpcs and gems. Found at <http://www-fle.desy.de/tpc/basics.php>, June 2007.
- [4] Nokia Corporation. Qt reference documentation. Found at <http://doc.qt.nokia.com/4.4/index.html>, 2008.
- [5] Ralf Diener. Setting up and using a conditions database with lccd and conddbmysql or lccd for dummies. Version v0.31, Accessible at DESY, February 2011.

Class Signal Comes From	Class Signal Goes To	Item In Class	Signal Emitted	Signal/Slot Receiving	Location of Connection
menuBar	objectControl		clearView()	clearSignal()	mainWindow:: mainWindow
menuBar	objectControl		docView()	docView()	mainWindow:: mainWindow
menuBar	objectControl		openFile()	openFile()	mainWindow:: mainWindow
menuBar	objectControl		closeFile()	closeFile()	mainWindow:: mainWindow
menuBar	objectControl		resizeGrid()	resizeGrid()	mainWindow:: mainWindow
menuBar	qApp	m_quitAct	triggered()	quit()	menuBar:: menuBar
menuBar	menuBar	openDBAct	triggered()	openFile()	menuBar:: menuBar
menuBar	menuBar	closeDBAct	triggered()	closeFile()	menuBar:: menuBar
menuBar	menuBar	clearDBAct	triggered()	clearView()	menuBar:: menuBar
menuBar	menuBar	resizeDBAct	triggered()	resizeGrid()	menuBar:: menuBar
menuBar	menuBar	docDBAct	triggered()	docView()	menuBar:: menuBar
objectControl	objectGrid		clearSignal()	clear()	objectControl:: objectControl
objectControl	objectGrid		sendObject ShapesTest (QList jobjectRect* <i>l</i>)	addObject ShapesTest (QList (QList jobjectRect* <i>l</i>)	objectControl:: objectControl
dbHierarchy	objectControl		signal_ openFolder (std::string)	openFile (std::string)	objectControl:: objectControl
tagWindow	objectGrid		signal_ showTagged Objects (std::string)	slot_ highlight TaggedObjects (std::string)	objectControl:: objectControl

Table 1: This table shows all the connections between the classes and within each class