IJCSE International Journal of Computer Sciences and Engineering Open Access
Review Paper Volume-5, Issue-3 E-ISSN: 2347-2693

Framework for Object Oriented WWW Applications using Embedded Concepts

A. Chagi

School of Computing & IT, REVA University, Bangalore, India

*Corresponding Author: anitachagi@reva.edu.in

Available online at: www.ijcseonline.org

Received:27/Feb/2017 Revised: 06/Mar/2017 Accepted: 25/Mar/2017 Published: 31/Mar/2017

***Abstract*—We present the design of a *C++* framework for building custom Web agent applications. Our framework includes abstractions for networking and communications as well as a format-independent set of classes for representing document components. We discuss the design and parts of the implementation of the framework and present possible extensions till date.**

***Keywords*—www; Agent; Abstraction; Object; Class; Framework.**

I. Introduction

Over the last several years, the World-Wide-Web (WWW) has grown enormously and has become an invaluable source of information of all kinds. The result is a great deal of popularity enjoyed by Mosaic, Netscape, internet explorer, chrome and other browsers. The availability of the Web has also made HTML (different versions) a common means for exchanging data. Many corporations are using Web connections as a standard means from the start for retrieving information from remote databases located within the enterprise as well as outside.With the availability of information in HTML form, it is natural to create custom applications which do not merely browse the Web but extract information to perform specific tasks. For example, one can create a collection of "agent" applications that conduct routine periodic activities. An agent that synchronizes clocks might retrieve time information from a global time server; a sports enthusiast's personal information manager might retrieve information about sports events on television and arrange his schedule to watch them.Building such agents is eased significantly if a collection of C++ classes is available that encapsulate the networking and document-specific aspects of interaction with the Web.
Such classes offer several advantages:
High-level abstractions: They can hide the low-level details of accessing the Web and Make it possible for the application programmer to focus on application-specific tasks.
Platform independence: The class interfaces can mask: out all platform-specific aspects so that the application that uses the classes is naturally portable across platforms.
Document format independence: We can define the class interfaces to represent abstract notions relating to documents rather than specific to a particular representation language such as HTML. This isolates the application from language details, thus preserving its extensibility. For example, if a subsequent version of the application needed support for a different document format such as Adobe PDF or BTEX [1], it is only necessary to change the implementations of the document classes; the rest of the application code is unaltered.
Embeddability: Unlike other interpreted languages such as Java and Smalltalk, C++ is usable in conjunction with other languages such as C or FORTRAN to the extent that one application can be built using more than one of these languages.
In this paper, we describe the design and implementation of such a set of classes. In addition to the above properties, our classes preserve extensibility and reusability by making full use of the inheritance and polymorphism available in C++. They also achieve robustness via utilization of C++ exception-handling facilities. The design of the classes uses some of the ideas of Booch [2] and design patterns expounded in [5]. Our classes are designed to leverage the portability and GUI capabilities of the existing class library YACL [9].We begin with the considerations that drive the design and discuss the resulting classes and their methods. We then show a few examples to illustrate how these classes may be used. We also address the question of how to support computations that are not directly performable via HTML. We also address the question of how to support computations that are not directly performable via HTML.

II. RELATED WORK

In this section, we briefly discuss two projects whose contents are related to the work described in this paper.

The Java project includes a framework for Web access, with the HotJava application for browsing. Java is an interpreted language, so that the entire Java interpreter is part of any Java application. The result is that any Java objects-even ones that neither the server nor the browser is aware of may be transmitted across the network via any Web server that supports the feature. This makes Java a powerful and highly programmable environment. On the other hand, this also entails other problems, such as reduced execution speed and greater difficulty in linking with non-Java libraries. There are also security concerns arising because entire Java applications can be transmitted in response to queries. We choose to use C++ in order to overcome these problems. However, it is possible to construct custom objects (discussed later in this paper) to facilitate some of the programmability offered by interpreted languages such as Java.

The ASX framework includes classes for distributed computation, with class abstractions for communication mechanisms, event demultiplexing, service configuration, and the like. ASX's emphasis is, however, is on providing a general-purpose framework for distributed processing. Our work in this paper is directed specifically at Web client and agent applications.
The CORBA standard provides a basis for object-oriented distributed computation. We propose to include support in subsequent extensions of this work [3].

III. DESIGN OF THE CLASSES

A Web agent needs at least four different aspects of Web-related functionality:

- Determining the resource id's (URL's) of relevant documents;
- Retrieving documents from a remote server;
- Rendering the documents, in part or whole, on a local display; and
- Extracting information from retrieved documents for subsequent use.
  Of these, we address the last three aspects. We omit the issue of determining appropriate URL's, since there are now several search engines available for this purpose.

Network-related classes
First, consider the aspect of retrieving a document. Ideally, a programmer would want to simply specify a resource (via its URL) and ask the library t o retrieve the resource via code that looks like this:

```
Document* aDoc ;
ResourceSpec aSpec ("http://nnn.cs.sc.edu");
aDoc = RetrieveDocument (aSpec);
```

This suggests the need for **ResourceSpec** and **Document** classes [4]. The **RetrieveDocument** function retrieves and returns the document with the specified **ResourceSpec** value. (In practice, rather than returning a **Document** instance, the function returns a pointer t o a **Document** object, enabling it to potentially return an instance of a derived class and thus exploit polymorphism). This code fragment reinforces the viewpoint that every **ResourceSpec** value has a unique **Document** associated with it. The code fragment above suggests the view that the application programmer is only concerned with the document itself, not the kind of connection used to retrieve the document. This is not always the case; occasionally, the programmer might wish to specify the kind of connection t o be used, e.g., *ftp* or *gopher*. This suggests the need for a *connector*, which is an object that creates connections. The connector is a *factory* that creates other objects. To illustrate its use, here is the **Retrieve Document** function.

```
extern Connector Theconnector;
Document* RetrieveDocument
(const ResourceSpec& spec)
{
// Get a connection from the factory:
Connection* httpConn = Theconnector
MakeHTTPConnection (spec.Address());
// Build a stream for the document: HTMLStream&
aStream = httpConn-> Documentstream
(spec.Document());
// Reconstruct the Document from the stream:
Document* aDoc
aStream>> aDoc; httpConn->Close();
return aDoc;
}
```

This function assumes the existence of a global instance **Theconnector of Connector**, which creates and returns connections. It also uses the **ResourceSpec**'s **Type, Address** and **Offset** member functions; thus a **ResourceSpec** object is thought of as encapsulating the *connection type* needed to get the resource (HTTP, FTP, gopher etc.), the *address* of the remote machine at which the resource is located, and the *offset* (or local URL) on that machine. The **HTMLStream** class is discussed later.

The library provides an abstract Connection class. From this class are derived the concrete classes HTTPConnection, FTPConnection and Gopherconnection (among others), which override some of Connection's methods. The abstract class provides a standard interface through which application code can interact with every kind of connection. This scheme decouples the application code from the implementations of different kinds of connections, so that new kinds of connections can be added, and existing connection types extended, without breaking application code.

Documents and related classes

The document classes represent two broad categories of objects: documents and their contents, and the media on which the documents are stored [5].

Storage media classes

We think of documents as stored in a *data stream.* A data stream stores a *passive representation* of a document (in contrast t o the active representation as a C++ object). Data streams can be classified according to the format in which they store data; thus, for example, we distinguish HTML data streams from LATEX data streams. The class abstractions naturally mirror these concepts: the **HTMLStream** class represents a data stream in **HTML** form, while the LaTeXStream class represents a data stream in LATEX form.

The stream classes view their content as character strings, and include the capabilities for searching the
string by content. This facility allows the application to cull any data it needs from the passive representation.

Document classes

The classes in the document category represent documents and their components. The classes are designed so that the application programmer can think of them as abstractions rather than in terms of their representations in HTML or some other language. For example, an **OrderedList** is a sequence of strings, while a **Table** is an object that represents a two-dimensional array of cells, each containing other document items. The member functions defined on these classes correspond in a natural way to the operations performable on the abstractions they represent. For example, an object of the **Table** class can be asked to return the item in a particular cell of the table, while an **OrderedList** can be asked to return one of its contained items.

Abstractly, a document, represented by the class Document, is a tree of *document items.* The tree corresponds naturally to the parse tree of an HTML document. A *document item,* represented by the class **DocumentItem**, is a part of a document. **DocumentItem** is an abstract class which, like **Connection**, provides a standard interface through which application code can access elements of a document. **A** document is a sequence of one or more document items.

The DocumentItem class provides protocol (via pure-virtual member functions) for conversion back and forth between the item and a passive representation on a data stream. This design requires the DocumentItem classes to interact with data stream classes, not directly with connection classes. This is natural, since documents can exist independently of network connections[6]. This design also makes possible the creation of applications that, manipulate documents without interaction with a network. Figure 1 shows part of the DocumentItem interface.

To ensure type safety (so that type errors are caught at compile time), the signatures of these functions include the kind of data stream involved.

Document items can be simple t.ext strings or composite entities such as ordered or unordered lists. They can also be text strings qualified with font changes (such as the HTML <FONT **SIZE=+2>** item) or emphasis change (such as the HTML <I>and **<B>** items or the LATEX {\tt } group). Thus each kind of document item has a corresponding class derived from **DocumentItem**.

The containment relationship that, exists among document items (e.g., an ordered list contains a sequence of items representing the elements of the list) carries over naturally t o this model. Thus the **OrderedList** class contains a sequence of **DocumentItem** objects; each of these may itself be of a class derived from **Document Item,** thus providing for nested structures to exist.

```
    Class DocumentItem{
    public:

bool ReadFrom (HTMLStream& astream) =0 ;
bool WriteTo (HTMLStream& aStream) = 0 ; bool
DisplayOn (VisualObject* v ,

    const  Rectangle& aRect)= 0 ;

};
```

Figure 1: The DocumentItem protocol

The DocumentItem class also includes a standard means for displaying the item on a part of the user’s screen. Every DocumentItem subclass determines its own visual representation, and therefore carries its own implementation of the **DisplayOn** function, using polymorphism[7]. The **DocumentItem** class also includes support for layout of items in a visually pleasing manner. This includes algorithms for typesetting the item on the display. The visual and graphic aspects of the classes, however, are not addressed in this paper.

There are two special kinds of document items: linkable items and forms. Both are represented by classes derived from **DocumentItem.** The **LinkableItem** class represents a document item that is tagged as a hyperlink, with an associated URL. The **Hyperlink** class encapsulates an actual link; it contains a **LinkableItem** and a **ResourceSpec.** The **Hyperlink** encapsulates the knowledge of how to retrieve and utilize the resource associated with the URL in the link. The **Hyperlink** class also includes the (GUI-related) ability t o notify the application when the user clicks on its visual representation[8].

The **Form** is a class representing forms on a browser’s display, representing the Form entity in HTML. Most of the **Form’s** capability is built using YACL’s GUI classes, with text entry regions, buttons and other graphical user interface elements.

Customizing DocumentItem classes

As pointed out earlier, the **DocumentItem** class represents the abstract notion of an item in a document and need not be tied to any particular passive representation. Therefore, an agent application can derive customized classes from **DocumentItem** for its own needs[9][10]. This provides a natural way of extending the framework of classes, and mitigates the dependence on any particular passive form such as HTML or PDF.

IV. CONCLUSION

Over the last several years, the WWW has grown enormously and has become an invaluable source of information of all kinds. We focused on describing an object-oriented framework designed to ease the task of building customized WWW agents. Our framework exploits C++'s object-oriented features of encapsulation and polymorphism to achieve extensibility while preserving efficiency and ease of use.
We described the design and implementation of set of classes. In addition to the some properties, our classes preserve extensibility and reusability by making full use of the inheritance and polymorphism available in C++.
We also discussed about the framework which included networking capabilities. It does not yet provide distributed processing facilities in the style of CORBA. In subsequent work, we will investigate this issue to provide more focused work. Another aspect to be investigated is the design of a toolkit of custom Web objects to illustrate our ideas.

REFERENCES

[1] The Adobe Portable Document Format (PDF) IEEE Explore Version 4.01B, May 1995.

[2] Grady Booch, *Object-Oriented Design with Applications*, Benjamin-Cummings, 1995.

[3] Booch, G., "*Object-Oriented Development*", IEEE Transactions on Software Engineering 12(2), February 1996.

[4] Coad, P. and Yourdon, E., *Object-Oriented Analysis*, Prentice Hall, 1998.

[5] A. Dave, M . Sefika and R. Campbell, "*Proxies, application interfaces, and distributed systems*," Proc. Second Annual Workshop on Object Orientation an Operating Systems, Paris, France, September 2000, pp. 212-220.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements* of *Reusable Object-Oriented Software*, Addison-Wesley, 2005.

[7] L. Lamport, BQY: *A Document Preparation System*, Addison-Wesley, 2009.

[8] D.C . Schmidt and T . Suda, "*An object-oriented framework for dynamically configuring extensible*.

[9] *Distributed systems*," 1EE Distributed Systems Engineering Journal, 2011.

[10] M. A . Sridhar, *Building Portable* C++ *Applications with YACL*, Addison-Wesley, 2015.

Authors Profile

Anita Chagi, an Assistant Professor, School of Computing & IT, REVA University, Bangalore. Her research interest is in the area of Object Orientation with several languages, IoT and Machine Learning. She is also a Research contributor at RU.