Chapter 3

GUI application architecture

Modern GUI applications may be composed from a number of different software components. For example, a GUI application may access remote databases, or other machines, or it may be standalone. In this section we characterize some of the common software architectures for GUI applications.

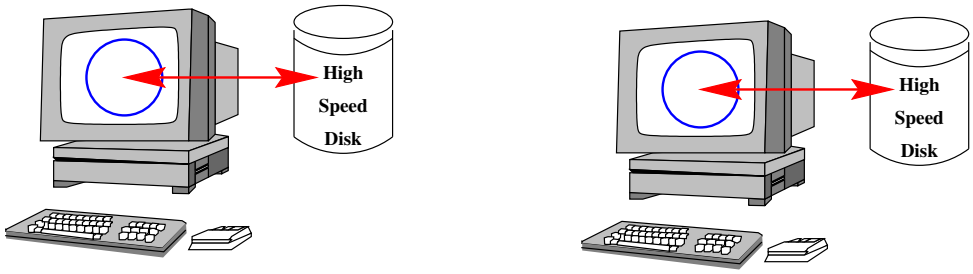
3.1 Architecture of GUI applications

We can categorise GUI applications in many different ways, but the overall communications architecture is of interest, and helps us select tools and development strategies. One classification is:

- Standalone
- Shared file
- Shared database
- Web based
  - Simple
  - Scripting
  - Java

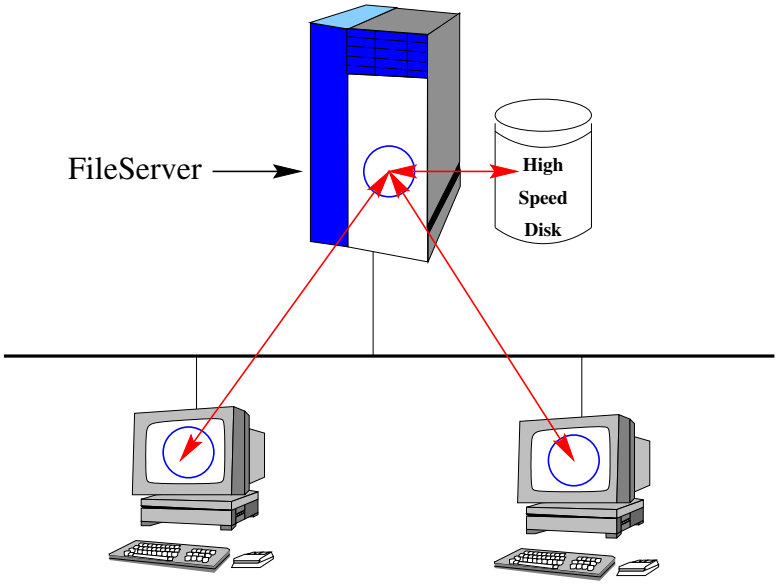
3.1.1 Standalone

A standalone GUI application runs on the user's PC, and reads and writes a local disk for files or (in a very general sense) databases. Note that Microsoft Access programs are normally of this sort.



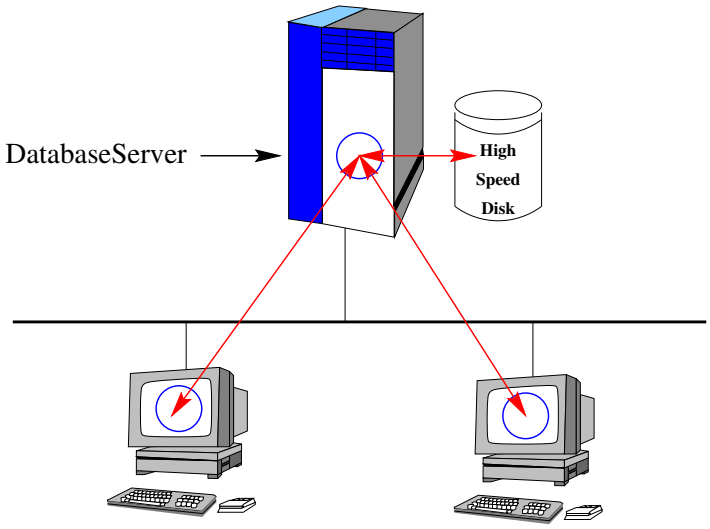
3.2 Shared file

A GUI application runs on the user's PC, and reads and writes a shared disk for files or (in a very general sense) databases. Note that Microsoft Access programs may be of this sort, but that Access files are generally not shareable.



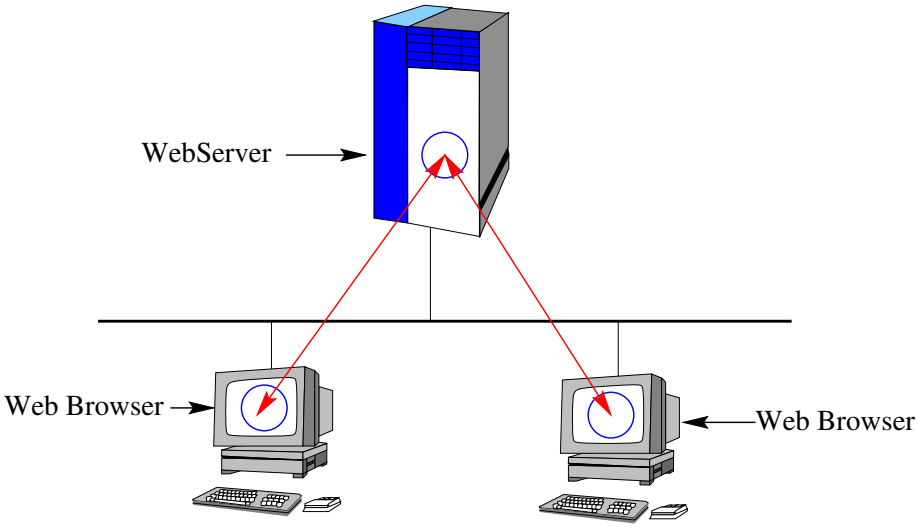
3.3 Shared database

A GUI application runs on the user's PC, and reads and writes a local database. Note that when your application uses a database like Oracle or Microsoft SQL server, it is likely to be of this sort.



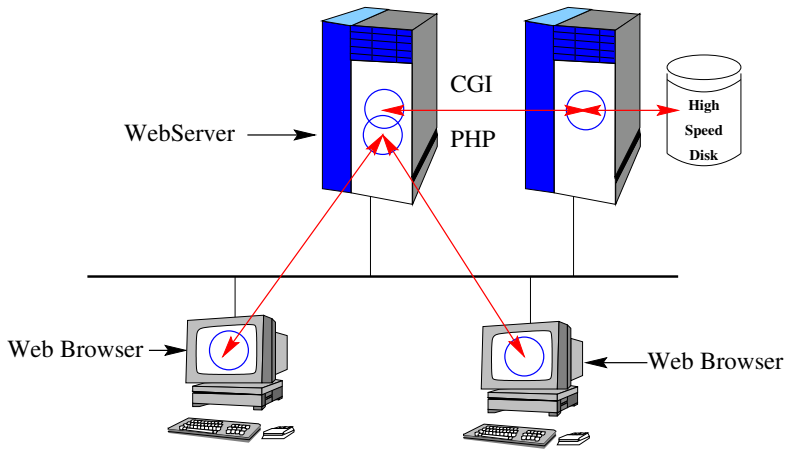
3.4 Web server applications

A VERY simple GUI application might be constructed using a series of interlinked web pages found on a server, and relying on a web browser on the client PCs. Help documentation applications are sometimes of this sort.



3.5 Web server with active scripting

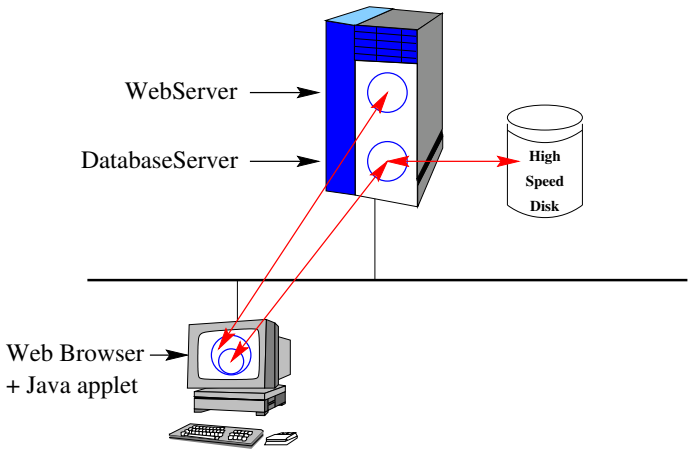
A more complex GUI application might be constructed using a series of interlinked web pages found on a server, and relying on a web browser on the client PCs. However, by using the Common Gateway Interface, or some other scheme of server-side scripting, we can return a program-derived web page to the application. CGI and PHP are both examples of this technique.



3.6 Web server with Java applet

An even more complex GUI application might be constructed using a series of interlinked web pages containing Java applets. The advantage of this, is two fold.

1. The processing load on the web server may be reduced.
2. The Java applet can directly[1] communicate with a database server.



[1]Note that there are some security concerns here.

3.7 Summary of topics

In this module, we introduced the following topic:

- GUI application architectures

Questions for Module 3

1. Characterize each of the programs you have written since you started studying at NUS.
2. Find an example of a site which is using PHP/MySQL with a large database. Give the URL, and a brief note on the site (size of database, type of user interface...).

Further study

- Pressman

Chapter 4

First steps in GUI programming

As we will discover, there is no one standard for GUI programming, although the techniques that you learn in one standard are generally transportable to another. In elementary programming styles, there is a single thread-of-control, which we can examine by reading the main(). This code determines how a user is expected to interact with the program. This general program architecture is satisfactory for small programs with simple command-line user interfaces.

However, graphical user interfaces have a much more complex thread-of-control. For example, at any time we may have a number of possible events about to occur - the user may ask for the window to be minimized, or resized, or click a button, or select a menu, or ...

Our programs must respond to each of these events. The normal way to do this is by restructuring our programs as a group of functions - each of which responds to an event. These functions are called *callbacks*.

Our GUI mainline code looks like this:

CODE LISTING **GUICode.c**

```
#include <any GUI header files needed>

int
main ()
{
    RegisterAllCallbacks ();
    LoopForever ();
}
```

An interesting area of GUI programming is the development of abstract windowed environments, where we program using an abstract API. These systems often allow the development of software which can be compiled for any environment.

4.1 How not to do GUI programming

Don't do it the hard way!

4.1.1 Direct calls to the X API

It is possible to write applications that use the X APIs directly. These programs tend to be long (that is - they have a lot of source lines). Here is a simple application:

The source is as follows:

CODE LISTING **xaw1.c**

```
#include  <stdio.h>
#include  <X11/Intrinsic.h>
#include  <X11/StringDefs.h>
#include  <X11/Xaw/Command.h>
#include  <X11/Xaw/Paned.h>
#include  <X11/Xaw/Label.h>

void
quit_callback (widget, client_data, call_data)
     Widget widget;
     caddr_t client_data;
     caddr_t call_data;
{
    exit (0);
}

main (argc, argv)
     int argc;
     char *argv[];
{                                /* main */
    Widget parent;
    Arg args[20];
    int n;
    Widget pane_widget, quit_widget;
    Widget label_widget;

    /* Set up top-level shell widget */
    parent = XtInitialize (argv[0], "Xaw1", NULL, 0, &argc, argv);
    /* Set up pane to control whole application */
    n = 0;
    pane_widget = XtCreateManagedWidget ("pane",
                                         panedWidgetClass, parent, args, n);
    /* Set up command widget to act as a push button */
    n = 0;
    quit_widget = XtCreateManagedWidget ("quit",
                                         commandWidgetClass,
                                         pane_widget, args, n);
    /* Set up a callback function */
    XtAddCallback (quit_widget, XtNcallback, quit_callback, (caddr_t) NULL);
    /* Set up label widget */
    n = 0;
    XtSetArg (args[n], XtNlabel, "This is a label.");
    n++;
    label_widget = XtCreateManagedWidget ("label",
                                          labelWidgetClass,
                                          pane_widget, args, n);
    /* Map widgets and handle events */
    XtRealizeWidget (parent);
    XtMainLoop ();
}
```

On a UNIX system we can compile this in the following way:

```
gcc -o xaw1 xaw1.c -lXt -lXaw
```

This programming technique is only included to demonstrate the underlying graphics primitives.

4.1.2 Direct calls to the Win32 API

It is also possible to write applications that use the Win32 API directly. These programs also tend to be long. We start with this program:

CODE LISTING **SimpleWin32.c**

```
#include <windows.h>

int STDCALL
WinMain (HINSTANCE hInst, HINSTANCE hPrev, LPSTR lpCmd, int nShow)
{
    MessageBox (NULL, "Hello, Windows!", "Hello", MB_OK);
    return 0;
}
```

This small example may be compiled using CYGWIN as follows:

```
gcc -oSimpleWin32 SimpleWin32.c -mwindows
```

And it produces the following application:

Another more complex example shows the flavour of raw Win32 programming, although I have removed about 380 lines for clarity:

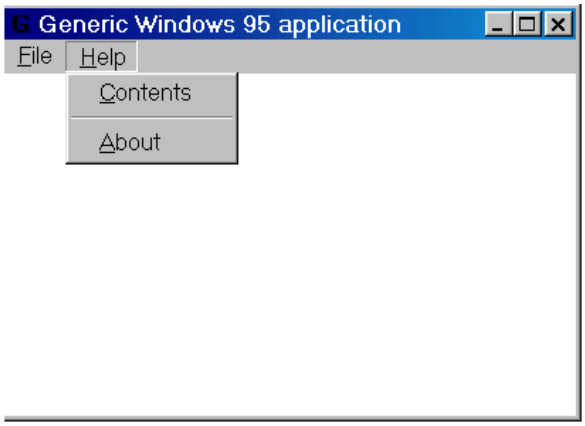
CODE LISTING **BiggerWin.c**

```
#include <windows.h>
#include <string.h>

int STDCALL
WinMain (HINSTANCE hInst, HINSTANCE hPrev, LPSTR lpCmd, int nShow)
{
    HWND hwndMain;              /* Handle for the main window. */
    MSG msg;                    /* A Win32 message structure. */
    WNDCLASSEX wndclass;        /* A window class structure. */
    char *szMainWndClass = "WinTestWin";
    memset (&wndclass, 0, sizeof (WNDCLASSEX));
    wndclass.lpszClassName = szMainWndClass;
    wndclass.cbSize = sizeof (WNDCLASSEX);
    wndclass.style = CS_HREDRAW | CS_VREDRAW;
    wndclass.lpfnWndProc = MainWndProc;
    wndclass.hInstance = hInst;
    wndclass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
    wndclass.hIconSm = LoadIcon (NULL, IDI_APPLICATION);
    wndclass.hCursor = LoadCursor (NULL, IDC_ARROW);
    wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
    RegisterClassEx (&wndclass);
    hwndMain = CreateWindow (szMainWndClass, "Hello", WS_OVERLAPPEDWINDOW,
                             CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
                             CW_USEDEFAULT, NULL, NULL, hInst, NULL);
    ShowWindow (hwndMain, nShow);
    UpdateWindow (hwndMain);
    while (GetMessage (&msg, NULL, 0, 0)) {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
    return msg.wParam;
}
```

The full source code and a makefile is available at http://www.comp.nus.edu.sg/~cs3283/ftp/generic.tgz.

When this is compiled, we get this:



Unfortunately, the full source code totals over 400 lines of C, and still doesn't actually do anything! If you are interested in learning more about Win32 programming, there is an interesting tutorial at http://www.winprog.org/tutorial.

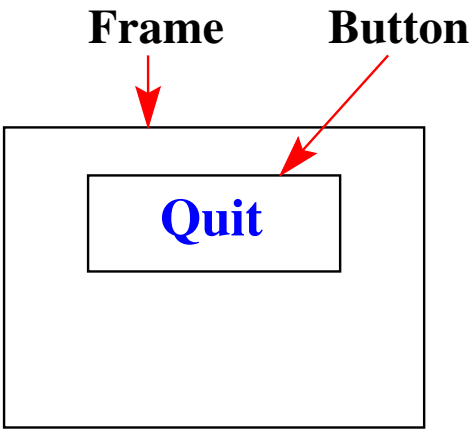


Figure 4.1: Nested GUI components

4.2 OO GUI toolkits

If we can reduce the size of code, we can be more assured that our code is correct. We are already using one mechanism to do this - we call Win32 or C functions which do quite complex things such as the call to **CreateWindow()** in the Win32 code above. A view of this might be that we are *hiding* the difficult parts from our application.

However, this sort of functional hiding has some flaws - our programs must maintain various sets of data representing our GUI environment, and can easily cause errors. Object-oriented technology provides a better mechanism, hiding the data from our programs - so that the only way in which the programs can modify the data is by using "approved" routines.

Unfortunately, there is no one object-oriented standard for GUI applications, leading to a fragmented situation - there are literally hundreds of commercial and free OO GUI framework/toolkits, with no one clear leader in the market place.

4.2.1 Event handling

When using OO GUI toolkits, the visible GUI part of the application often contains nested components as in Figure 4.1.

Each component may be the current focus of an event, and may have an event handler. If the inner component does not handle the event, then the event is passed up to the containing component.

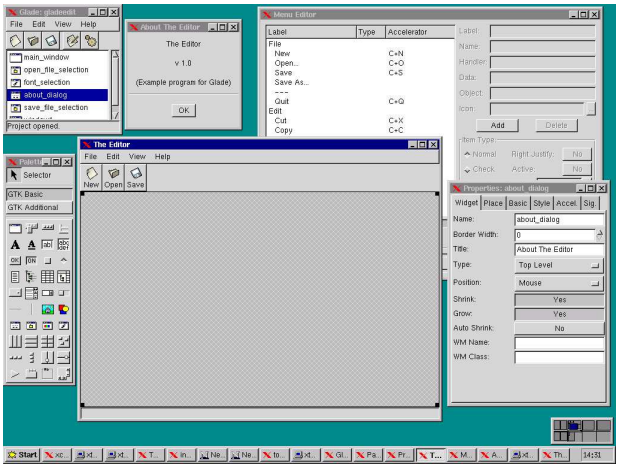
This general structure maps easily onto OO software architectures, when the innermost components are specializations (i.e. inherit from) the outer components. If you overide the event handling method in the inner component, then it will handle the event. If not, the parent component method handles the event.

4.2.2 GTK+ and glade

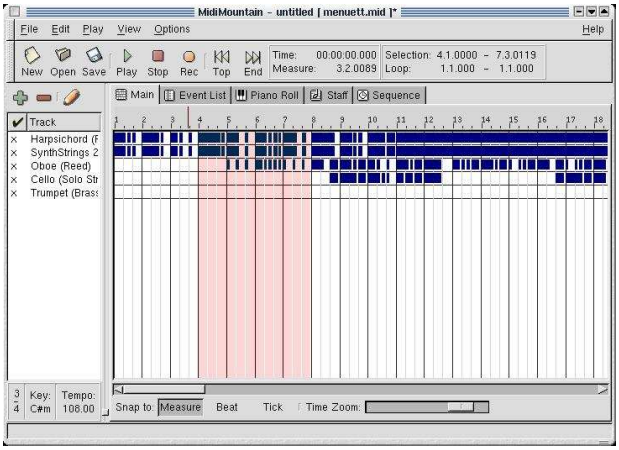
GTK+ is a multi-platform toolkit for creating graphical user interfaces, originally designed for the X Window System. However, by compiling using the CygWin GNU compiler, it is possible to use GTK+ on Win32. GTK+ is free software and part of the GNU Project. GTK+ has an object-oriented architecture for maximum flexibility, consisting of the following component libraries:

- GDK - A wrapper for low-level windowing functions.
- GTK - An advanced widget set.

One of the parts of the GTK+ distribution is a GUI builder called **glade**, which can build user interfaces very quickly.



Here is an example of an application built using **glade**:



4.2.3 MFC

The Microsoft Foundation Classes are an OO toolkit used to access Win32 system calls, and especially to construct GUI applications. A DLL contains the code for MFC, and is normally linked at runtime.

The base class is CObject, and all MFC classes inherit from this class.

One characteristic of MFC programs is the use of Hungarian (prefix) notation for variable names. It is common to see MFC program variables prefixed with type identifiers. For example:

- **dLocalMax** is a double variable
- **iLocalMin** is an integer variable.

4.2.4 Java/Swing

Originally the graphical toolkit for Java was AWT, the Abstract Windowing Toolkit. It is fairly primitive, and the new Swing toolkit provides much more functionality. AWT is native code, with a Java API, but Swing is implemented on-top-of AWT.

Swing components inherit from **java.awt.component**, and the Swing classes that are similar to AWT classes are prefixed with the letter "**J**". For example, the AWT **Button** class is renamed **JButton**. You can mix-and-match AWT and Swing components.

Java/Swing may be used in two distinct ways:

1. Producing a standalone application.
2. Producing an applet to run within a web browser.

One of the features of Swing is that it implements a pluggable look-and-feel. The look-and-feel can even be changed dynamically.

4.3 Web interfaces

It is possible to develop sophisticated applications with a web-based interface. We might divide the methods into the following categories:

- **Server-side dynamic pages** - using (for example) the CGI (Common Gateway Interface) to execute small programs or scripts on the server. This method is very common, and programs are typically written in perl.

- **Server-side scripting** - using (for example) PHP3 pages, and a specialized server which can interpret the PHP.
- **Client-side scripting** - using (for example) Javascript, and an interpreter on the client machine.
- **Client-side applets** - using (for example) a Java applet, precompiled and executed on a JVM interpreter on the client machine.

We will look at some of these methods later in the course.

4.4 Scripting languages

Scripting languages which can produce GUI interfaces are relatively easy to use. An effective strategy for building GUI applications is to write the GUI part in a scripting language, and to write the core 'difficult' part in C.

4.5 Summary of topics

In this module, we introduced the following topics:

- Programming styles to avoid
- Event driven architectures
- OO toolkits
- Web-based systems
- Scripting languages

Questions for Module 4

1. In the code listing on page 36 are a series of library calls to the **libXaw** library. In which call does the program spend the most time?
2. In this same code listing, draw the relationship between **parent**, **pane_widget**, **quit_widget** and **label_widget**.
3. In Figure 3.1, if a mouse was clicked over the **Button**, in what order would the event be processed by the **Frame** and **Button** event handlers?
4. What is a (Microsoft) DLL?
5. Find the code for a minimal Swing "HelloWorld" application (and check that it works).

Further study

- http://www.public.asu.edu/~tobiazz/papers/thesis/local/gui_toolkit_list.html

---