Plugins architecture for e-learning systems

BOGDAN BRUMAR
Department of Computer Science
Faculty of Sciences
Lucian Blaga University of Sibiu
Str. Ion Ratiu, no. 5-7, 550012
Sibiu, ROMANIA

EMIL MARIN POPA
Department of Computer Science
Faculty of Sciences
Lucian Blaga University of Sibiu
Str. Ion Ratiu, no. 5-7, 550012
Sibiu, ROMANIA

IULIAN PAH
Department of Sociology,
Babes-Bolyai University
Cluj-Napoca, Bd.21 decembrie 1989, nr.128-130, 400604,
Cluj-Napoca, ROMANIA

Abstract: This paper describes the use of plugins for e-learning systems. As a plugin programmer can use this architecture to improve your plugins and make them integrate better into e-learning systems. Recently, the industry has come to capture such processes at all levels, from analysis and design to coding. When these process definitions are presented with context, rationale, and examples, they are known as patterns [1]. Patterns are among the hottest topics in software engineering today. The concept is simple-find techniques that work and document them in a standard format.

Key-Words: patterns, plugins architecture, plugin instance, plugin method.

1 Introduction - Patterns

The history of software engineering is a progression from chaos to structured methods, object-oriented methods, and an emphasis on defined, repeatable processes. Patterns are the logical next step in that progression. They represent a "customary practice" that has been documented to provide context, rationale, and additional information.

The term was coined by architect Christopher Alexander. In his 1977 book, A Pattern Language (Oxford University Press), Alexander describes patterns as recurring themes in architecture. The "Window on Two Sides of Every Room" pattern describes a general rule with specific benefits but does not prescribe the size of the windows, their height from the floor, or the distance between them.corporation’s environment.

Patterns are characterized by three elements in pattern documentation:
- An issue in system design. This often is a problem to solve or a situation that you may encounter. This issue serves as the trigger that tells the designer a particular pattern is appropriate.
- A description of a solution. Although concrete examples should be given, the heart of the description is abstract (because the system design issue is abstract).
- The consequences of applying the abstract structure to a system architecture. These consequences can be described in terms of trade-offs. They help the system designer determine whether or not the pattern is appropriate in light of the system's overall constraints.

Various models exist for design patterns. This section shows Alexander's template. The next section shows the template adopted by the Gang of Four.

Alexander's patterns typically are written in a narrative style. They include a concrete example of a problem the pattern solves, in context, and then a solution. A typical Alexandrian pattern mapped into the software domain has seven sections:

- Pattern Name
- Problem
- Context
- Forces
- Solutions
- Force Resolution
- Design Rationale

The "forces" referred to by Alexander correspond to the "-ilities" of software engineering (such as portability, reliability, maintainability, and readability). Force resolution describes how these trade-offs are made in a particular pattern and leaves the door open for other patterns in the same problem with different weights to the various forces.

The "Gang of Four" Template [1] The Gang of Four provides a more formal structure for documenting patterns. This template serves as a checklist for the pattern author and provides a convenient reference for users of the pattern. The thirteen elements of the Gang of Four's pattern template, along with their section numbers, are as follows:

1. Name-A succinct identifier that becomes part of the design vocabulary. Typical names include "Builder," "Proxy," and "Walker."
2. Intent-A statement of the issue the pattern addresses and a description of how the pattern
improves the design.

3. **Also Known As**—Synonyms for the pattern.

4. **Motivation**—A concrete scenario of the issue described in the Intent section. The scenario shows the classes and objects used in the pattern that addresses the issue.

5. **Applicability**—A short statement of the problem or conditions that trigger the use of this pattern.

6. **Structure**—Typically, a graphical representation of the pattern, using a notation based on the Object Modeling Technique (OMT), with pseudocode for the methods.

7. **Participants**—A description of the classes and/or objects of the pattern, showing their responsibilities and collaborators. The standard information on CRC cards (Classes/Responsibilities/Collaborators) is an appropriate model for this section.

8. **Collaborations**—A description of the collaboration relationships among the collaborators named in the Participants section.

9. **Consequences**—A discussion of the trade-offs of using this pattern. What system design objectives are met and at what cost in terms of other objectives? The software engineering "-ilities" are a good starting point for listing design objectives.

10. **Implementation**—Concrete tips and hints that may be useful to the users of the pattern.

11. **Sample Code**—Whenever possible, at least two examples from different domains.

12. **Known Uses**—Well-known problems that are solved using this pattern.

13. **Related Patterns**—Other patterns that might be used along with this one, including patterns with an alternative approach to the same issues.

The first three sections are used to unambiguously identify the pattern. Section 4 captures most of the content of an Alexandrian pattern and sets a context for the pattern. Section 4 also includes a concrete example to set the scene for the sections that follow. Sections 5 through 9 define the pattern in abstract terms. Sections 10 and 11 return to the concrete. Section 12 could be a bibliography, and Section 13 could be a cross-reference within a pattern catalog or even a cross-reference to other well-known catalogs and repositories.

One way of thinking about patterns is in the context of Figure 1. This figure, from the paper, *Design Patterns: Abstraction and Reuse of Object-Oriented Design*, by the Gang of Four [1], shows a design space that encompasses all of the patterns in their pattern catalog.

---

**Fig. 1 - Most patterns for object-oriented development fit somewhere on this map.**

In the context of this figure, each pattern lives at the intersection of a "jurisdiction" and a "characterization." Jurisdiction has to do with whether the pattern works on classes, objects, or more complex structures such as peer objects.

Creational characterization is used for patterns that make new elements of their jurisdiction. Structural characterization means that the pattern adds functionality. Behavioral characterization patterns are used to show how elements cooperate to fulfill their semantics.

The Observer pattern, for example, abstracts the synchronization of state or behavior. The Strategy pattern provides a way to capture an algorithm in an object. Many STL-style algorithms are examples of Strategies. STL-style iterators are examples of patterns with object jurisdiction and behavioral characterization.

As an example of a typical pattern, consider the problem of the pointer in C and C++. C and C++ programmers are plagued because dereferencing a null pointer usually causes a program to exit unexpectedly. One pattern that addresses this issue is the protected proxy (a member of the Proxy family of patterns).

A protected proxy can be dereferenced at any time—if it doesn't point to anything, it raises an exception. Proxies are associated with specific objects. Structural characterization has to do with adding new functionality. Consequently, the Proxy pattern lives at the intersection of Object jurisdiction and Structural characterization [2].

For completeness, the jurisdiction and characterization should appear at the top of a pattern description, near the name.
## 2 Plugins architecture

This section introduces three ways of starting a plug-in: embedded, hidden, and full-page. It then moves into a high-level look at plug-in design. This section points out patterns that are associated with the plug-in architectures [1].

### Three Ways of Invoking a Plug-In

Plug-ins fit into HTML in three different ways: embedded, hidden, or full-page. Although these terms are sometimes used as though they are types of plug-ins, they actually are just ways of calling the plug-in. There are coding differences, however; if the plug-in is designed to be embedded, check when it is started to see how it was invoked and warn the user if it was called improperly. The HTML writer will see this warning when testing the page, so the warning should give the HTML writer enough information to fix the problem.

**Embedded Plug-Ins**

 Netscape has added the `<EMBED>` tag to the set of extensions recognized by Navigator. The `<EMBED>` tag includes the SRC attribute, so an HTML writer can write the following line, and the browser attempts to get the requested entity:

```
<EMBED SRC="http://www.server.com/aMovie.avi">
```

The server uses the file extension (.avi) to select a MIME content type. When the server sends the Content-type header line back to the browser, Navigator selects the plug-in that handles this media type.

The HTML writer can also use the TYPE attribute of the `<EMBED>` tag to force Navigator to look for a plug-in to handle the specified type. The TYPE attribute makes sense when a plug-in doesn't need a data stream, or when the data is generated dynamically and doesn't have a well-defined MIME media type.

Navigator defines a rectangle in the Navigator window to hold embedded data. The HTML writer can use the HEIGHT and WIDTH attributes of the `<EMBED>` tag to specify the size of the rectangle.

**Hidden Plug-Ins**

A plug-in also may be hidden. Use the `<EMBED>` tag, but specify a HIDDEN attribute. (You can say HIDDEN=true if you prefer-true is the default.)

**Full-Page Plug-Ins**

Sometimes you want to allocate a whole window to the plug-in content. Rather than using the `<EMBED>` tag, just link to the file. For example, an HTML writer can write the following:

Examine

```
<A HREF="http://www.server.com/myFile.xyz">
my file</A>
```

When the site user follows this link, the server looks up the MIME media type based on the xyz file extension and begins streaming the data to the browser. Navigator looks up the media type from among its registered plug-ins-if it finds a match, it loads the plug-in and starts an instance.

Navigator plug-ins are implemented as code resources that execute in the same address space as Navigator. They are distinguished from other applications by the fact that they call and are called by Navigator. Figure 2 shows the context of a Navigator plug-in.

![Fig. 2 - A Navigator plug-in lives in the space between Navigator and the native operating system.](image)

Most plug-ins have examples of patterns with object jurisdiction and behavioral characterization. Recall that when Navigator starts, it registers the MIME media types used by the plug-ins and waits until it's called on to process one of those types [2]. When Navigator reads a content type that it doesn't handle internally, it looks at the list of registered plug-ins to see if the type can be handled by one of the plug-ins. If so, it loads the plug-in code, instantiates two classes (one in the plug-in and one in Navigator), and makes a series of calls to the plug-in object. Figure 3 illustrates these two objects. All calls from Navigator to the plug-in start with the characters "NPP_." Calls from the plug-in to Navigator start with "NPN_"

![Fig. 3 - Within its context, a Navigator plug-in is defined by two objects.](image)
All of the methods that are part of the plug-in Application Program Interface (API) are defined in the npapi.h file.

You can use C++ exceptions to signal conditions within your plug-in. Error conditions to and from Navigator are passed through result codes from the methods. npapi.h has definitions of the result codes in NPError type [3].

Figure 4 illustrates the calls Navigator makes to the plug-in. When the plug-in is loaded (and before it is instantiated), Navigator calls NPP_Initialize(). As soon as a copy of the plug-in is instantiated, Navigator calls NPP_New(). If more copies of the plug-in are opened while the plug-in is loaded, NPP_New() is called for each new instance.

Fig. 4 - During its life, Navigator makes a series of calls to a plug-in.

When NPP_New() is called, Navigator sets its pluginType parameter to either NP_EMBED or NP_FULL to reflect the way in which the plug-in was invoked.

Note: The plug-in documentation described three types of plug-ins: embedded plug-ins, full-page plug-ins, and hidden plug-ins. In the current implementation, Netscape supports embedded plug-ins and full-page plug-ins as pluginType types in NPP_New(). If the HTML writer uses an <EMBED> tag with the HIDDEN attribute to make a hidden plug-in, Netscape sets pluginType to NP_EMBED but does not call NPP_SetWindow() to tell the plug-in to draw into the window.

When the instance window is closed, Netscape calls NPP_Destroy() on that instance, and then deletes the instance. When the last instance of a plug-in is destroyed, Netscape calls NPP_Shutdown(), and then unloads the plug-in.

You, as the plug-in programmer, are free to take any appropriate action when Netscape calls your plug-in. Generally, NPP_New() is used to do all initialization not done by the constructor. NPP_Destroy() can delete any objects instantiated during initialization.

You also may want to use NPP_Destroy() to save information related to an instance. When possible, Navigator restores that information to a new plug-in instance that is instantiated on the same content.

Suppose that a user opens the myMovie.avi file. Navigator starts an AVI-reading plug-in that begins playing the movie. Before the movie is finished, the user closes the plug-in window. If the user later opens myMovie.avi again, the plug-in could use stored information to return the user to the exact frame where he or she left off.

The Plug-In Object Most plug-ins interact with windows in the native environment. A plug-in that draws into a rectangular space in the Navigator window is known as an embedded plug-in. A plug-in that opens its own window is referred to as a full-page plug-in.

At one time, Netscape envisioned a third type of plug-in, which would run in the background and have no open window. Their latest thinking is that functionality should be provided with the existing types of plug-ins, but that the HTML writer should pass the HIDDEN attribute to the plug-in through the <EMBED> tag.

After it is instantiated, Netscape calls the plug-in's method NPP_SetWindow(). As a parameter for this method, Netscape passes the handle for the native window provided for the plug-in's use.

Unless you make other arrangements, the plug-in's methods are part of Navigator's main thread of control, which means that if a plug-in seizes control for a long time, Navigator looks to the user as though it has stopped responding [2].

After the plug-in instance is instantiated and has a handle to its window, Navigator can pass data to it. Navigator signals the plug-in that a stream of data is available by calling the plug-in's NPP_NewStream() method.

When the stream is no longer available, Navigator calls NPP_DestroyStream(). Netscape recommends that plug-ins wait while Navigator buffers data, and then calls NPP_WriteReady() and NPP_Write() to "write" the data into the plug-in.

When Navigator calls NPP_NewStream(), the plug-in sets up a buffer to receive incoming data. Before actually writing the data to the plug-in, Navigator calls NPP_WriteReady(). That method reports back to Navigator the space remaining in the buffer and clears the way for Navigator to write that number of bytes.
After Navigator puts data into the buffer, the plug-in is free to process this data asynchronously. Before Navigator writes to the plug-in again, it calls NPP_WriteReady() to see how much buffer space is available. NPP_WriteReady() is an example of an Observer pattern, providing synchronization and coordination between two objects.

The plug-in printing mechanism is an example of a Mediator pattern. The printing mechanism is decoupled from Navigator by the plug-in object. If the plug-in is using a full window, Navigator even offers the plug-in control of the print dialogs.

NPP_Print() is used when Navigator wants the plug-in to print itself. If the instance is full-page, NPP_Print() is first called with platformPrint->mode set to NP_FULL before Netscape displays any print dialogs. This technique gives the plug-in an opportunity to put up its own dialogs and to control the printing process. If it does this, it should set pluginPrinted to true before returning.

If you want to allow Navigator to set up the printing process, set pluginPrinted to false and return. Navigator calls NPP_Print() again, this time with platformPrint->set to NP_EMBED. The plug-in that sees platformPrint->mode set to NP_EMBED knows that it is responsible only for drawing its representation into the native window. Navigator sends this window in the member platformPrint->embedPrint.window.

Windows programmers should note that the coordinates of the window rectangle are in TWIPS—a Microsoft-coined term that means "twentieth of a point." Make sure that you use the DPtoLP() method of the CClientDC object to convert the coordinates from device coordinates to logical coordinates before drawing text into the window.

NPP_URLNotify() is used in connection with methods on the Netscape peer to tell the plug-in that an URL-related request to Netscape has completed. This method is described in greater detail in the next section, "The Netscape Peer."

The Netscape Peer Recall that when Navigator instantiates a plug-in object, it also instantiates a peer object of its own. You can make calls to the peer with methods that begin with "NPN_." These methods represent services that Navigator performs for the plug-in [3]. Some of these services are rather mundane. NPN_UserAgent(), for example, returns the browser's ID string. NPN_Version() returns the version numbers of the browser and the plug-in SDK. There is platform-specific code (in npmac.h and npwin.h) to compare the browser's major version number with the plug-in's version—if the user tries to load a version 3.x plug-in with a version 2.x browser, the plug-in exits.

3 Plugins direction for e-learning

Multimedia

One major use of plug-ins is to extend Navigator to handle new media types as though they were native. By using the <EMBED> tag as though it were, say, an <IMG> tag, the HTML programmer can place advanced graphics and movies on the page. The plug-in is responsible for reading the incoming data stream and drawing a representation of the data directly to the window.

If the <EMBED> tag is called with the HIDDEN attribute set to true, no window appears. The plug-in designed to play a sound may be used in this way.

Document Viewers

Sometimes the non-native media type is best thought of as a document rather than as a component to an HTML page. The Adobe Acrobat PDF viewer is an example of this type. When a user follows a link to a PDF document, Navigator starts the Acrobat plug-in in a full window and displays the document.

It is still possible to observe the Netscape look and feel in a full-page plug-in. When the user moves the cursor over a link, call NPN_Status() to show the link. When the user selects a link, call either NPN_GetURL() to open a new window or NPN_NewStream() with current in the target. Then call NPN_Write() to overwrite the old contents of the current window.

A Plugin Based Application

Although most plug-ins are multimedia-type viewers or document viewers, plug-ins give the programmer the potential of building a full client-server application.

The Story Pattern

The Story pattern is triggered when someone decides to computerize something. The key element of Story is to encourage the customer to talk about ways the end user will use the application. For this example, your customer is a manufacturer of lawn tools. The vice president of sales tells you stories about the sales staff. The company distributes its products through distributors such as hardware stores, garden supply stores, and department stores. Here is one of their stories [1].

The sales staff is expected to review their accounts looking for customers who haven't bought for a while or whose purchase volume has declined. The staff members are also encouraged to keep in touch with their accounts—they try to call every customer at least three times a year, even if their volume is high.

Sales are highly seasonal—the summer months are the best, but sales of rakes, leaf blowers, and work gloves are good in the fall. Snow blowers make up the bulk of the winter sales.
Every day each account rep makes a list of which customers to call. In larger firms customers usually speak to someone in purchasing. At smaller firms they try to speak to the owner. Most of the accounts are smaller firms.

When account reps have the customer on the phone, they talk about the customer's recent purchases. They are often called on to answer simple technical support questions about the products or to describe features. They also like to have on hand a complete record of the customer's buying history.

The account reps have several problems with the current manual system. The product catalog is often out of date—by the time the summer catalog is out, the fall selling season has begun.

Technical support notes are packaged separately from the catalog—often, account reps can find the product in the catalog (they can describe the features) but cannot find the tech support notes (so they cannot answer questions about the use of the product).

Calls from account reps often come as an interruption to their customers, so time is of the essence. Sometimes they are asked to call back. More often the owner can give them a few minutes and may even place an order if the account rep has the necessary information on hand.

Another problem the account reps have is that orders are sometimes put on hold by the credit department. This practice has led to some hard feelings and lost accounts: the account rep takes the order and is often unaware that the order has been held up for days waiting for the credit manager to approve it.

The User Decision Pattern After you have a set of stories, you can begin to map them to the elements of a user interface. Figure 5 shows a graphical representation of this mapping. The best user interfaces show consistency.

Fig. 5 - Mapping stories to user interface elements requires a transformation from chronological order to spatial organization.

4 Conclusion
Plug-ins are based on a pair of collaborating objects—