Imagix

Understanding Class Structure and Interfaces Using Imagix 4D

The abstraction and encapsulation of object oriented programming offer many advantages when developing software, including limiting the amount of information necessary to utilize existing classes. Conversely, when making changes to a class or assessing its usability in a different context, it can be difficult to comprehend the inner workings and overall usage model. This is due to the challenge of understanding enough of the multifaceted nature of a class to ensure that any changes are compatible with the current use and in line with the original design intent.

Achieving this understanding is one of the uses of Imagix 4D. The focus of this application note is on Imagix 4D's graphical displays, and the range of insights they provide about a class, especially about its interface to other classes in a software system. We'll see how the displays work together to provide a thorough understanding of the class, its usage and dependencies.

The example in this note will use source code from FileZilla, an FTP client application available as open source from SourceForge. With Imagix 4D, we'll examine one particular class, CIOThread, and investigate how it interacts with the other code in FileZilla.

Graphical Display Concepts

In a software system of any significant scope, the amount of information about the software – its dependencies, control flow, etc - quickly becomes overwhelming. For a program comprehension and analysis tool such as Imagix 4D, the ability to find and present relevant information in an understandable way is fundamental.

This application note will focus on the actual displays of Imagix 4D and what they reveal about CIOThread's structure and interfaces. In order to keep Imagix 4D simultaneously informative, intuitive and easy to apply, these displays employ several user interface concepts:

Drill Downs – Imagix 4D provides an extensive range of graphical view types, These can start simple and then show more and more detail with a few clicks. Drilling down through increasingly granular views provides more detailed understanding of specific portions of the software.

Abstraction – A corollary of drill downs, abstraction is used to represent information at a less granular level than it actually occurs in the software. For example, function calls can be abstracted to the class level, so that ClassA::FunctionA calling ClassB::FunctionB can be represented as ClassA calling ClassB.

Display Scoping – While working within a given view type, the ability to focus on a specific portion of the software of interest is enabled by the ability to extend or trim the scope of any display.

View Partitioning – Distributing information between different view types, and enabling these multiple view types to be displayed side by side, avoids single displays becoming unusable due to data overload. For example, a graph showing the hierarchy of calls between functions can be viewed next to a display showing where those functions reside in source files in the directory structure.

Simple, Automated Transitions – Complementing the displays themselves is the ability to follow a line of inquiry from one display to the logical next step, simply and quickly.

The first four of these concepts will become more readily apparent through the displays you'll be seeing in this application note. What won't be as obvious is that the transitions between these displays are accomplished by one or just a few mouse or menu actions. Rather than describe these actions as part of the narrative, they are documented in the appendix at the end of this note.

With this background, we'll explore how Imagix 4D's displays provide insight into the class CIOThread.

Class Member Info

Imagix 4D provides many alternatives to get a quick overview of a class and its members. These include the Members and the Source Code tabs from the Symbol panel, as well as the class' source code itself in a File Editor. As the focus of this app note is on Imagix 4D's graphical views, we'll start with a UML Class Diagram of just the class CIOThread.

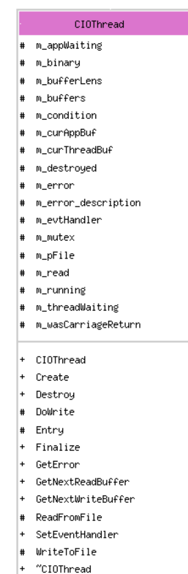

Figure 1. UML Class Diagram for single class, CIOThread, showing the class's member variables (upper section) and functions (lower section).

We immediately see that CIOThread contains many members, with seventeen variables in the attributes section and thirteen functions (methods) in the operations section. The # and + notation in front of the member names indicates that variables are all protected members while the functions are split between public and protected members.

We can learn a bit more about the functions in this class by looking at their internal calling relationships.

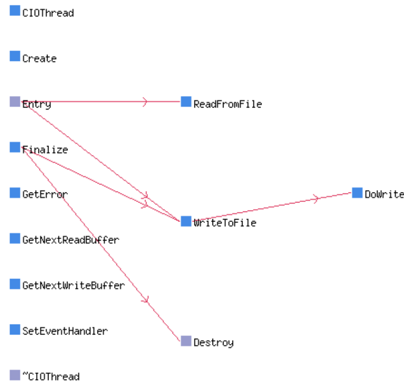

Figure 2. Function Call Tree view of CIOThread's member functions (blue squares) and their internal calls (red lines). The arrows and layout (left to right) indicate the hierarchy of the function calls.

The graph shows that, from a control flow sense, most of the functions are independent of each other. Only the protected members ReadFromFile, WriteToFile and DoWrite, along with the virtual function Destroy, are called from other member functions.

Displaying usage of variable members by the function members results in a much more complex and interrelated graph.

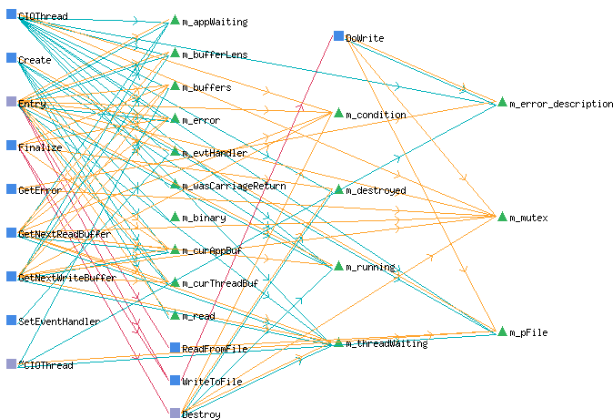

Figure 3. Function Calls with Variables extends the Function Call Tree by adding reads (orange lines) and sets (aqua lines) of member variables (green triangles).

In this view, the graph quickly and clearly indicates that while the member functions don't interact much from the control flow sense, they do a lot of data sharing.

In order to explore how a particular member variable is used, the next step would be to focus the graph on that specific variable.

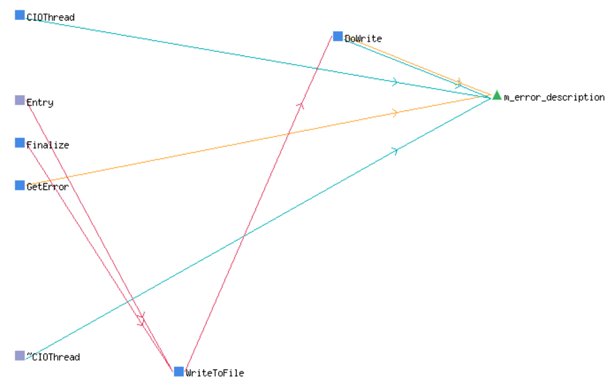

Figure 4. Function Calls with Variables simplified to show just the class members using the specific variable m_error_description.

The resulting graph identifies those functions having a direct or transitive dependency on the particular variable, m_error_description in this case. Typcially however, when first exploring a class, this more detailed inspection of individual variable usage would be omitted.

Class Level Views

Now that we have some initial knowledge about CIOThread itself, we'll begin to explore its interface to others classes in the system. An initial step might be to examine the class's inheritance by bringing up a Class Inheritance graph.

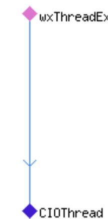

Figure 5. Class Inheritance view

From the graph, we immediately observe that CIOThread is derived from wxThreadEx, and that no further classes are derived from CIOThread.

At this point, we might choose to examine wxThreadEx, using the same approach we just used to study CIOThread's members. But for the purposes of this application note, we'll continue on with CIOThread itself, and explore its control flow interface by switching to the Class Calls view. The resulting graph shows classes calling and being called by CIOThread.

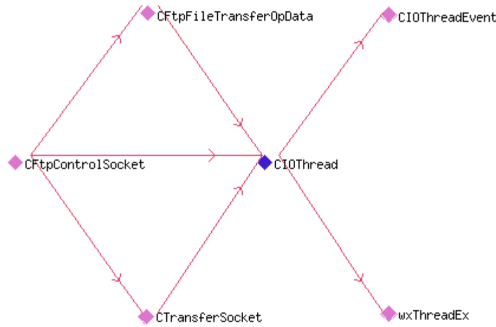

Figure 6. Class Calls view abstracts calls into and out of CIOThread to the class level.

This is of course an abstraction, since classes don't actually call other classes. So the call relationship from class CTransferSocket to CIOThread indicates that a member function of CTransferSocket calls a member function of CIOThread.

At this abstract level, we see that CIOThread's control flow interface is pretty clean. Only three classes make calls into CIOThread, and only two classes are called from CIOThread.

UML Class Diagrams

At this point, we have a pretty clear high level sense of how CIOThread is tied to other classes in a control flow sense. As a next step, we can explore these inter-class dependencies at a more granular level.

One view we might use for this is the UML Class Diagram. Generating this from the Class Calls view automatically results in a UML Diagram showing the same classes. Within each class, just the function members having call relationships (associations) to functions in other classes are displayed, in contrast to the full list of members that we saw in Figure 1.

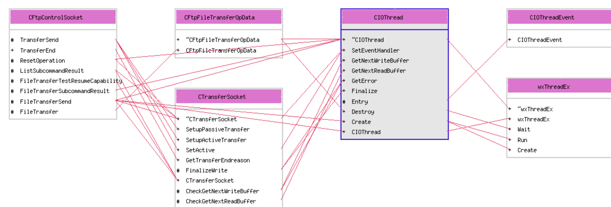

Figure 7. UML Class Diagram showing calling relationships between CIOThread and its associated classes.

Here, we're able to examine in more detail the function calls that were abstracted in the Class Calls view. We see the actual member functions making the calls, and those being called.

Because of CIOThread's relatively simple class-level calling hierarchy, this UML diagram is still fairly understandable. Often, a class's calling hierarchy, as displayed in the Class Calls view, is more complex. In such situations, the corresponding UML Class Diagrams contain a lot of data, and may exceed the amount of data that can be efficiently examined.

In these cases, it's useful to study the Class Calls graph side by side with a more focused UML Class Diagram. The Class Calls graph provides overall context and navigation. The focused UML Class Diagram is used to study in more detail the relationships between a specific pair of classes. For example, we can focus the UML Diagram on the relationships between CIOThread and wxThreadEx.

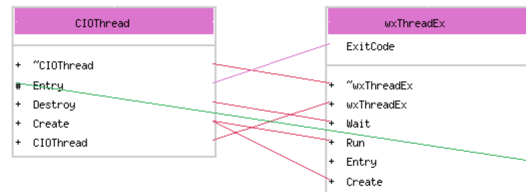

Figure 8. UML Class Diagram, showing all of the relationships between CIOThread and wxThreadEx.

By default, this diagram is configured to hide those class members having no relationship to the other class. We see specific functions of CIOThread that call into wxThreadEx. By electing to display all of the relationships between the classes, we also see that wxThreadEx::Entry is overridden by CIOThread::Entry (green line). And that wxThreadEX::ExitCode is used as a type by CIOThread::Entry (pink line).

Member Level Views

Thus far, our study of CIOThread's control flow interfaces has generally been at the class level. The UML Class Diagrams have enabled us to examine these at a more granular level, but still within the context of classes.

An alternative approach is to study the interfaces as a whole. If we're considering making changes to the class members, it becomes effective to focus on the members themselves, considering classes only secondarily.

One automatically generated graph shows the full inbound calling interface of a class, showing all the uses of the class's member functions.

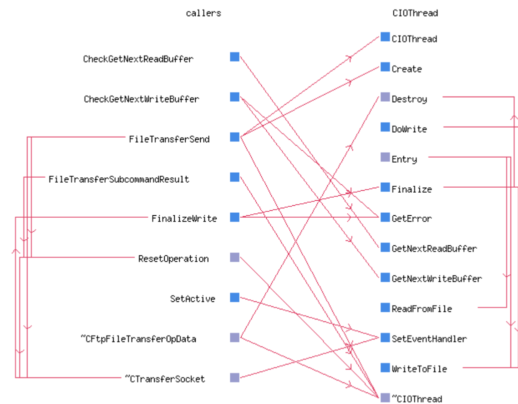

Figure 9. Function Call Tree view showing external users (on left) of CIOThreat's functions.

We're able to quickly get a sense of which functions are used most often, and which are rarely invoked.

In a related graph, we can examine the inbound interface of CIOThread's member variables.

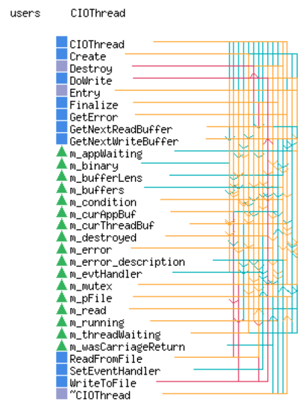

Figure 10. Function Calls with Variables view showing no external users of CIOThread's variable members

We see that there are no functions setting or reading the member variables from outside of CIOThread. This quickly confirms that the variables are fully encapsulated within the class.

A similar approach can be taken for analyzing CIOThread's dependencies, its full interface to external functions that are called by its member functions.

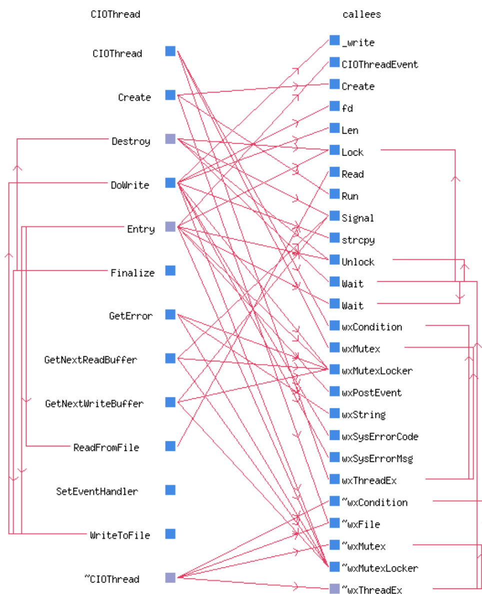

Figure 11. Function Call Tree showing external functions (on right) called from CIOThread member functions (on left).

This graph provides a first impression of the complexity of the member functions. We could choose to further study those functions making a lot of external calls. One approach would be to examine them in a flow chart.

Imagix 4D has a concept of library functions. These are functions that are declared, but not defined, in the source code being studied. Use of these library functions is typically of less interest when examining a given interface, and you may chose to hide the library functions from the graph.

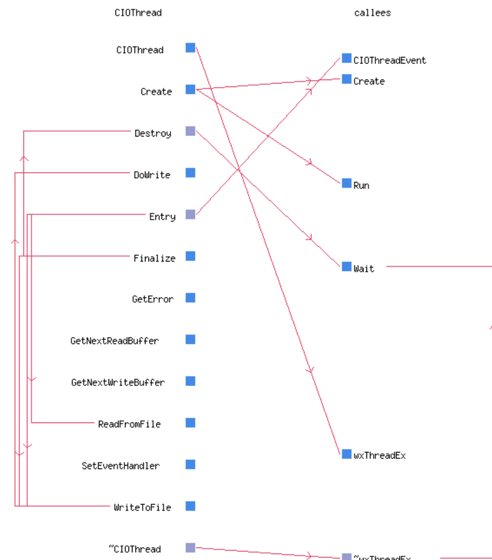

Figure 12. Function Call Tree resulting from hiding called library functions from Figure 11 display.

Eliminating the library functions from the graph results in a more focused graph, showing just the dependencies of CIOThread on other functions defined in the FileZilla source code.

Complementary / Combined Displays

Supplementing this range of graph displays, Imagix 4D has a series of display tabs providing additional information about the software.

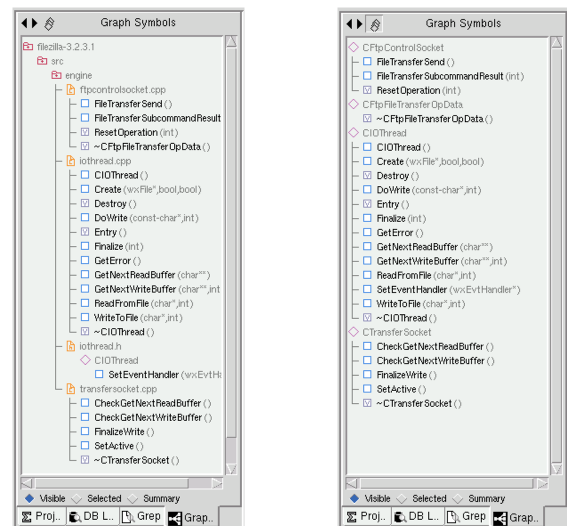

Figure 13. Graph Symbols tab, show file containers (left) and class / namespace containers (right) of CIOThread's member functions and their callers (figure 9).

The info in these tabs is complementary to that presented in the graphs. The particular tab most directly linked to the graph displays is titled Graph Symbols.

This tab lists the same symbols that appear in the current graph itself. But rather than show the relationships between those symbols, the Graph Symbols tab presents a different dimension of information. It indicates where the symbols reside within the software system, either physically (by file) or logically (by class and namespace).

The ability to view multiple displays side by side is one of the benefits of Imagix 4D. For example, suppose you wanted to get a full understanding of a class's control flow dependencies. You might have a Class Calls graph (figure 6) to provide a class level perspective. Next to it, you'd have a Function Call Tree graph showing the actual calls made from the class (figure 11) to external functions. And next to that, the Graph Symbols tab (figure 13) would enable you to identify which files and directories are involved.

By avoiding too much information in any one display, each display remains readily understandable. And together, they provide a very comprehensive picture of a class's dependencies.

Appendix – Mouse / Menu Actions

As explained with the graphical display concepts, minimizing the mouse and menu actions required for transitioning from one display to another is a key factor in keeping Imagix 4D simultaneously informative, intuitive and easy to apply. Descriptions of these actions have been omitted from the main body of this note, but are included here to aid any readers interested in retracing the steps described above.

Of the transitions from one display to the next in this application note, the majority involve the context sensitive menu that is invoked through clicking the right mouse button. In the following listing of mouse and menu actions, this is represented as **CS**. Following each CS is an indication of what needs to be clicked on. For example, CS (CIOThread) indicates that the mouse needs to be positioned over some representation of the class CIOThread when right-clicked. Initially, the easiest place to find the class CIOThread listed is in the Class Index tab.

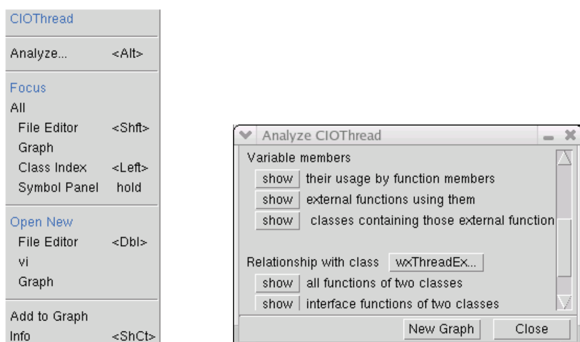

Figure 14. Context sensitive pop-up menu for CIOThread (left) and a scrolled portion of Analyze dialog (right).

One of these context sensitive menu actions is to bring up the Analyze dialog. The dialog first appears as the result of a step in creating figure 2. In this case, the dialog is focused on CIOThread.

The dialog automates the generation of a range of graphic displays about the focus symbol (CIOThread in this case). Several of these displays are discussed in this note. Many more exist. Display actions invoked from the Analyze dialog are represented below by **AD**.

GM indicates that the action is a menu item from the graph display.

Figure 1
- CS (CIOThread) > Focus Graph
- GM Filter > Isolate Selected
- GM View > UML Class Diagram

Figure 2
- CS (CIOThread) > Analyze...
- AD > Show function internal calling hierarchy

Figure 3
- AD > Show variable usage by members

Figure 4
- CS (m_error_description) > Select
- GM Traverse > Full Up
- GM Filter > Isolate Selected

Figure 5
- CS (CIOThread) > Open New Graph

Figure 6
- View > Class Calls
- CS (CIOThread) > Focus Graph

Figure 7
- CS (Class Calls graph, black background) > Create UML Diagram

Figure 8
- CS (Class Calls graph, line from CIOThread to wxThreadEx) > Analyze All Relationships

Figure 9
- AD > Show external functions calling members

Figure 10
- AD > Show external functions using variables

Figure 11
- AD > Show external functions called by members

Figure 12
- GM Filter > Hide Library

Copyright © 2009, Imagix Corporation
www.imagix.com
Imagix and Imagix 4D are trademarks of Imagix Corp.