MacTech MAGAZINE

WEBTECH

By Aaron Montgomery, Valparaiso, IN

Threaded ACGIs in PowerPlant

The Monsterworks' Framework

About the author...

Aaron Montgomery is a mathematics professor at Purdue University North Central. He has been programming as a hobby since 1996 and in his spare time he also enjoys being with his family and mountain biking. Currently he is working on upgrading the shareware text-editor Alpha. You can contact Aaron at <agm@purduenc.edu>.

INTRODUCTION

This article describes the Monsterworks' CGI Framework. The framework arose out of work I did implementing a distance-learning course at Purdue University North Central. I had written Java applets to allow students to take quizzes over the Internet and there was a need for a CGI application to collect the student responses. While other frameworks for CGI applications are available (see the references at the end of this article), the Monsterworks' Framework provides the following advantages:

- It is written in C++ (instead of C).
- It builds on the PowerPlant framework with which you may already be familiar.
- It provides threaded request handling with little effort by the user.

This article assumes that the reader has some familiarity with the C++ language and the PowerPlant application framework. Some background with CGI applications, the C++ container classes and threading might also be helpful since the framework uses these, but you might be able to get what you need from examining the sample projects. I provide references that cover these topics at the end of the article. You will need to have a Macintosh Web server to use the applications produced by the framework. If you have a Macintosh computer with a permanent IP address, then there are a number of free Web server software packages (e.g., Quid Pro Quo).

You may download the source and project files of the Monsterworks' Framework from <http://faculty.purduenc.edu/agm/web/cgi.html>. The two projects described below and the project for building Server Proxy (described in the debugging section) are there now. Currently, work is underway to provide a Web interface to the SQL database provided by dtF. You will be able to download that project at the same location when it is complete. This framework requires Apple's Universal Headers (version 3.2), the PowerPlant framework (version 1.9.3) and the C++ standard library (MSL version 4.1.05).

The next section, *The Setting: CGI Basics*, will provide a brief introduction to CGI programming which should be sufficient to allow you to understand the remainder of the article. Following this, the article presents some facilities the framework offers in *The Characters of Monsterworks' Framework*. Once you have been introduced to the basics of the framework the section *Act One: CGI Base* presents a simple CGI program that I built using the framework. Following this is an *Intermission: Debugging and Error Handling* where the article discusses some issues in building CGI programs with the framework. The next section, *Act Two: CGI Mailer*, presents a more complicated example of a CGI program. The last section, *Behind the Scenes*, then describes some of the code which implements the framework. The article concludes with a number of references on the topic of CGI programming and PowerPlant.

THE SETTING: CGI BASICS

This section provides a quick survey of CGI programming as well as some of the details of CGI programming on the Macintosh. If you are already familiar with CGI programming, you can skip to the next section. You can find a complete discussion of this subject in the references provided at the end of this article.

The primary task of a Web server is to retrieve files for a Web browser. However, Web servers treat some types of documents differently. When a Web browser requests a CGI application, the server does not just return the contents of the file. Instead, it launches the application and passes it information about the request. The application executes and returns a response to the Web server and the server passes the response back to the Web browser as if it were the contents of the file. To perform as a CGI application, an application must accomplish three tasks: decode the data from the server; perform application specific processing; return data to the server.

The actual data transfer between the Web server and the CGI application is platform dependent and Macintosh machines use `AppleEvent`s. The Web server will break the data down into a number of different CGI variables and the enumeration `ECGIKeys` (found in `UtCGIKeys.h`) lists the names of the variables available to CGI applications working with a WebSTAR style server. The three most commonly used CGI variables are `ECGIKey_POST`, `ECGIKey_SEARCH` and `ECGIKey_PATH`. The descriptions of these variables will use the following URL:

<http://www.server.com/app.cgi$path%20data?keyword+another>

`ECGIKey_POST:` A Web browser generates this variable in response to an HTML `<FORM>` tag with the "post" action or a Java applet can send data using this variable. This variable can be up to 32K in length (longer than any other CGI variable).

`ECGIKey_SEARCH:` A Web browser generates this variable in response to an HTML `<ISINDEX>` tag or in response to an HTML `<FORM>` tag with the "get" action. Alternatively, the HTML author can include it in the URL following a question mark. In the sample URL, the `ECGIKey_SEARCH` variable would contain the string "keyword+another".

`ECGIKey_PATH:` The HTML author can include this variable in the URL following a dollar sign. In the sample URL, the `ECGIKey_PATH` variable would contain the string "path%20data".

The `ECGIKey_POST`, `ECGIKey_SEARCH` and `ECGIKey_PATH` variables will arrive at the CGI application as www-url-encoded strings. The following is a loose (and slightly inaccurate) description that will suffice for this article, please see the references for a complete (and

accurate) description. The encoding specifies that most non-alphanumeric characters should be converted to three-character codes of the form "%XX" where the XX is the hexadecimal ASCII value for the character. In the example above, "path%20data" is the encoded form of "path data". For <ISINDEX> data, the data will be a sequence of encoded keywords separated by unencoded '+' characters. In the example above, the ECGIKey_SEARCH variable has two keywords: "keyword" and "another". For <FORM> data, the data will be a sequence of key-value pairs where unencoded '=' characters separate the keys from the values and unencoded '&' characters separate the pairs. For example, the string "Thatcher=5%20years&Conor=2%20months" has two pairs: "Thatcher" associated with "5 years" and "Conor" associated with "2 months".

You can find descriptions of the remaining CGI variables in Appendix C of the WebSTAR 3 Manual. WebSTAR servers encode all of their CGI variables as strings with two exceptions: the ECGIKey_CONNECTION variable (an SInt32); and the ECGIKey_DIRE variable (an FSSpec).

THE CHARACTERS OF MONSTERWORKS' FRAMEWORK

This section will present the classes of the Monsterworks' Framework that we will use in the following examples. If you are eager to see actual code, you can skip to the next section, *Act One: CGI Base*, and deduce the roles from the uses of the classes in the code. The descriptions below do not replace the information found in the Monsterworks' API or the header files included with the projects, but should allow you to understand the sample projects.

The Monsterworks' Framework will handle the first and third tasks of a CGI application: decoding data from the server and returning data to the server. The three classes in the CCGIThreadApp component handle most of the work: CCGIThread, CCGIApp and CCGIFactory. The CCGIThread object is responsible for handling the CGI requests. The CCGIApp object is responsible for normal Macintosh application routines. The CCGIFactory class is responsible for thread management.

Before introducing the classes individually, we discuss the pervasive CheckMe method. This public virtual method checks the internal structure of an object and throws an exception if it finds something wrong. Despite their virtual declaration, all other class methods bind the method at compile time (see Listing 1 for examples). The discussion of CCGIMailApp's constructor in *Act Two: CGI Mailer* explains the need for this binding.

CGI Request Handling & CCGIThread

To use the framework you must implement the CCGIThread's abstract HandleWWWEvent method in a subclass. For simple CGI applications, this subclass and a small adjustment of main is all that is needed to have a working CGI application. The code implementing CCGIThread is in CCGIThreadApp.cp and the article presents some of it in Listing 5.

The CCGIThread's Parameter method handles the first task of a CGI application (decoding data sent from the server). Use Parameter(ECGIKeys) when the data type is a string and ECGIKeys includes the desired CGI variable. Since the AppleEvent containing the data is not directly available to the thread, Parameter(AEKeyword, ...) provides an inline wrapper of AEGetParamPtr. This article will use Parameter(ECGIKeys) exclusively since we won't need to access any non-string data.

The class `UtWWWCodec` provides methods to support the decoding of CGI variables. The method `FromWWW` will decode an www-url-encoded string and will work with either a C++ `string` class or a PowerPlant `LStr255`. Since decoding a www-url-encoded string will not cause the variable to lengthen, `FromWWW` will not truncate its return value when applied to an `LStr255`.

The `UtWWWCodec` method `StringToPairs` will decode the key-value pairs from an HTML `<FORM>` into a `FormPairs` object (a `multimap<string, string>`). Similarly, the method `StringToWords` will decode the keywords from an HTML `<ISINDEX>` query to a `QueryWords` object (a `set<string>`). Even if you have no previous experience with these C++ container classes, you can use the code in the examples below as a guide on iterating through the contents of these containers as if they were linked lists.

`HandleWWWEvent` handles the second task of a CGI application (processing the data). As stated above, you will need to implement a version of this in your subclass. `HandleWWWEvent` returns a `void*` and this value will be the return value from the thread's `Run` method. The framework does not use this value so you may return `nil`. The `CCGIThread`'s `Run` method will catch any uncaught exception escaping `HandleWWWEvent`.

The `SetReply` method handles the third task of a CGI application (returning data to the server). Usually the reply will be the text of an HTML document (along with a HTTP header) which the server will return to the client's browser. As in the case of the `Parameter` method, `SetReply` has two versions, the first, `SetReply(AEKeyword, ...)`, is an inline wrapper for the Macintosh Toolbox call `AEPutParamPtr`. You can use the second, `SetReply(const string&)`, if you construct your reply as a C++ `string` object. This article will use `SetReply(const string&)` exclusively since our replies will all be C++ `string` objects.

The class `UtHTMLWriter` provides methods to support the generation of HTML documents. The method `HTMLHeader` provides a standard HTTP header followed by an HTML header and the opening `<BODY>` tag. The first argument provides the title for the HTML document and the other arguments further customize the page. The method `HTMLFooter` provides the closing `</BODY>` tag as well as the closing `</HTML>` tag. The `Redirect` method takes a single string which should be a complete URL and will generate an HTML page which redirects the browser to this URL.

Application Routines & `CCGIApp`

The `CCGIApp` is a complete class and you can use it directly (see the first example) or you can use a subclass to provide application-wide functionality such as preferences (see the second example). The code implementing `CCGIApp` is in `CCGIThreadApp.cp` and the article presents some of it in Listing 6.

The `CCGIApp` constructor will terminate the application if you call it twice (it is a singleton in the terminology of Gamma, et. al. 1995). The `CCGIApp` destructor is `protected` and you must `delete` the `CCGIApp` with a call to `CCGIApp`'s `DeleteApp` method. Together, these insure that there is at most one `CCGIApp` in a program and that it was allocated by a call to `new`. You can access the single `CCGIApp` through the `static` method `CCGIAppP` which returns a pointer to a `CCGIApp`. This method will return a non-`nil` pointer between the call to `CCGIApp`'s constructor and the call to `CCGIApp`'s `DeleteApp` method. The `CCGIAppP` method will throw a `nil`-pointer exception if you call it elsewhere.

The use of a singleton arises from the following problem. `CCGIApp` must install an `AppleEvent` Handler and the compiler will need to know the address of this handler (so it must be a `static` method). However, subclasses of `CCGIApp` should be capable of overriding this handler (hence it should also be `virtual`). The `static` pointer solves this problem by allowing the installed `AppleEvent` handler (`WWWHandler`) to call a virtual function (`HandleWWWEvent`) through the pointer.

`CCGIApp` uses a preference file to determine whether it will use a log file as well as the name of the log file. The method `Log(const LStr255&)` writes data to the log file (if the preferences indicate that the application should log its activities). `Log(const LStr255&)` will update the application's window logs the application name, the date and time and the argument to the method.

The method `LogError(const LStr255&, bool)` handles errors. `LogError(const LStr255&, bool)` first passes the `LStr255&` to the `Log` method and will indicate an error in the CGI application's window. If the `bool` argument is `true`, the error is fatal and the `CCGIApp` calls its `DoQuit` method and then yields to the thread containing the `main` function. This will result in application termination.

Thread Management and `CCGIFactory`

The `CCGIFactory` class is responsible for generating the `CCGIApp` to be used by the application as well as managing all the `CCGIThread`s. The source implementing `CCGIFactory` is in `CCGIThreadApp.cp` and the article presents some of it in Listing 7.

The `CCGIFactory` class is a singleton (so you cannot construct two of them) and its destructor is `protected` (so you will use `DeleteFactory` to `delete` it). As in the case of the `CCGIApp` class, access to the single `CCGIFactory` is through a `static` method: `CCGIFactoryP`. The only other method you will need to use directly is the `CreateApp` method you will use to create a `CCGIApp`. The framework uses the other methods in CCGIFactory internally.

Because `CCGIFactory` has two abstract methods, you cannot use the class directly, but the framework provides a template class with default implementations for these methods and we will use the template in both of our examples. The template takes three parameters: a `CCGIThread` type, a `CCGIApp` type and a `CCGIFactory::CCGITerminator` type. The template requires the `CCGIThread` type because `CCGIThread` is an abstract class. The `CCGIApp` type defaults to a `CCGIApp`. We will use this default in *Act One: CGI Base* and we will provide our own `CCGIApp` type in *Act Two: CGI Mailer*. The framework uses `CCGIFactory::CCGITerminator` internally and you should use the default unless you are customizing the framework's thread management scheme.

In most cases the template class suffices, but there are a number of situations where you might want to write your own class. Because the `CCGIFactory` has access to all the Web server data, it could generate different thread types based on this data. For example, if your CGI application is acting as a front-end to a database, the `CCGIFactory` could use a CGI variable to determine whether to create a thread designed to retrieve data or a thread designed to update data. As another example, a `CCGIFactory` could assign different priorities to the threads it creates based on the IP address of the browser (you could assign in-house requests a higher priority). In both cases the switching seems to be peripheral to the processing of the event or standard application handling and so doesn't belong in the `CCGIThread` or `CCGIApp` classes.

ACT ONE: CGI BASE

We are now ready to generate our first CGI application. The "Hello World" application of the CGI world is the echo application that generates an HTML page listing the data that the Web server presented to CGI application. The `CGI Base` project builds this application (and works well as stationary for other CGI projects).

This application tests some of the framework's features, in particular, it should: return all the information posted; allow threading to occur; handle fatal and non-fatal errors correctly. You can test the application using the `ECGIKey_PATH` variable. If the `ECGIKey_PATH` variable is a positive integer, the `CCGIEchoThread` handling the request will sleep for that many seconds before continuing. If the variable is a negative integer between -1 and -9, then the `CCGIEchoThread` will log an error and then sleep for the negative of the value in seconds (e.g., a -7 value results in a sleep of 7 seconds). If the variable is a negative integer less than -9, then the `CCGIEchoThread` will log a fatal error and the application will terminate.

To get this application running we need to subclass `CCGIThread` and write the function `main`. The code implementing `CCGIEchoThread` is in `CCGIEchoThread.cp` and the article presents all of it in Listing 1. The code implementing `main` is in `main.cp` and the article presents the code for the function `main` in Listing 2.

The constructor of the `CCGIEchoThread` passes everything to `CCGIThread`'s constructor. It then uses `CCGIApp`'s `Log` method to log the thread's creation and the destructor logs the thread's destruction. The log entries will include the addresses of the object so that we can verify that the application permits later threads to start processing before the completion of earlier threads.

There are two utility methods in `CCGIEchoThread` which are used to format the CGI variable values: `H2Echo` and `LineEcho`. The only difference between them is that `H2Echo` will produce a `<H2>` HTML heading while `LineEcho` places its output with the heading indicated by a `<STRONG>` tag.

The `HandleWWWEvent` method does the CGI processing. To get the most out of threading, you should call `Yield` frequently in your `HandleWWWEvent` method. The `HandleWWWEvent` method presented below checks the `ECGIKey_PATH` variable to determine if the thread should sleep or log an error. Then it goes through the CGI variables and writes an HTML page presenting their values. `HandleWWWEvent` formats the `ECGIKey_POST` variable in two styles: as a raw string or as the result of a `<FORM ACTION = "post">` request. Similarly, it formats the `ECGIKey_SEARCH` variable in three styles: as a raw string, as the result of a `<FORM ACTION = "get">` request and as the result of an `<ISINDEX>` request.

Listing 1: CCGIEchoThread.cp

CCGIEchoThread

```
CCGIEchoThread::CCGIEchoThread(
  const AppleEvent& inEvent,
  const AppleEvent& outReply,
  long inRefCon)

: CCGIThread(inEvent, outReply, inRefCon)
{
  CCGIThread::CheckMe();
```

```
  //Add a log entry to the log file
  LStr255 theLogEntry = "Starting Thread @ ";
  theLogEntry += reinterpret_cast<long>(this);
  CCGIApp::CCGIAppP()->Log(theLogEntry);
}
```

~CCGIEchoThread

```
CCGIEchoThread::~CCGIEchoThread(void)
{
  try
  {
    //Add a log entry to the log file
    LStr255 theLogEntry = "Ending Thread @ ";
    theLogEntry += reinterpret_cast<long>(this);
    CCGIApp::CCGIAppP()->Log(theLogEntry);
  }
  catch (...)
  {
  }
}
```

H2Echo

```
string CCGIEchoThread::H2Echo(
  const string& inHeader,
  ECGIKeys inParameter) const
{
  CCGIEchoThread::CheckMe();

  //generate a <H2> heading from first argument
  string theResult = "<H2>" + inHeader + "</H2>\r\n";

  //add the value of the appropriate CGI parameter
  theResult += Parameter(inParameter);

  //and a new line
  theResult += "\r\n";

  //return the string
  return theResult;
}
```

LineEcho

```
string CCGIEchoThread::LineEcho(
  const string& inHeader,
  ECGIKeys inParameter) const
{
  CCGIEchoThread::CheckMe();

  //generate a strong heading from the first argument
  string theResult = "<STRONG>" + inHeader +
"</STRONG>\r\n";
```

```
  //add the value of the appropriate CGI parameter
  theResult += Parameter(inParameter);

  //add a <BR> and a new line
  theResult += "<BR>\r\n";

  //return the string
  return theResult;
}
```

HandleWWWEvent

```
void* CCGIEchoThread::HandleWWWEvent(void)
{
  CCGIEchoThread::CheckMe();

  //get the PATH CGI variable
  string theWaitStr = Parameter(CGIKey_PATH);
  const int kBaseTen = 10;
  //strtol will return 0 if theWaitStr is not an integer
  long theWait = strtol(theWaitStr.c_str(), nil, kBaseTen);
  if (theWait < 0)
  {
    //test the error logging facility
    bool isFatal = (theWait <= -10);
    LStr255 theErrorString
      = StringLiteral_("Negative Wait Time");
    CCGIApp::CCGIAppP()->LogError(theErrorString, isFatal);
    //convert wait to positive
    theWait = -theWait;
  }

  //simulate a lengthy request
  Sleep(theWait*1000);

  //generate the HTML header for the reply page
  string theReply = UtHTMLWriter::HTMLHeader("Your
Request");

  //play nicely
  Yield();

  //generate a <H2> section
  //returning the POST CGI variable as a raw string
  theReply += H2Echo("Post", CGIKey_POST);

  //generate a <H2> section
  //returning the POST CGI variable as a <FORM ACTION="post"> request
  theReply += "<H2>Post As Form Submission</H2>\r\n";
  FormPairs thePairs
    = UtWWWCodec::StringToPairs(Parameter(CGIKey_POST));
  for (FormPairs::iterator theIterator = thePairs.begin();
      theIterator != thePairs.end();
```

```
    ++theIterator)
{
  theReply += theIterator->first;
  theReply += "<FONT COLOR=\"#FF0000\"> = </FONT>";
  theReply += theIterator->second;
  theReply += "<BR>\r\n";
}

//play nicely
Yield();

//generate a <H2> section
//returning the SEARCH CGI variable as a raw string
theReply += H2Echo("Search", CGIKey_SEARCH);

//generate a <H2> section
//returning the SEARCH CGI variable as a <FORM ACTION="get"> request
theReply += "<H2>Search As Form Submission</H2>\r\n";
thePairs
  = UtWWWCodec::StringToPairs(Parameter(CGIKey_SEARCH));
for (FormPairs::iterator theIterator = thePairs.begin();
    theIterator != thePairs.end();
    ++theIterator)
{
  theReply += theIterator->first;
  theReply += "<FONT COLOR=\"#FF0000\"> = </FONT>";
  theReply += theIterator->second;
  theReply += "<BR>\r\n";
}

//generate a <H2> section
//returning the SEARCH CGI variable as an <ISINDEX> request
theReply += "<H2>Search As Query Submission</H2>\r\n";
QueryWords theKeywords
  = UtWWWCodec::StringToWords(Parameter(CGIKey_SEARCH));
for (QueryWords::iterator theIterator
      = theKeywords.begin();
    theIterator != theKeywords.end();
    ++theIterator)
{
  theReply += *theIterator;
  theReply += "<BR>\r\n";
}

//play nicely
Yield();

//generate a <H2> section
//returning the PATH CGI variable as a raw string
theReply += H2Echo("Path", CGIKey_PATH);
theReply += "\r\n";
```

```
    //play nicely
    Yield();

    //generate a <H2> section to display the user information
    theReply += "<H2>Client Information</H2>\r\n";
    theReply += LineEcho("User", CGIKey_USER);
    theReply += LineEcho("Pass", CGIKey_PASS);
    theReply += LineEcho("Address", CGIKey_CLIENT);

//code omitted
//we go through all of the CGI variables
//that are strings and append them to theReply

    theReply += LineEcho("Action", CGIKey_ACTION);
    theReply += LineEcho("Action Path", CGIKey_ACTION_PATH);

    //play nicely
    Yield();

    //generate the HTML footer for the reply page
    theReply += UtHTMLWriter::HTMLFooter();

    //play nicely
    Yield();

    //return the HTML page to the server
    //which will send it to the browser
    SetReply(theReply);

    //we don't need to pass info to other threads
    //so we return nil from this method
    return nil;
}
```

Before turning our attention to `main`, let me point out that there is one typedef in `CCGIEchoThread.h` which we will use.

```
typedef  TCGIFactory<CCGIEchoThread>CCGIEchoFactory;
```

Although you could just use a template directly in `main`, the `typedef` is worthwhile for two reasons: first, it saves the you some typing (especially if you later change your definition) and second, it guarantees that the template parameters are compatible. The second reason will become especially important in the next example (*Act Two: CGI Mailer*) where our `CCGIThread` subclass relies on specialized features of our `CCGIApp` subclass.

The `main` function initializes the Macintosh Toolbox, creates a `CCGIEchoFactory` and uses the factory to produce a `CCGIApp`. Notice that we do not need to store the locations of these objects since we have access to them through `static` methods. Then `main` runs the application inside a `try` block (unlike `LApplication`s, `CCGIApp`s do not catch all exceptions in their `Run` method). After the application is through running, we call `DeleteApp` and `DeleteFactory` to clean up.

Listing 2: main.cp

main

```
int main(void)
{
   //standard toolbox initialization
   InitializeHeap(3);
   UQDGlobals::InitializeToolbox(&qd);

   //create a factory (actually a CCGIEchoFactory)
   //this is probably the only line of code
   //you will need to modify
   //
   //there is only one and we can always
   //get to it through the static pointer,
   //we don't even need to use a variable
   //
   //NEW is a debugging macro defined by DebugNew.h
   //it calls new
   NEW CCGIEchoFactory();
   CCGIFactory::CCGIFactoryP()->CheckMe();

   //create an app (the factory determines which type exactly)
   //again, there is only one and we can always
   //get to it through the static pointer
   CCGIFactory::CCGIFactoryP()->CreateApp();
   CCGIApp::CCGIAppP()->CheckMe();

   try
   {
      //try to run the app
      //CCGIApp overrides LApplication's Run() method and
      //removes its try-block, so you will need to catch
      //exceptions here
      CCGIApp::CCGIAppP()->Run();
   }
   catch (...)
   {
      //uncaught exceptions are fatal
      const bool kIsFatal (true);
      CCGIApp::CCGIAppP()->LogError(
        StringLiteral_("Uncaught exception in CCGIApp::Run"),
        kIsFatal);
   }

   //delete the application (which will also
   //delete any running CCGIThreads)
   CCGIApp::CCGIAppP()->DeleteApp();

   //delete the factory
   CCGIFactory::CCGIFactoryP()->DeleteFactory();

   //return that everything went okay
```

```
  return noErr;
}
```

INTERMISSION: DEBUGGING AND ERROR HANDLING

Now that you can create CGI applications, you need a method to debug them. Unlike normal Macintosh applications, CGI applications respond to `AppleEvents` while running in the background and this means that you will not interact with them directly. To debug such an application, you will need to have a method to send it `AppleEvents` and the article presents three such methods in the *Debugging* subsection.

Once you have debugged your application placed it on the server, it will likely run without user interaction which means that your error handling code needs to differ from normal Macintosh error handling code (where you notify the user). You need to be particularly careful with your code because of the threaded nature of the framework. The article discusses this issue in the *Error Handling* subsection below.

Debugging

The most obvious technique for debugging is to set up your own Web server. This provides the best possible debugging environment since it debugs the CGI application in precisely the environment in which you will use it. However, there are a number of disadvantages to this approach. One disadvantage to this is that your computer will need to be on a network. If you are on a dial-up network, then you will be tying up your phone while you debug your application. Another disadvantage is that you may run into memory problems (you will need to have a browser, the Web server, the debugger and your application all running simultaneously). Yet another disadvantage is that you cannot save the POST data and so you will need to regenerate it each time.

One way to debug which avoids some of the problems with running your own server is to use AppleScript to send events to your CGI application. This does not require you to be on a network or to have many large applications open at once. Furthermore, you can save the scripts and reuse them when needed. The disadvantage is that you will need to encode the CGI variables by hand and this is tedious (and hence error prone).

To avoid some of the problems of the previous two solutions, the Monsterworks' Framework provides a small application called `Server Proxy`. To use `Server Proxy` for debugging, you launch it (which will create a new document); fill in the data; select the CGI application and then send the event. You can edit the reply from the CGI application (in particular, you should remove the HTTP header), save the reply as a text file and view the reply with a browser. `Server Proxy` can www-url-encode `<FORM>` and `<ISINDEX>` variables so that you won't have to do this by hand and `Server Proxy` is capable of saving CGI variables so you won't need to re-enter data from one session to the next.

Probably the best way to debug is to use two methods. Use either AppleScript or `Server Proxy` to get what appears to be a final version. Then exercise this version on an actual Web server to make sure that it really works.

Now that you have an idea of how to debug, you should be aware of one situation that will arise when debugging background applications built with PowerPlant. Here are instructions describing how to reproduce the situation using `Server Proxy` and `CGI Base`.

- Set the debugger to put up an alert if an exception is thrown.
- Do something to throw an exception (use `Server Proxy` to send an event to `CGI Base` with the PATH variable equal to "-10").
- Switch to the `CGI Base` by clicking on the alert box which appears.
- Dismiss the alert box.

At this point, no PowerPlant dialog box will let itself come to the front (check this by selecting `Logging…` in the `Settings` menu), but system dialog boxes will continue to work (check this by selecting `About` in the `Apple` menu). Here's an explanation and solution (provided by John C. Daub): switching to the CGI application by clicking the alert box causes the Mac OS to bring your application to the front but PowerPlant does not get notified. As a result, all the PowerPlant classes believe they are still in the background and will not activate. The remedy is to do something to let PowerPlant know that it is in the foreground and the most convenient method of doing this is to switching to another application and then switching back to your CGI application. This particular behavior will only arise in the debug builds of your application and so it shouldn't be a problem in any final build.

Error Handling

Error handling is particularly important for CGI applications because often these applications will be running unattended and their failure can completely disable the server. In particular, you will need to avoid dialog boxes as a way to handle errors since it is unlikely that there will be anyone to read them. This subsection will provide some guidance on how to write error handling code that will work in this environment.

The Monsterworks' Framework treats fatal errors severely in hopes that a quick exit by the application will result in less collateral damage. In the provided implementation, passing `true` as `LogError`'s second argument causes `LogError` to call `CCGIApp`'s `DoQuit` method. This will cause the `CCGIApp` object to exit its `Run` method and once this occurs, you should call the `CCGIApp`'s `DeleteApp` method to `delete` all `CCGIThread`s without resuming their `Run` method. In order to allow the threads to return some sort of reply to the server, the `DeleteApp` method allows each thread to run the `ShutDown` method (inherited from `CCGIThread`). You can override the `ShutDown` method to handle any actions that must occur and which would normally occur if `Run` were allowed to complete. After deleting all the running `CCGIThread`s, `DeleteApp` deletes the `CCGIApp`. After this, you can call the `CCGIFactory`'s `DeleteFactory` method and exit the application.

Because threads may be deleted before completing their `Run` method, your error handling code needs to be carefully planned. Stack based classes (PowerPlant's `St` classes and Monsterworks' `Au` classes) work well in a single-thread environment but can fail in a multi-threaded environment. The following sequence of events presents an example that could lead to problems:

- Your thread's `HandleWWWEvent` method creates a stack variable that places a lock on a database upon creation and then removes it upon destruction.
- Your thread calls `Yield` either directly or indirectly.
- The thread to which you yield encounters a fatal error and calls the application's `LogError` method and indicates a fatal error.

- Your thread's `ShutDown` method is called.
- The application exits.

Since control never returns to your thread's `Run` method, local variables are not destructed. You can solve some of these problems by storing these stack-based variables as data members of you thread.

ACT TWO: CGI MAILER

In this section we will put together a more complicated example of a CGI application than `CGI Base`. `CGI Mailer` mails data from an HTML form to users. The preference file contains the following information:

- A "to-suffix" that limits who uses the mailer. The application will append the suffix to every address to which the mailer sends information. For example, the suffix on the mailer I use is "@purduenc.edu", as a result, only people with mail accounts at Purdue University North Central can use the mailer. This measure will prevent someone from using your mailer to relay spam.
- A "from-address" that identifies the sender of the e-mail message. You could set this to the web master or an address indicating that the mail came from the CGI application.
- The address of the mail server that sends the mail.

The application will get the following information from the CGI variables:

- The address to which the server mails the data. The application will append the "to-suffix" described above onto this variable. The application will look for this data in the CGI variable associated with `ECGIKey_PATH`.
- The application will look for the subject of the message in the CGI variable associated with `ECGIKey_SEARCH`.
- Optionally, an URL to which the application will redirect the Web browser if the mailing was successful. The application will look for this data in the CGI variable associated with the `ECGIKey_POST`. Because the data is coming from an HTML `<FORM>` tag, this information will be in a name-value pair whose name is "replyPage". If no pair exists, the application will generate a default page.
- Optionally, an URL to which the application will redirect the Web browser if the mailing was unsuccessful. The application will look for this data in the CGI variable associated with the `ECGIKey_POST`. Because the data is coming from an HTML `<FORM>` tag, this information will be in a name-value pair whose name is "errorPage". If no pair exists, the application will generate a default page.

We will need to subclass both `CCGIThread` (to provide the mailer) and `CCGIApp` (to provide the preferences) but we will use a `TCGIFactory`. Since all the CGI request handling occurs within the `CCGIMailThread`, we will focus on the `CCGIMailThread` code and then discuss `CCGIMailApp`'s handling of user preferences.

CCGIMailThread

The `CCGIMailThread` overrides the `HandleWWWEvent` and introduces three helper methods. Because we are building on top of PowerPlant, we can delegate the communication

with the mail server to PowerPlant's Networking classes. The code implementing CCGIMailThread is in CCGIMailer.cp and the article presents some of it in Listing 3.

The ParseForm method parses the HTML <FORM> data and identifies the values associated with the names "replyPage" and "errorPage". ParseForm returns these values in the output arguments and formats all other name-value pairs into a single string suitable for human consumption.

The SetReplyOK method and the SetReplyERR methods generate the reply to the server. If the string passed into one of these methods is non-empty, it should be an URL to which the server will redirect the browser. If the string is empty, then each of these methods will generate a default page. The code for SetReplyOK is in Listing 3 (the code for SetReplyERR is identical except for the default string).

The HandleWWWEvent method gets the user preferences by using the static CCGIAppP pointer and a dynamic_cast to a CCGIMailApp. The preferences are then available from the CCGIMailApp's accessor functions. After getting the preferences, HandleWWWEvent uses ParseForm to decode the <FORM> data. Then the method places the formatted name-value pairs in a mail message and sends this message using PowerPlant's Networking Classes. Finally, HandleWWWEvent either calls SetReplyOK and SetReplyERR to set the CGI reply.

Although we do not directly call Yield in our HandleWWWEvent method, it is possible that some PowerPlant classes call Yield in their methods.

Listing 3: CCGIMailer.cp - Thread methods

ParseForm

```
string CCGIMailThread::ParseForm(
  string* outReplyPageP,
  string* outErrorPageP) const
{
  CCGIMailThread::CheckMe();

  ThrowIfNil_(outReplyPageP);
  ThrowIfNil_(outErrorPageP);

  string theResult;

  FormPairs theFormData
    = UtWWWCodec::StringToPairs(Parameter(CGIKey_POST));
  for (FormPairs::iterator theIterator
        = theFormData.begin();
      theIterator != theFormData.end();
      ++theIterator)
  {
    if ( theIterator->first == "replyPage")
    {
      *outReplyPageP = theIterator->second;
    }
    else if ( theIterator->first == "errorPage")
    {
      *outErrorPageP = theIterator->second;
    }
    else
```

```
    {
      theResult += "==================\r\n";
      theResult += theIterator->first;
      theResult += ":\r\n\t";
      theResult += theIterator->second;
      theResult += "\r\n\r\n";
    }
  }
  return theResult;
}
```

SetReplyOK

```
void CCGIMailThread::SetReplyOK(string* inReplyP) const
{
  CCGIMailThread::CheckMe();

  ThrowIfNil_(inReplyP);
  if (inReplyP->empty())
  {
    *inReplyP = UtHTMLWriter::HTMLHeader("Your
Submission");
    *inReplyP += "<P>Your submission has been mailed.";
    *inReplyP += UtHTMLWriter::HTMLFooter();
  }
  else
  {
    *inReplyP = UtHTMLWriter::Redirect(*inReplyP);
  }
  SetReply(*inReplyP);
}
```

HandleWWWEvent

```
void* CCGIMailThread::HandleWWWEvent(void)
{
  CCGIMailThread::CheckMe();

  string theReply;
  string theError;

  try
  {
    const CCGIMailApp* theMailAppP
      = dynamic_cast<const CCGIMailApp*>(
        CCGIApp::CCGIAppP());
    ThrowIfNil_(theMailAppP);

    const int kStringLength  (256);

    //Get Mail Preferences
    char theToSuffix[kStringLength];
    theMailAppP->GetToSuffix(theToSuffix, kStringLength);
```

```
    char theFromAddress[kStringLength];
    theMailAppP->GetFromAddress(
      theFromAddress, kStringLength);

    Str255 theMailServer;
    theMailAppP->GetMailServer(theMailServer);

    //Fill in Message
    LMailMessage theMessage;
    string theToString = Parameter(CGIKey_PATH);
    theToString += theToSuffix;
    theMessage.SetTo(theToString.c_str());

    theMessage.SetSubject(
      UtWWWCodec::FromWWW(
        Parameter(CGIKey_SEARCH)).c_str());

    theMessage.SetFrom(theFromAddress);

    string theFormData = ParseForm(&theReply, &theError);
    theMessage.SetMessageBody(
      theFormData.c_str(), theFormData.length());

    //Send the Message
    LSMTPConnection theConnection(*this);
    theConnection.SendOneMessage(theMailServer,
theMessage);

    SetReplyOK(&theReply);
  } catch (...) {
    SetReplyERR(&theError);
  }
  return nil;
}
```

This completes the discussion of `CCGIMailThread`. The Monsterworks' Framework hides the interactions with the Web server and PowerPlant hides the interactions with the mail server.

CCGIMailApp

We now turn our attention to `CCGIMailApp` which subclasses `CCGIApp` to provide preferences. I begin by quickly describing the non-CGI related methods: we use `FindCommandStatus`, `ObeyCommand`, `SetMailPreferences` to add a command to set the preferences; `GetToSuffix`, `GetFromAddress` and `GetMailServer` provide access to the preferences. The return type of the `Get` methods matches the requirements of the PowerPlant's `LMailMessage` methods. The code implementing `CCGIMailApp` is in `CCGIMailer.cp` and the article presents some of it in Listing 4.

The reason for writing a subclass was to provide preferences and the constructor will load these preferences during construction and lock them down allowing the `CCGIMailApp` to hold a pointer to them. As promised above, we explain why all the class methods bind the

CheckMe method at compile time. In the CCGIMailApp constructor we call the CCGIApp method PrefsP. If CCGIApp's call to CheckMe were bound dynamically it would resolve to the method defined by CCGIMailApp. Since the CCGIMailApp is not completely constructed, this would result in an exception being thrown. We resolve this by binding the call at compile time so that derived classes may safely call any base class method in their constructor. An alternative strategy would have been to split each method into a protected implementation and a public interface. The public interface would call CheckMe and then the protected implementation and the constructors of derived classes could call the protected version. We opted for the explicit binding for two reasons: first, it is simpler to implement and second, if the method is a base class method, only the base class portion of the object needs to be valid to complete the call. The chosen method allows the application to continue to maintain low-level functionality (such as logging) during a crisis.

Listing 4: CCGIMailer.cp - Application methods

CCGIMailApp

```
CCGIMailApp::CCGIMailApp(void)
: myMailPrefsH(),
  myMailPrefsP(nil)
{
  CCGIApp::CheckMe();

  const bool kIsFatal(true);

  //Set up Mailing Preferences
  try
  {
    StCurResFile theCurrentResFile();
    SInt16 theRefNum
      = PrefsP()->OpenResourceFork(fsRdWrPerm);

    const bool kCurResOnly(true);
    {
      StNewResource theNewResource(PrefResType, PrefResID,
        sizeof(SCGIMailPrefs), kCurResOnly);
      if (!theNewResource.ResourceExisted())
      {
        SCGIMailPrefs** theDefaultPrefsH
          = reinterpret_cast<SCGIMailPrefs**>(
            theNewResource.Get());

        //the default preferences
        LStr255 theString
          = StringLiteral_("Monsterworks Mailer");
        LString::CopyPStr(
          theString, (**theDefaultPrefsH).FromAddress,
          sizeof((**theDefaultPrefsH).FromAddress));

        theString = StringLiteral_("@purduenc.edu");
        LString::CopyPStr(
          theString, (**theDefaultPrefsH).ToSuffix,
          sizeof((**theDefaultPrefsH).ToSuffix));
```

```
        theString
          = StringLiteral_("centaur.cc.purduenc.edu");
        LString::CopyPStr(
          theString, (**theDefaultPrefsH).MailServer,
          sizeof((**theDefaultPrefsH).MailServer));
      }
    }
    const bool kThrowIfFail(true);
    myMailPrefsH.GetResource(PrefResType, PrefResID,
      kThrowIfFail, kCurResOnly);
    ::HLockHi(myMailPrefsH.Get());
    myMailPrefsP
      = *reinterpret_cast<SCGIMailPrefs**>(
        myMailPrefsH.Get());
    ValidatePtr_(myMailPrefsP);
  }
  catch (...)
  {
    LogError(
      StringLiteral_("Couldn't set up mailing
preferences"),
      kIsFatal);
  }

  //Register PowerPlant Classes
  try
  {
    RegisterClass_(LTabGroup);
  }
  catch (...)
  {
    LogError(
      StringLiteral_("Couldn't register PowerPlant
Classes"),
      kIsFatal);
  }

  //Install myself as the Static CCGIApp
  try
  {
    SetCCGIAppP(this);
  }
  catch (...)
  {
    LogError(
      StringLiteral_("Couldn't install
        CCGIMailApp as static CCGIApp"),
      kIsFatal);
  }
}
```

```
CCGIMailApp::~CCGIMailApp(void)
{
}
```

Now that we have a preference file, we need to give the user a method of accessing it. We will do this by appending an item to the `Settings` menu. We could store the name of the item in a resource, load the resource and then add the menu item to the existing menu. Instead we will override the `Menu` resource. To do this, we take the `Settings` menu resource (number 131) from `CGIBase.ppob` and added it to a new resource file named `CCGIMailer.ppob`. We can then add a new menu item (`Mailing…`) to this and give it the appropriate command identifier. If you do this, you must link `CGIMailer.ppob` into your application before linking `CGIBase.ppob`. This will happen if you place `CCGIMailer.ppob` higher in the link window than `CGIBase.ppob` as shown in Figure 1. When the linker reaches `CGIBase.ppob`, it will report that it is ignoring the duplicated menu item. Make sure that it is ignoring the one from `CGIBase.ppob` and not the one from `CCGIMailer.ppob`.

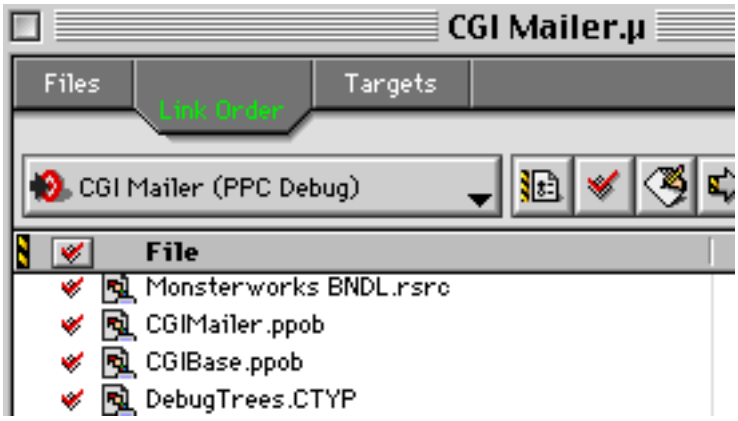

Figure 1: Link Order for CGIMailer

The final item of interest is the `CCGIMailFactory` provided with the following:

```
typedef  TCGIFactory<CCGIMailThread, CCGIMailApp>
      CCGIMailFactory;
```

Now users can use a `CCGIMailFactory` to insure that they are using compatible subclasses of `CCGIThreads` and `CCGIApps`.

main

The only change to the `main` function from the `CGI Base` project (given in Listing 2) is that the the created `CCGIFactory` is a `CCGIMailFactory` instead of a `CCGIEchoFactory`. This single change will allow for the creation of a `CCGIMailApp` in place of the `CCGIApp` and `CCGIMailThreads` in place of the `CCGIEchoThreads`.

BEHIND THE SCENES

Now that you know how to use the framework, we present some of the code behind the interface. In the next three subsections, we take a closer look at the implementations some of the code in the three primary classes: `CCGIThread`, `CCGIApp` and `CCGIFactory`. The `CCGIThread` is responsible for suspending the `AppleEvent`, processing the `AppleEvent` and then resuming the `AppleEvent`. The `CCGIApp` is responsible for handling user preferences and for dispatching the CGI `AppleEvent`s to a `CCGIThread`. The `CCGIFactory` class is responsible for thread management, both the creation of threads and their orderly destruction in the case of an error.

CCGIThread

We start with the `CCGIThread` class because it is responsible for CGI handling. The code implementing `CCGIThread` is in `CCGIThreadApp.cp` and the article presents some of it in Listing 5.

The `CCGIThread` constructor stores the `AppleEvent` information in data members because the `AppleEvent`s provided by the operating system will be invalid after we call `SuspendEvent`. Since we use stack-based classes for all the dynamically allocated members of the `CCGIThread` class, we do not need to do anything in the destructor.

The framework places the work required of all `CCGIThread`s in the `Run` method and you should override the `Run` method if they need to do some processing before calling `SuspendEvent` or after calling `ResumeEvent`. Any override of `Run` must call `SuspendEvent` to permit multi-threading of `AppleEvent` processing and `ResumeEvent` to send the event back to the server (see Grant 1996). We have placed the complete contents of the `Run` method in a `try` block to insure that your application is aware of any uncaught exceptions. If you leave out this block, `LThread` will `catch` (and then ignore) any uncaught exceptions so you can leave the block out if you override the `Run` method.

The `ShutDown` method handles situations where the `Run` method fails to complete. The default implementation simply returns a HTML page to the server that says the CGI processing could not be completed. If you override the `ShutDown` method, try to make your code as short and simple as possible since it is likely that something is already wrong with the application. Before overriding the `ShutDown` method, consider whether it would suffice to use a stack-based class member in your `CCGIThread` subclass.

`Parameter(AEKeyword, ...)` makes an inline call to `AEGetParamPtr` and so we will focus on `Parameter(ECGIKeys)`. `Parameter(ECGIKeys)` uses a `static` buffer that is capable of holding the largest CGI variable coming from a WebSTAR server (as determined by the class `UtCGIKey`). Although it could use an automatic variable and tailor the buffer size for each call of `Parameter(ECGIKeys)`, the framework assumes that you will call the method frequently enough that making a single request for a large chunk of data will be more efficient. Since the buffer is `static`, all the threads share it and the framework only incurs a 32KB overhead.

`SetReply` is the other side of the coin and provides access to the reply `AppleEvent`. As in the case of `Parameter`, `SetReply` comes in two versions. The first version, `SetReply(AEKeyword,...)`, makes an inline call to `AEPutParamPtr` and we will present the code for `SetReply(const string&)`.

Listing 5: CCGIThreadApp.cp - Thread methods

Run

```
void* CCGIThread::Run(void)
{
  void*  theResult   (nil);
  try
  {
    CCGIThread::CheckMe();

    SuspendEvent();
    theResult = HandleWWWEvent();
    ResumeEvent();
  }
  catch (...)
  {
    const bool kIsFatal (true);
    CCGIApp::CCGIAppP()->LogError(
      "Uncaught exception in CCGIThread::Run",
      kIsFatal);
  }
  return theResult;
}
```

SuspendEvent

```
void CCGIThread::SuspendEvent(void)
{
  CCGIThread::CheckMe();

  ThrowIfOSErr_(::AESuspendTheCurrentEvent(&myEvent));
  CCGIApp::CCGIAppP()->EventStarted();
}
```

ResumeEvent

```
void CCGIThread::ResumeEvent(void)
{
  CCGIThread::CheckMe();

  CCGIApp::CCGIAppP()->EventFinished();
  ThrowIfOSErr_(::AEResumeTheCurrentEvent(
    &myEvent, &myReply,
    static_cast<AEEventHandlerUPP>(kAENoDispatch), 0));
}
```

ShutDown

```
string CCGIThread::ourShutDownMessage
      = UtHTMLWriter::HTMLHeader("Oops")
      + "<P>This service has unexpectedly needed to shut down.<BR><BR>"
      + "\r\nSorry\r\n" + UtHTMLWriter::HTMLFooter());

void CCGIThread::ShutDown(void)
{
  CCGIThread::CheckMe();
```

```
  SetReply(ourShutDownMessage);
  ResumeEvent();
}
```

Parameter

```
string CCGIThread::Parameter(ECGIKeys inKey) const
{
  CCGIThread::CheckMe();

  Size     theRealSize    (0);
  DescType   theRealType    (typeNull);
  Size     theMaxSize   (UtCGIKey::MaxSize(inKey));
       //the + 1 is to handle the terminating \0 if needed
  static char* ourBuffer
    = NEW char[UtCGIKey::MaxSize(CGIKey_MAX_DATA) + 1]);

  ThrowIfOSErr_(Parameter(inKey, typeChar,
    &theRealType, ourBuffer, theMaxSize, &theRealSize));

  if (theMaxSize < theRealSize)
  {
    theRealSize = theMaxSize;
    LStr255 theError
      = StringLiteral_("Parameter Overflow: ");
    theError.Append(&inKey, 4);
    CCGIApp::CCGIAppP()->LogError(theError);
  }
  ourBuffer[theRealSize] = '\0';

  return string(ourBuffer);
}
```

SetReply

```
void CCGIThread::SetReply(const string& inReply)
{
  CCGIThread::CheckMe();

  ThrowIfOSErr_(SetReply(keyDirectObject, typeChar,
    inReply.c_str(), inReply.length()));
}
```

CCGIApp

Most of the methods in `CCGIApp` do not directly relate to CGI event handling, if you wish to learn about them, consult the PowerPlant references and Monsterworks' API. The code implementing `CCGIApp` is in `CCGIThreadApp.cp` and the article presents some of it in Listing 6.

We begin with the `DeleteApp` method. This method is necessary because the destructor is `protected`. The `DeleteApp` method first `delete`s any `CCGIThread`s using `CCGIFactory`'s `TerminateThreads` method. `DeleteApp` finishes by deleting the `static` pointer

ourCCGIAppP. Because control of the CPU cannot return to a deleted thread, you should not call DeleteApp from within a CCGIThread.

CCGIApp installs WWWHandler as the CGI AppleEvent handler in its constructor. Its code below is short, the method uses the static pointer CCGIAppP to pass its arguments to the virtual method HandleWWWEvent. HandleWWWEvent uses the CCGIFactory to create a CCGIThread and then starts the thread running.

Listing 6: CCGIThreadApp.cp - Application methods

DeleteApp

```
void CCGIApp::DeleteApp(void)
{
  CCGIApp::CheckMe();

  //delete all CCGIThreads
  CCGIFactory::CCGIFactoryP()->TerminateThreads();

  //delete ourselves
  DisposeOf_(ourCCGIAppP);
}
```

CheckMe

```
void CCGIApp::CheckMe(void)
{
  ValidateThis_();

  ValidateObject_(myWindowP);
  ValidateObject_(myLastHitPaneP);
  ValidateObject_(myTotalHitsPaneP);
  ValidateObject_(myFinishedHitsPaneP);
  ValidateObject_(myLogFilePaneP);

  ValidateHandle_(myLogPrefsH.Get());
  AssertHandleLocked_(myLogPrefsH.Get());
  ValidatePtr_(myLogPrefsP);

  ValidateObject_(ourCCGIAppP);
}
```

CheckUs

```
void CCGIApp::CheckUs(void)
{
  ValidateObject_(ourCCGIAppP);
}
```

WWWHandler

```
pascal OSErr CCGIApp::WWWHandler(
  const AppleEvent* inEventP,
  AppleEvent* outReplyP,
  long inRefCon)
{
  OSErr theResult(noErr);
```

```
  try
  {
    CCGIApp::CheckUs();
    theResult = CCGIAppP()->HandleWWWEvent(
      inEventP, outReplyP, inRefCon);
  }
  catch (...)
  {
    theResult = errAEEventNotHandled;
  }
  return theResult;
}
```

HandleWWWEvent

```
OSErr CCGIApp::HandleWWWEvent(
  const AppleEvent* inEventP,
  AppleEvent* outReplyP,
  long inRefCon)
{
  CCGIApp::CheckMe();

  OSErr theResult(noErr);

  ThrowIfNil_(inEventP);
  ThrowIfNil_(outReplyP);

  CCGIThread* theCGIThreadP
    = CCGIFactory::CCGIFactoryP()->CreateThread(
      *inEventP, *outReplyP, inRefCon);

  ThrowIfNil_(theCGIThreadP);
  theCGIThreadP->CheckMe();
  theCGIThreadP->Resume();

  return theResult;
}
```

CCGIFactory

While the `CCGIFactory` class has the least amount of code attached to it, it is probably the most difficult of the three classes to design. The code implementing `CCGIFactory` is in `CCGIThreadApp.cp` and the article presents some of it in Listing 7.

The `TerminateThreads` method will terminate all the `CCGIThread`s that are currently running. Since you should be able to call it from anywhere (even within a `CCGIThread`), the method spawns a new thread to handle the terminations. Be aware that if you do call it from within a `CCGIThread`, control will not return to the calling point after the method completes.

The `CCGITerminator`'s `Run` method carries out the deletions. It first creates a `StCritical` object to prevent any other thread from taking control and then calls `UtThread`'s `DoForEach(UtThreadBoolIterator, void*)` method. A

UtThreadBoolIterator is a function which takes an LThread and a void* and returns a bool. CCGITerminator's TerminateThread method is such a method and is described later in this section. UtThread's DoForEach method is a slight variation of LThread's DoForEach method. Both will iterate through all the threads known to PowerPlant and apply their first argument to each of the threads. Unfortunately, LThread's method does not allow the passed in function to delete threads and UtThread overcomes this. The return value of the UtThreadBoolIterator argument determines whether the thread is still valid with a false value indicating the function deleted the thread.

Finally, we look at TerminateThread. This method first uses a dynamic_cast to determine if the thread is a CCGIThread. If it is, then TerminateThread calls the CCGIThread's ShutDown method and deletes the thread.

Listing 7: CCGIThreadApp.cp - Factory methods

TerminateThreads

```
void CCGIFactory::TerminateThreads(void)
{
  CCGIFactory::CheckMe();

  if (myTerminatorP == nil)
  {
    CreateTerminator();
  }

  ThrowIfNil_(myTerminatorP);
  myTerminatorP->Resume();
}
```

Run

```
void* CCGIFactory::CCGITerminator::Run(void)
{
  StCritical noYielding();
  UtThread::DoForEach(TerminateThread, nil);
  return nil;
}
```

TerminateThread

```
bool CCGIFactory::CCGITerminator::TerminateThread(LThread&
inThread, void*)
{
  bool theResult = true;
  try
  {
    CCGIThread& theCGIThread
      = dynamic_cast<CCGIThread&>(inThread);
    theCGIThread.ShutDown();
    theCGIThread.DeleteThread();
    theResult = false;
  }
  catch (bad_cast& theException)
  {
    //it wasn't a CCGIThread
```

```
  }
  catch (...)
  {
    //keep going
  }

  return theResult;
}
```

Thread management is a confusing subject and the following example may help describe the above strategy. Diagram 2 provides a general road-map of the example (time flows from the top to the bottom of the diagram). There are four threads: Thread 0x64 is the `UMainThread`, usually associated with the `CCGIApp` object. PowerPlant uses Thread 0x65 internally and we will ignore it here. Thread 0x66 is the `CCGITerminator` thread usually associated with the `CCGITerminator` object; and Threads 0x67 and 0x68 are both `CCGIThread`s handling `AppleEvent`s from the Web server. The diagram indicates the times the threads are valid by solid lines and the times the threads are executing by rectangles. The numbered black dots indicate the break-points described in Listing 8.

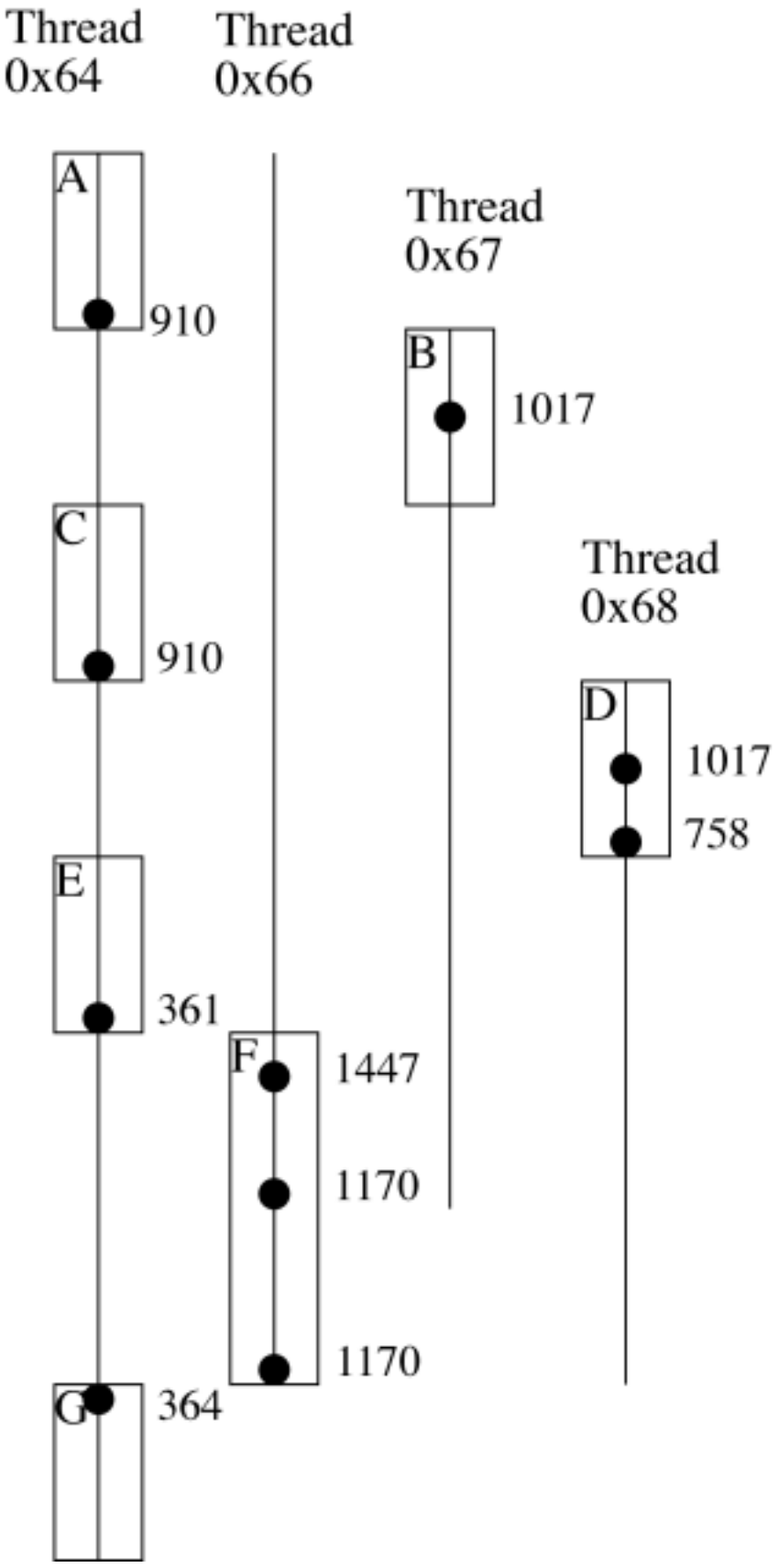

Figure 2: Some Threading

The example will probably make more sense if you follow it in the debugger with `CGI Base` (and one of the debug builds). You will want to set break points on the lines in `CCGIThreadApp.cp` indicated by Listing 8.

Listing 8 CCGIThreadApp.cp

```
                                                    Break-point locations
 361 CCGIFactory::CCGIFactoryP()->TerminateThreads();
 364 DisposeOf_(ourCCGIAppP);
 758 Yield(LThread::GetMainThread());
 910 theCGIThreadP->Resume();
1017 theResult = HandleWWWEvent();
1170 SetReply(ourShutDownMessage);
1447 UtThread::DoForEach(TerminateThread, nil);
```

Enable the debugger and run `CGI Base`. Then send the 10 minute wait to `CGI Base` from `Server Proxy` (either type 600 in the PATH pane or use the document `CGI Base - 10 min wait`). You should jump back to the debugger at line 910 in Thread 0x64 (the debugger displays thread numbers in the window title). This is the end of block A where we are about to start the `CCGIEchoThread` running with a call to `Resume`. Click the `Run` button and you should end up at line 1017 in Thread 0x67. This is the center of block B and the `CCGIEchoThread` now has control of the CPU and is executing its `Run` method. Hit the `Run` button and both of the debugger windows (0x64 and 0x67) will indicate that the application is running. What has happened is that control returned to the main thread (0x64) at the `Sleep` statement in `CCGIEchoThread`'s `HandleWWWEvent` method. The event loop has resumed and this is block C.

Now go back to `Server Proxy` and send in a fatal error (either type -10 in the PATH pane or use the document `CGI Base - fatal error`). You should jump back to the debugger at line 910 in Thread 0x64 again where the application is preparing to `Resume` the second `CCGIEchoThread`. You are now at the end of block C. Push the `Run` button and you will stop at line 1017 in Thread 0x68 (in block D) (just like the first thread stopped). Click the `Run` button and the break-point at line 758 in Thread 0x68 will stop the debugger. This is at the end of block D inside `CCGIApp`'s `LogError` method. The call to `Yield` will return you to the main thread where the `CCGIApp` will exit its `Run` method.

The next break-point is at line 361 in Thread 0x64 (end of block E). The `CCGIApp` has exited its `Run` method and `DeleteApp` is now executing. Push the `Run` method and you will go to line 1447 in Thread 0x66 where the `CCGITerminator` thread is now preparing to eliminate any remaining `CCGIEchoThread`s. Push the `Run` button. If you have set up your debugger to `Break on C++ Exceptions` there will be three failed `dynamic_cast`s here (Threads 0x64, 0x65 and 0x66 are not `CCGIEchoThread`s) and you will need to push the `Run` button after each of them. The next break-point is at line 1170 in Thread 0x66 (middle of block F). This is a `CCGIThread`'s `ShutDown` method and you can determine which `CCGIThread` by examining the `mThread` data member of the this object in the debugger. The debugger shows this data member in decimal (not hexadecimal like the window titles) so it will display 103 = 0x67 and you are in the `ShutDown` method of the first `CCGIEchoThread`. Push the `Run` button and you

will break at the same line (line 1170 in Thread 0x66, end of block F) but this time you will see that you are calling the `ShutDown` method of the second `CCGIEchoThread`.

This time when you push the `Run` button something interesting will happen. If you have not closed any of the thread windows, all threads will appear to be at line 364 (top of block G). What has happened is that Threads 0x66, 0x67 and 0x68 are no longer valid but the debugger hasn't noticed this. This behavior will happen anytime you have windows open for threads that have been deleted (either by calling `DeleteThread` directly or by completing their `Run` method). You can manipulate the application from any of these windows, except that the window title is incorrect. In this case, only Thread 0x64 is still running and so you can close all the other windows and push the `Run` button. The application will then finish off the main function and terminate.

If you opt to override the error handling in the Monsterworks' Framework, it is probably a good idea to sketch out the expected flow of control and then check it by placing break-points at key locations. It is very easy to place crucial error handling code at a location where it cannot be reached. For instance, if you call `DeleteApp` from a `CCGIThread`, then the call to `TerminateThreads` would execute correctly, but you would never return to call the `DisposeOf_(ourCCGIApp)`. You can find this type of error by using a diagram like Diagram 2 or by walking through the code.

CURTAIN CALL

That wraps up the introduction to the Monsterworks' Framework for CGI applications. At this point, you may want to use some of the other sources provided in the References section or just start experimenting on your own. A good way to get started is to try to port existing C or C++ CGI code to the framework. Once you become comfortable, you will find that writing CGI applications is very similar to writing ordinary applications except for the I/O handling.

While the framework probably still contains some undocumented features (pessimists use the term "bugs"), I currently use it for a variety of tasks and it has been very stable. If you find an undocumented feature or discover that the basic classes lack some functionality that you cannot provide through inheritance, send me an e-mail and I will do what I can to correct the situation.

REFERENCES

The following references provide coverage of some of the topics discussed above. Within each category, I have listed them in a suggested reading order and have provided some commentary on the contents of each.

CGI Programming, articles

If you have the MacTech CD-ROM, then you probably already have easy access to the articles listed.

O'Fallon, John, "Writing CGI Applications in C", *MacTech Magazine*, 11:9 (September 1995) provides an introduction to CGI programming. It covers many of the basic issues of CGI handling that I glossed over.

Neufeld, Grant, "Threading Apple Events", *MacTech Magazine*, 12:4 (April 1996) describes the code needed to thread AppleEvents. The Monsterworks' Framework and PowerPlant

provide much of the coding that the article describes, but the example he presents is a CGI application and so it provides another CGI example. The author of this article has written Grant's CGI Framework that provides a threaded framework in C. The most recent version I have found is at <http://www.nisto.com/cgi/framework>.

Urquhart, Ken, "High Performance ACGIs in C", *develop*, 29 (March 1997) presents an application shell for handling threaded CGI applications.

Halfmann, Klaus, "Writing ACGIs with MacApp", MacTech Magazine, 14:3 (March 1998) presents a shell for writing ACGI applications using MacApp (not surprising given the title). If you use MacApp instead of PowerPlant, this might be a good introduction.

Various articles, *MacTech Magazine*, 11:5–12:1 (May 1995–January 1996) provides articles about Web servers and CGI applications in a variety of programming and scripting languages. In particular, the article by Jon Wiederspan entitled "CGI Applications and WebSTAR" (July 1995) describes the interaction between Web servers and CGI applications.

CGI Programming, online

"Quid Pro Quo" is a free HTTP server that you can download at <http://www.socialeng.com>. It provides some examples of CGI programming in both C and AppleScript.

"WebSTAR Documentation" (version 3) from StarNine is available at <http://www.starnine.com> and contains information describing how to manage a WebSTAR server. The information about CGI applications is in Appendix C: "Extending WebSTAR: CGI's, Plug-Ins and Java". The company also provides an online tutorial at their site that provides more information about CGI development for WebSTAR servers.

CGI Programming, books

John December and Mark Ginsburg, "HTML 3.2 & CGI Unleashed" (1996) is where I learned much of my HTML and CGI. The problem is that the book is big (1300 pages). If you are looking for a very complete reference book, I would look into a newer version of this book. If you are looking for a gentle introduction to Web design, this book is probably a little too heavy (literally and figuratively).

PowerPlant

Metrowerks Corporation, PowerPlant Book (1998) covers basic PowerPlant programming.

Metrowerks Corporation, PowerPlant Advanced Topics (1999) covers the threading and networking classes provided by PowerPlant.

C++

Musser, David R. and Saini, Atul, "STL Tutorial and Reference Guide" (1996) is probably a little out of date (I don't know enough to be positive), but it does provide a good introduction to the ideas behind the container classes in the standard C++ library.

Stroustrup, Bjarne, "The C++ Programming Language" (1997) is not a good starting place for beginners, but it does provide a useful reference for the language, including the container classes in the standard C++ library.

Gamma, Helm, Johnson and Vlissides, "Design Patterns" (1995) provides some design patterns that are present in the PowerPlant and Monsterworks' Frameworks.