A Software Design Pattern for Embedded Web Applications

Charles E. Matthews
Fifth Generation Systems, Ltd.
June 20, 2020

This paper describes a new software design pattern for embedded web applications. It is important to define the domain of interest as server side software programming rather than user interface design because the web application literature often uses the term “design pattern” for both. However, the same term refers to distinctly different concepts in this problem space. After presenting the pattern, I will discuss some advantages and faults of the pattern.

ProbDB Software Application

The sample application that illustrates this design pattern is a small web-based application that uses the CGI (Common Gateway Interface) interface. I wrote the **ProbDB** application in 1999 as a mechanism for learning CGI programming and to provide a defect tracking system for some of my consulting clients. The **ProbDB** application is a light-weight defect tracking system that I used to track software program defects and feature requests. I successfully deployed this system for a number of my clients over the years to provide a defect tracking system if they did not have one of their own. It has been relatively stable with very few bugs of its own.

I have used this application on both Linux and Windows platforms. On Linux, I used the Apache web server. On Windows, I used both IIS and Apache. After the recent FastCGI redesign, I used the Nginx web server.

The user interacts with the application via a set of HTML pages that **ProbDB** creates. **ProbDB** uses an XML file for the defect database. There is some initial setup of the XML file that is performed by hand to tailor the data file for a specific customer – the list of products in the file and the list of developers who record, edit, and resolve defects.

In retrospect, I have two observations about this project that can only be made because of the amount of time that has passed since I first wrote the application. First, it has been surprisingly useful for a much larger number of my clients than I ever expected it to be. Back in my early days of consulting, I believed that most clients that I would acquire would already have a defect tracking system that they would be using – either a commercial system or an open source system. There are a large number of both types in existence. What surprised me was the number of client companies who didn’t have any company standard for defect tracking. In some cases they were using Excel files for defect tracking. However, in a surprisingly large number of cases, many companies simply were not tracking defects at all. Being able to provide a defect tracking system for them (for free) made me look much smarter to them than I deserved to be.

After I reached a point when I felt that the product was ready for general use and released it, I found relatively few errors in it during the following years. There were a couple minor bugs, but I observed no significant software defects during the past twenty years that required me to perform software maintenance. Recently, I made substantial modifications to the application for the purpose of this paper and added a complete set of unit tests to the application, which leads me to my second observation. For a product that worked relatively well in the field for the past twenty years, I found an astonishingly large number of subtle defects and program errors when I began looking for them. I was surprised by the number of errors that I found by refactoring the application and adding unit tests to a program that had appeared to be defect free for such a long time. Evidently, appearances were deceiving.

Design Patterns for Web Applications – Software vs. User Interface

In the web application literature, the term “design pattern” often refers to the process of constructing the graphical interface for the application. The user interface consists of components such as buttons, menus, labels, text boxes, and others. According to Fitzgerald, “*[These] components are the building blocks of a website, and patterns are established ways that users are used to performing tasks.*”[1] This context constrains the concept of a design pattern to the actions that a user performs when interacting with the web site. The pattern does not refer to any single component. Rather, it refers to the aggregation of a number of components to accomplish user oriented tasks.

The advantage for the user is that these patterns make the experience easier for the user. Consider patterns like the following for a fictional web app.

- User configuration pattern
- Device configuration pattern
- Network setup pattern
- Task scheduling pattern

Common tasks that are implemented in similar ways make the experience easier for the user in many ways. By using common design patterns, the application establishes trust with the user. Common design patterns are intuitive. Also, design patterns, by their nature, provide a common language that becomes shared between designers of the application.[2]

Other common web application design patterns are:[3]

- Navigation design pattern
- Form design pattern
- Table and List design pattern
- Search, Classification, and Filtering design pattern
- Graphic design pattern

In contrast to user interface design patterns, the definition of software design patterns is based on the original definition for pattern languages by Christopher Alexander for the field of architecture.

The elements of this language are entities called patterns. Each pattern describes a problem, which occurs over and over again in our environment, and then describes the core of the solution to that

---

[1] Abbey Fitzgerald, *How to Design for the Web Using Design Patterns*, https://getflywheel.com/layout/design-for-web-using-design-patterns-how-to, Posted: September 17, n2018, Read: May 29, 2020, p. 1.
[2] Ibid., p. 2.
[3] Summer Ye, *Top 6 Golden Principles for Web App Design Patterns in 2017*, https://www.mockplus.com/blog/post/web-app-design-patterns, Posted: June 18, 2017, Read: May 29,2020, p. 1-5.

problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.[4]

The literature contains examples of software design patterns in web applications that either illustrate instances of previously documented patterns[5] [6] or software architectural choices that are inappropriately labeled as design patterns.[7] I found no examples in the literature of previously undocumented design patterns for web applications. As Douglass shows with examples of design patterns for embedded software[8], design patterns do not need to be completely generic for all software development. They can still provide value even if they are specific to a software domain.

Design Pattern Template

I use the design pattern template as defined by Douglass[9]. Although this template differs slightly from the original template by Gamma, Helm, Johnson, and Vlissides[10], it seems more appropriate for a design pattern that is specific to a particular target domain rather than a general software pattern.

Name:

Embedded Web Pattern

Abstract:

Web servers use a small number of documented interfaces to connect the web server to an external application that runs on the device. Two of these interfaces are CGI (Common Gateway Interface) and FastCGI. CGI applications have been available for a very long time. FastCGI is a more recent evolution of the specification that addresses some of the performance problems with the CGI interface. Because the CGI and FastCGI interfaces can be implemented in low resource environments, they are useful components for embedded devices that require internet-enabled configuration or monitoring on a small, resource-constrained device platform.

The Embedded Web Pattern provides a stable design for small embedded web applications that implement either the CGI or the FastCGI interface. Although this design pattern may be applicable to other web interfaces, also, I have not yet implemented it with anything other than CGI and FastCGI.

Problem:

The basic architecture of a CGI application is quite simple. A user enters an URL in a client browser that activates an HTML page. The URL could contain a reference to a CGI application on a specific web server, or the HTML page that displays could contain an embedded reference to the CGI application. At some

[4] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, Shlomo Angel, *A Pattern Language – Towns – Buildings – Construction*, Oxford University Press, New York, NY, 1977, p X.
[5] Milecia McG, *4 Designs in Web Development*, https://dev.to/flippedcoding/4-design-patterns-in-web-development-55p7, Posted August 16, 2019, Read: May 29, 2020, p 1-2.
[6] *Design Patterns*, https://sourcemaking.com/design_patterns, Read: May 29, 2020, p 1-3.
[7] *Web Application – Design Patterns*, http://researchhubs.com/post/computing/web-application/design-pattern.html, Read: May 29, 2020, p 1-3.
[8] Bruce Powel Douglass, *Design Patterns for Embedded Systems in C – An Embedded Software Engineering Toolkit*, Elsevier, Inc., Burlington, MA, 2011.
[9] Ibid., p 7.
[10] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Company, Reading, MA, 1995, p 6-7.

point, an HTTP message request arrives at a web server that references the CGI application. The HTTP message contains HTML variables and values from the page that originated the HTTP message.

The web server spawns a process that executes the CGI application. The web server provides information about the HTTP message in a set of standardized environment variables and the STDIN input file stream. The CGI application processes the message request and responds by writing a formatted text stream to the STDOUT output file stream. In most cases, the output is a formatted HTML page that the web server sends to the client browser as a response to its request. The CGI process then terminates.

RFC 7231[11] of the Internet Engineering Task Force defines the types of allowed HTTP message requests. In most cases, the CGI application only needs to handle GET and POST messages. The pseudo-code for the CGI application is as follows.

```
initialize the application;
determine the requested activity;
handle the requested activity;
exit;
```

A significant difference between the CGI and FastCGI specifications is that the FastCGI application never terminates. It loops continuously waiting for the next HTTP request message from the web server. It processes each message and waits for the next.

Pattern Structure:

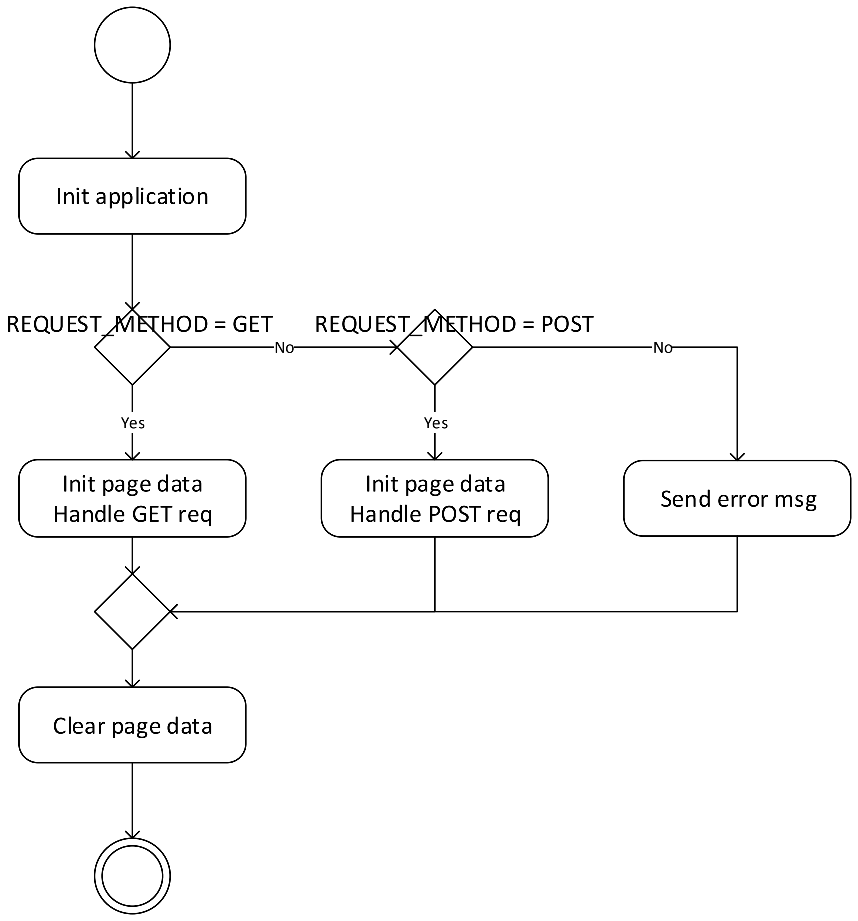
Robert Martin studied an existing application that contained ~30,000 lines of C++ code[12]. He documented a number of design patterns from this application and classified these patterns into similar categories of patterns.

- Container patterns
- High-level design patterns
- Low-level design patterns
- C++ related patterns

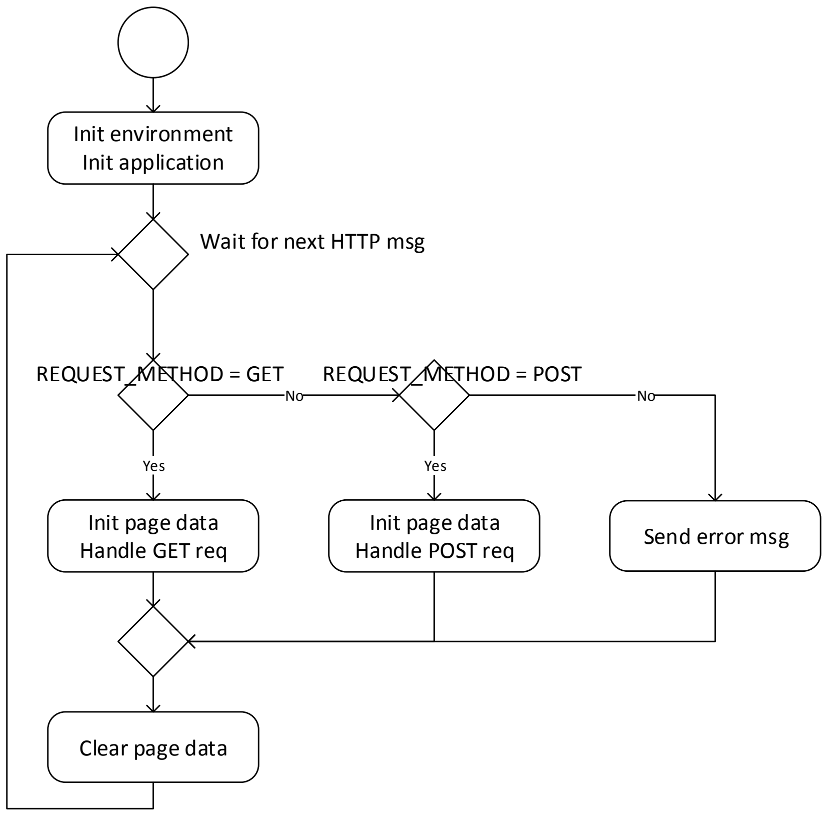
The Embedded Web Pattern can be regarded as a high-level design pattern because it provides top-level executive control for the application. The activity flow for the CGI version of this controller is as follows:

[11] R. Fielding, (ed.) and J. Reschke, (ed.), *RFC 7231: Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*, Internet Engineering Task Force (IETF), June 2014.

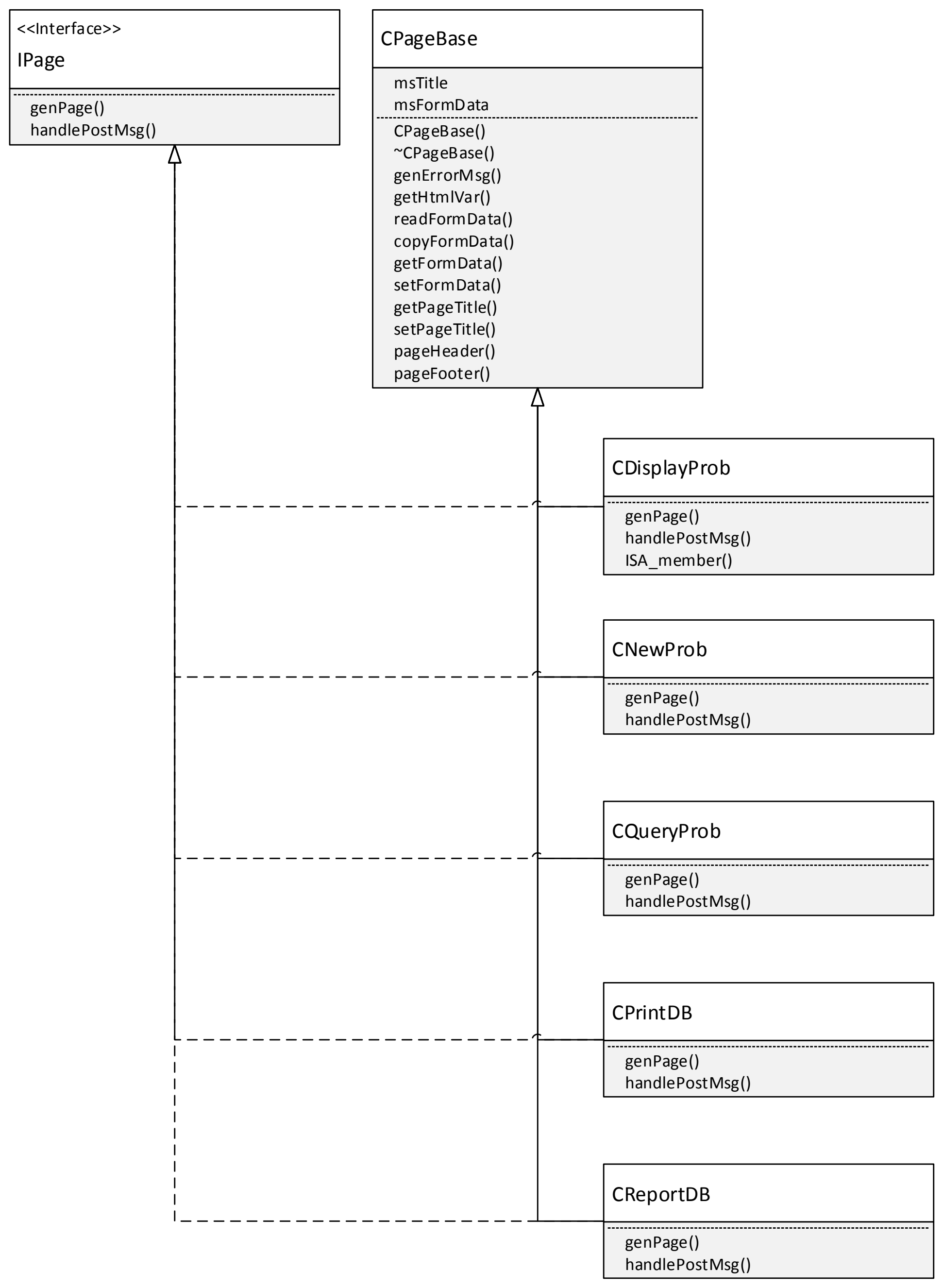
[12] James O. Coplien, (ed.) and Douglas C. Schmidt, (ed.), *Pattern Languages of Program Design*, Addison-Wesley Publishing Company, Reading, MA, 1995, p 366-367.



The activity flow for the FastCGI version of the controller is slightly different because of the FastCGI interface. The application does not exit after each HTTP message. The process is persistent and waits for the next HTTP message from the web server in a continuous loop.



Classes represent each HTML page that the application serves to the web server. These classes must contain at least two member functions as shown in the IPage interface class in the class diagram below – a function to generate the formatted HTML and a function to process the HTTP POST message from the client browser.



In general, each **handlePostMsg()** member function should process the HTML form data to determine what actions to perform and the next HTML page to display to the user. If the HTML page does not send any HTTP POST message, the **handlePostMsg()** member function displays an error message to the user because it should never be called.

Collaboration Roles:

For a complex CGI application, there may be other design patterns implemented in the member functions that process the HTTP messages or generate the HTML pages. There is no intrinsic collaboration between those patterns and this pattern.

Consequences:

The CGI and FastCGI interface specifications define, to a large extent, the overall architecture of this family of applications. However, the application developer has the freedom to choose how to implement the code that manages the interactions between the user, the web server, and the back end application. This pattern emphasizes the importance of the HTML page as a defining criteria on how to manage the application. It is possible that other choices could be made. Whether they result in a more efficient application depends upon the requirements of the individual application. This approach appears to work well for a large set of scenarios.

Implementation Strategies:

Issues to consider when implementing the Embedded Web Pattern:

1. Identify all HTML pages that can be accessed with HTTP GET requests. This pattern is based upon the architectural decision to organize the web application around the different HTML pages. You need to identify each of the pages that can be accessed via a GET request. Any page that is only activated via a POST message will be processed by the POST message request handler. However, any page that is accessed via a GET request could be activated directly by the URL that the user supplies in the web browser. Once you identify the HTML pages that can be accessed via GET requests, you must verify that HTML variables that can be supplied in the URL address are valid.
2. Identify all user actions that result in an HTTP POST message. Typically, only one user action occurs for any HTML page. It is the responsibility of the application to decode the user action and process the action.
3. Identify any HTTP messages other than GET and POST that must be handled. The web server identifies the type of HTTP message in the environment variable REQUEST_METHOD. Although GET and POST messages are the typical messages that an application handles, the RFC standard identifies additional message types: HEAD, PUT, DELETE, CONNECT, OPTIONS, and TRACE. The application architect must choose which message types need to be handled and add the appropriate request message handling code.

Related Patterns:

The Embedded Web Pattern has some similarity to the Strategy design pattern. The Strategy design pattern allows the designer to define a family of algorithms, encapsulate each one, and make them interchangeable. If one replaces “family of algorithms” with “family of HTML pages”, then there is some similarity to the Embedded Web Pattern.

However, this pattern is specific to web applications. By treating it as a unique pattern for the domain of web applications, it is easier to communicate the application design to another developer. Therefore, it has merit for consideration as a separate design pattern.

Example:

In embedded design systems that are memory constrained, it is a common coding practice to avoid all dynamic memory allocation at run-time. This application uses classes to represent each of the HTML pages that it manages. If you follow that coding practice, each of the classes would be statically declared at compile time and accessed as global variables. I chose not to follow that coding practice. I believe that the examples will be more easily understood if the reference to global variables is minimized.

The **main()** function for the application appears in the following box. For CGI, there is no environment to initialize. So, the code performs some application initialization and then detects whether the HTTP message is a GET request or a POST request. The function calls the appropriate message request handler and exits.

```
int main(int argc, char** argv)
{
    char  *psReqMethod;
    char  sMsg[100];

    //
    //  Initialize the app.
    //
    memset(gsURI, '\0', 30 * sizeof(char));
    memset(gsAppName, '\0', 20 * sizeof(char));
#ifdef PCNT
    sprintf(gsAppName, "cgi-bin/probDB.exe");
#else
    strcpy(gsAppName, "probDB");
#endif
    if (getenv("HTTP_HOST"))
    {
        sprintf(gsURI, "http://%s/", getenv("HTTP_HOST"));
    }
    else sprintf(gsURI, "http://");
    //
    //  REQUEST_METHOD should be GET or POST
    //  If GET, display the default page.
    //  If POST, read from STDIN for CONTENT_LENGTH number of bytes.
    //
    psReqMethod=getenv("REQUEST_METHOD");
    if (!_stricmp("GET", psReqMethod))
    {
        handleHttpGetReq();
    }
    else if (!_stricmp("POST", psReqMethod))
    {
        handleHttpPostReq();
    }
    else {
        //
        //  Error: Unknown request method.
        //
        CPageBase *pObj = new CPageBase();

        sprintf(sMsg, "ERROR: Unknown request method - '%s'.", psReqMethod);
        pObj->genErrorMsg(sMsg);
    }

    return (ERROR_SUCCESS);
}
```

By comparing **main()** to the activity diagram for the CGI interface, the activity flow is clear. After a small amount of initialization, the code determines the type of HTTP request by reading the

REQUEST_METHOD environment variable. If the request is a GET request, the code calls **handleHttpGetReq()**. If the request is a POST request, the code calls **handleHttpPostReq()**.

The HTTP message request handlers appear below. In this example, there is no default GET request page. Therefore, the code creates an HTML page that displays a user error in response to an HTTP GET request.

```
void handleHttpGetReq()
{
    CPageBase *pObj = new CPageBase();

    //
    //  GET Method: There is no default GET request page.
    //
    pObj->genErrorMsg("There is no default request page.");
    delete pObj;
}
```

This application generates and processes five HTML pages – a) *display a problem*, b) *create a new problem*, c) *query the problem database*, d) *print the problem database*, and e) *create a report of the problem database*. However, only four pages can send an HTTP POST message. The HTML pages contain variables that determine how the application must process the POST message. All of the HTML pages have three common variables: **activity**, **page**, and **Database**. The **page** value identifies which HTML page sent the POST message. The **activity** value identifies the action that the user requested on that page. The **Database** value identifies the XML file that contains the problem database. From these value, the application determines the action to take and the file that holds the problem database. (*Note: For a deeper understanding on how the CGI interface works, please review other documentation on the CGI standard.*[13])

```
void handleHttpPostReq()
{
    char        sVarPage[MAX_VAR_SIZE];
    char        sMsg[100];
    size_t      nBufferLength;
    int         nContentLength = atoi(getenv("CONTENT_LENGTH"));
    CPageBase *pPageBase = new CPageBase();

    //
    //  POST Method: Parse stdin for variables
    //  Read the buffer.
    //
    pPageBase->readFormData();
    nBufferLength = strlen(pPageBase->getFormData());

    if (nBufferLength != nContentLength)
    {
        //
        //  Error.
        //
        sprintf(sMsg,
```

[13] Rafe Colburn, *Teach Yourself CGI Programming in a Week*, Sams.net Publishing, Indianapolis, IN, 1998.

```
"handleHttpPostReq - Error reading the HTML variables.\n    CONTENT_LENGTH=%
d\n    Length read=%u",
            nContentLength, (unsigned int)nBufferLength);
        pPageBase->genErrorMsg(sMsg);
    }
    else {
        pPageBase->getHtmlVar("page", sVarPage);
        //
        //  Which page posted the message?
        //
        if (!_stricmp("query", sVarPage))
        {
            CQueryDB *pQueryDB = new CQueryDB();

            pQueryDB->copyFormData(pPageBase);
            pQueryDB->handlePostMsg();
            delete pQueryDB;
        }
        else if (!_stricmp("displayProb", sVarPage))
        {
            CDisplayProb *pDisplayProb = new CDisplayProb();

            pDisplayProb->copyFormData(pPageBase);
            pDisplayProb->handlePostMsg();
            delete pDisplayProb;
        }
        else if (!_stricmp("newProb", sVarPage))
        {
            CNewProb *pNewProb = new CNewProb();

            pNewProb->copyFormData(pPageBase);
            pNewProb->handlePostMsg();
            delete pNewProb;
        }
        else if (!_stricmp("printDB", sVarPage))
        {
            CPrintDB *pPrintDB = new CPrintDB();

            pPrintDB->copyFormData(pPageBase);
            pPrintDB->handlePostMsg();
            delete pPrintDB;
        }
        else {
            //
            //  Error. Unrecognized page.
            //
            sprintf(sMsg, "Unknown page request: [%s]", sVarPage);
                    pPageBase->genErrorMsg(sMsg);
        }
    }
    delete pPageBase;
}
```

The **handleHttpPostReq()** function reads the HTML form data and uses the **page** variable to determine which page sent the HTTP message. The function creates an instance of that page class and calls its **handlePostMsg()** member function to process the HTTP message.

The following code illustrates the implementation of the **handlePostMsg()** member function for **CPrintDB** – the print database class. The only valid activity for this page is "*Display Incident*", which displays the details for a specific problem record. The class method creates an instance of the **CDisplayProb** class, copies the HTML form data to it, and generates the HTML page that displays the details for a specific problem record. The **genPage()** function writes the requested HTML page to the web server.

The **handlePostMsg()** member function then performs some cleanup and exits. The calling functions also exit until ultimately the CGI application terminates until the web server sends a new HTTP message to the application.

```
void CPrintDB::handlePostMsg()
{
    char        sVarDBName[MAX_VAR_SIZE];
    char        sVarActivity[MAX_VAR_SIZE];
    char        sMsg[100];

    getHtmlVar("activity", sVarActivity);
    getHtmlVar("Database", sVarDBName);
    if (!_stricmp("Display Incident", sVarActivity))
    {
        CDisplayProb *pDisplayProb = new CDisplayProb();

        //
        //  Display an existing problem entry.
        //
        ParseFile(sVarDBName, NULL);
        pDisplayProb->copyFormData(this);
        pDisplayProb->genPage();
        //
        //  Garbage collection.
        //
        ClearDB();

        delete pDisplayProb;
    }
    else {
        //
        //  Error. Unrecognized activity.
        //
        sprintf(sMsg, "Invalid activity request for printDB page: [%s]",
                sVarActivity);
        genErrorMsg(sMsg);
    }
}
```

Advantages and Faults

Design patterns have a number of advantages. They increase developer productivity because they provide proven solutions to common problems. Once you identify patterns that you can use in your design, the remaining effort to architect your solution is less because using known design patterns decreases design time that you would need to expend if you designed everything from scratch. A solution that uses known design patterns is more easily understood and communicated among the developer team because everyone who is familiar with the pattern can understand the solution at a higher level of abstraction. Also, when the software product is either a family of related products or an evolving architecture that undergoes multiple phased releases, the use of design patterns can increase the consistency and cohesiveness of the software design.

However, there are also some disadvantages with design patterns. A design pattern that does not fit the software requirement can lead to an inefficient solution. Often this occurs when the designers are not well-trained on the design patterns that they are using. So, the developer team must have a sufficient level of training for the design patterns that they are using.[14]

With respect to the pattern presented in this paper, it is important to note that this design pattern may not be appropriate for all categories of web applications. Attempting to use it outside the architectural boundaries that I have identified could lead to an inefficient solution. If it can be adapted to other types of web enabled architectures that is a task for another day.

Summary

I did not identify this pattern when I initially wrote the **ProbDB** application. In fact, I didn't realize that the software could be refactored into the pattern until 20 years passed and I wrote a few more CGI applications. The experience is similar to what Gamma et al describes.

*It's easy to spot patterns when you look at enough systems. But **finding** patterns is much easier than **describing** them. If you build systems and then reflect on what you build, you will see patterns in what you do. But it's hard to describe patterns so that people who don't know them will understand them and realize why they are important.*[15]

This pattern is important because it is useful for embedded web applications.

[14] https://sourcemaking.com/design_patterns, p 4.
[15] Gamma, *Design Patterns*, p 355.