Modeling Web Application Architectures with UML

By: Jim Conallen, Rational Software
June 1999
A version of this material appears in the October 1999 (volume 42, number 10) issue of Communications of the ACM.

Rational Software White Paper

Rational®
the e-development company™

Table of Contents

Abstract 1
Overview 1
Modeling 2
Web Application Architecture 3
Modeling Web Pages 4
Forms 7
Frames 8
Conclusion 9

Abstract

Web applications are becoming increasingly complex and mission critical. To help manage this complexity, they need to be modeled. Unified Modeling Language (UML) is the standard language for modeling software intensive systems. When attempting to model web applications with UML, it becomes apparent that some of its components don't fit nicely into standard UML modeling elements. In order to stick with one modeling notation for the entire system (web components and traditional middle tier components), UML must be extended. This paper presents an extension to the UML, using its formal extension mechanism. The extension is designed so that web-specific components can be integrated with the rest of the system's model, and to exhibit the proper level of abstraction and detail suitable for designers, implementers, and architects of web applications.

Overview

A new term has entered the IT vocabulary in the past few years: Web Application. It seems like everyone involved with business software systems has plans for building web applications, with many non-business related software efforts interested as well. For many early adopters of this architecture the term web application, like the systems themselves, have evolved from small successful web sites add-ons to robust n-tiered applications. It's not uncommon for a web application to service tens of thousands of simultaneous users, distributed all over the entire world. Architecting web applications is serious business.

When put to the test, the term web application has slightly different meanings to different people. Some believe a web application is anything that uses Java; others consider web applications anything that uses a web server. The general consensus is somewhere in between. For our purposes in this paper, we'll loosely define a web application to be a web system (web server, network, HTTP, browser) where user input (navigation and data input) affects the state of the business. This definition attempts to establish that a web application is a software system with business state, and that its "front end" is in large part delivered via a web system.

The general architecture of a web application is that of a client server system, with a few notable distinctions. One of the most significant advantages of a web application is its deployment. Deploying a web application is usually a matter of setting up the server side components on a network. No special software or configuration is required on the part of the client. Another significant difference is the nature of client and server communication. A web application's principal communication protocol is HTTP, which is a connectionless protocol that was designed for robustness and fault tolerance instead of maximum communication throughput. Communication between a client and server in a web application typically revolves around the navigation of web pages, not direct communications between server side and client side objects. At one level of abstraction, all messaging in a web application can be described as the request and reception of web page entities. Generally speaking, the architecture of a web application is not that much different from that of a dynamic web site.

The differences between a web application and a web site, even from a dynamic one, involve its usage. Web applications implement business logic, and its use changes the state of the business (as captured by system). This is important because it defines the focus of the modeling effort. Web applications execute business logic and so the most important models of the system focus on the business logic and business state, not on the presentation details. Presentation is important (otherwise the system wouldn't do anyone any good), however, you should strive for a clear separation between business and presentation concerns. If presentation issues are important, or even complicated, then they too should be modeled, but not necessarily as an integral part of the business logic model. Additionally the resources that work on presentation tend to be more artistic and less concerned with the implementation of business rules.

One methodology/notation associated with the development of Web Systems is the Relationship Management Methodology (RMM). RMM is a methodology for the design, construction, and maintenance of intranet and Internet web systems. Its principal goal is to reduce the maintenance costs of dynamic, database-driven web sites. It advocates a visual representation of the system to facilitate design discussions. It is an iterative process that includes the decomposition of the visual elements in the web pages, and their association with database entities. RMM is a "soup to nuts" approach to the creation and maintenance of dynamic web sites.

RMM falls short when building web applications. Web applications, being business logic centric, include a number of technological mechanisms for implementing business logic that are not adequately covered by RMM notation. Such technologies as client side scripting, applets, and ActiveX controls often make significant contributions to the execution of the system's business rules. Additionally web applications can be used as a delivery mechanism for a distributed object system. Applets and ActiveX controls can contain components that asynchronously interact with server side components via RMI or

DCOM, independent of the web server. Sophisticated applications also make use of multiple browser instances and frames on the client, which establish and maintain their own communication mechanisms.

Since all of these mechanisms contribute to the business logic of the system, they need to be modeled as such. Additionally because they only represent part of the business logic, they need to be integrated with the rest of the system's models. In many situations, the bulk of business logic is executed behind the web server in one of the server side tiers. The choice of modeling language and notation is typically decided by the needs of this side the application. With the acceptance of UML by the OMG as an official object modeling language, more and more systems are being expressed with UML notation. For many, UML is the language of choice for modeling software intensive systems. The main issues in modeling web applications then becomes: "How do I express the business logic being executed in my web-specific components alongside the rest of my application?" The answer lies in our ability to express the execution of the system's business logic in those web-specific elements and technologies with UML.

This paper is intended as an introduction to the issues and possible solutions for modeling web applications. It focuses on the architecturally significant components particular to web applications and how to model them with UML. It is assumed that the reader is familiar with UML, object-oriented principals, and web application development. The work described in this paper is based on some fairly innocuous assumptions:

- Web applications are software intensive systems that are becoming more complex, and are inserting themselves in more mission critical roles.
- One way to manage complexity in software systems is to abstract and model them.
- A software system typically has multiple models, each representing a different viewpoint, level of abstraction, and detail.
- The proper level of abstraction and detail depends on the artifacts and activities in the development process.
- The standard modeling language for software intensive systems is the Unified Modeling Language (UML).

A fuller treatment of the concepts and ideas expressed in this paper are being developed in an upcoming book: "Building Web Applications with UML" expected to be published in the Object Technology Series by Addison Wesley Longman later this year.

Modeling

Models help us understand the system by simplifying some of the details. The choice of what to model has an enormous effect on the understanding of the problem and the shape of the solution. Web applications, like other software intensive systems, are typically represented with a set of models; use case model, implementation model, deployment model, security model, and so on. An additional model used exclusively by web systems is the site map, an abstraction of the web pages and navigation routes throughout the system.

Most modeling techniques practiced today are well suited to development of the various models of a web application, and do not need further discussion. One very important model however, the Analysis/Design Model (ADM), does present some difficulties when an attempt is made to include web pages, and the executable code associated with them, alongside the other elements in the model.

When deciding how to model something, determining the correct level of abstraction and detail is critical to providing something that will be of benefit to the users of the model. Generally speaking it's best to model the artifacts of the system—those real life entities that will be constructed and manipulated to produce the final product. Modeling the internals of the web server, or the details of the web browser, is not going to help the designers and architects of a web application. Modeling the pages, their links to each other, all the dynamic content that went into creating the pages, and the dynamic content of the pages once on the client is important—very important. It's these artifacts that designers design and implementers implement. Pages, hyperlinks, and dynamic content on the client and server are what need to be modeled.

The next step is mapping these artifacts to modeling elements. Hyperlinks, for example, map naturally to association elements in the model. A hyperlink represents a navigational path from one page to another. Extending this thought, pages might map to classes in the logical view of the model. If a web page were a class in the model, then the page's scripts would map naturally to operations of the class. Any page-scoped variables in the scripts would map to class attributes. A problem arises when you consider that a web page may contain a set of scripts that execute on the server (preparing the dynamic content of the page) and a completely different set of scripts that only execute on the client (that is, JavaScript). In this scenario when we look at a web page class in the model, there is confusion over what operations, attributes, and even relationships are active on

the server (while the page is being prepared) and which ones are active when the user is interacting with the page on the client. Additionally a web page as delivered in a web application is really best modeled as a component of the system. Simply mapping a web page to a UML class does not help us understand the system better.

The creators of the UML realized that there would always be situations where the UML, out of the box, would not be sufficient to capture the relevant semantics of a particular domain or architecture. To address this purpose, a formal extension mechanism was defined to allow practitioners to extend the semantics of the UML. The mechanism allows us to define *stereotypes*, *tagged values,* and *constraints* that can be applied to model elements.

A *stereotype* is an adornment that allows us to define a new semantic meaning for a modeling element. *Tagged values* are key value pairs that can be associated with a modeling element that allow us to "tag" any value onto a modeling element. *Constraints* are rules that define the well formedness of a model. They can be expressed as free-form text or with the more formal Object Constraint Language (OCL).

The work discussed in this paper introduces an extension to UML for web applications. This extension, in its entirety, is beyond the scope of this paper, however, most of the concepts and explanations are discussed here.

One final point on modeling—a very clear distinction needs to be made between business logic and presentation logic. For the typical business application, only the business logic should be part of the ADM. Presentation details like animated buttons, fly-over help, and other UI enhancements do not normally belong in the ADM. If a separate UI model is constructed for the application, then this is the place for such things. The ADM needs to remain focused on the expression of the business problem and solution space. In this day of web artists, the look and feel of a web page is best designed and implemented by a specialist (technical graphic artist) and not by the traditional developer.

Web Application Architecture

The basic architecture of a web application includes browsers, a network, and a web server. Browsers request "web pages" from the server. Each page is a mix of content and formatting instructions, expressed with HTML. Some pages include client side scripts that are interpreted by the browser. These scripts define additional dynamic behavior for the display page and often interact with the browser, page content, and additional controls (Applets, ActiveX controls, and plug-ins) contained in the page. The user views and interacts with the content in the page. Sometimes the user enters in information in field elements in the page and submits them to the server for processing. The user can also interact with system by navigating to different pages in the system via hyperlinks. In either case, the user is supplying input to the system that may alter the "business state" of the system.

From the client's perspective, the web page is always an HTML formatted document. On the server, however, a "web page" may manifest itself in a number of different ways. In the earliest web applications, dynamic web pages were built with the Common Gateway Interface (CGI). CGI defines an interface for scripts and compiled modules to use to gain access to the information passed along with a page request. In a CGI based system, a special directory is typically configured on the web server to be able to execute scripts in response to page requests. When a CGI script is requested, the server, instead of just returning the contents of the file (as it would for any HTML formatted file), processes or executes the file with the appropriate interpreter (usually a PERL shell) and streams the output back to the requesting client. The ultimate result of this processing is an HTML formatted stream that is sent back to the requesting client. Business logic is executed in the system while processing the file. During that time, it has the potential to interact with server side resources such as databases and middle tier components.

Today's web servers have improved upon this basic design. Today they are much more security aware, and include features like management of client state on the server, transaction processing integration, remote administration, and resource pooling to name just a few. Collectively the latest generation of web servers is addressing those issues important to architects of mission critical, scalable, and robust applications.

When looking at the role of CGI scripts, today's web servers can be divided into three major categories: scripted pages, compiled pages, and a hybrid of the two. In the first category, each web page that a client browser can request is represented on the web server's file system as a scripted file. This file is typically a mix of HTML and some other scripting language. When the page is requested, the web server delegates the processing of this page to an engine that recognizes it, with the ultimate result that an HTML formatted stream is sent back to the requesting client. Examples of this are Microsoft's Active Server Pages, Java Server Pages, and Cold Fusion.

In the second category, compiled pages, the web server loads and executes a binary component. This component, like with scripted pages, has access to all the information that came along with the page request (values of form fields and parameters). The compiled code uses the request details, and typically accesses server side resources to produce the HTML stream returned to the client. Although not a rule, compiled pages tend to encompass a larger functionality than scripted pages. By passing parameters to the compiled page request, different functionality can be obtained. Any one compiled component may

actually include all the functionality of an entire directory's scripted pages. The technologies that represent this type of architecture are Microsoft's ISAPI and Netscape's NSAPI.

The third category represents scripted pages that, once requested, are compiled and this compiled version is then used thereafter by all subsequent requests. Only when the original page's contents change, while the page undergoes another compile. This category is a compromise between the flexibility of scripted pages and the efficiency of compiled pages.

Modeling Web Pages

Web pages, either scripted or compiled, map one-to-one to components in UML. A component is a "physical" and replaceable part of the system. The Implementation View (Component View) of the model describes the system's components and their relationships. In a web application, this view describes all the web pages of the system, and their relationships with each other (that is, hyperlinks). At one level, a component diagram of a web system is like a site map.

Since components only represent the physical packaging of interfaces, they are not suitable for modeling the collaborations inside of the pages. This level of abstraction, extremely important to the designer and implementer, still needs to be part of the model. To start, we could say that each web page is a UML class in the model's Design View (Logical View), and that its relationships to other pages (associations) represent hyperlinks. However, this abstraction breaks down when you consider that any given web page can potentially represent a set of functions and collaborations that exist only on the server, and that a completely different set may exist only on the client. Any server scripted web page that employs Dynamic HTML (client side scripting) as part of its output is an example of such a page. The knee-jerk reaction to this problem might be to stereotype each attribute or operation in the class to indicate whether or not it was valid on the server or client side. At this point, our model, originally intended to help simplify things, is getting quite complex.

A better approach to the problem is to consider the principal of "separation of concerns". Logically speaking, the behavior of a web page on the server is completely different than on the client. While executing on the server, it has access to (that is, relationships with) server side resources (middle tier components, databases, file system, and so on). That same page, or the streamed HTML output of that page, on the client has a completely different behavior and set of relationships. On the client, a scripted page has relationships with the browser itself via the Document Object Model (DOM) and with any Java Applets, ActiveX controls or plug-ins that the page specifies. For the serious designer, there can be additional relationships with other "active" pages on the client that appear in another HTML frame or browser instance.

Separating concerns, we can model the server side aspect of a web page with one class, and the client side aspect with another. We distinguish the two by using UML's extension mechanism to define stereotypes and icons for each—«server page» and «client page». Stereotypes in UML allow us to define new semantics for a modeling element. Stereotyped classes can be rendered in a UML diagram with either a custom icon, or simply adorned with the stereotype name between guillemets («»). The icons are useful for overview diagrams, where using the simple tags is best when class attributes and operations are exposed.

For web pages, the stereotypes indicate that the class is an abstraction of the logical behavior of a web page on either the client or the server. The two abstractions are related to each other with a directional relationship between the two. This association is stereotyped as «build», since it can be said that a server page builds a client page (Figure 1). Every dynamic web page (that is, pages whose content is determined at runtime) is constructed with a server page. Every client page is built by, at most, a single server page; however, it's possible for a server page to build multiple client pages.

A common relationship between web pages is the hyperlink. A hyperlink in a web application represents a navigation path through the system. This relationship is expressed in the model with a «link» stereotyped association. This association always originates from a client page and points to either a client or server page.

Hyperlinks are implemented in the system as a request for a web page, and web pages are modeled as components in the Implementation View. A link association to a client page is, for the most part, equivalent to a link association to the server page that builds the client page. This is because a link is actually a request for a page, not either of the class abstractions. Since a web page component realizes both page abstractions, a link to any of the classes realized by the page component is equivalent.

Tagged values are used to define the parameters that are passed along with a link request. The «link» association tagged value "Parameters" is a list of parameter names (and optional values) that are expected and used by the server page that processes the request. In Figure 2, the SearchResults page contains a variable number of hyperlinks (0..*) to the GetProduct server page, where each link has a different value for the productId parameter. The GetProduct page builds the ProductDetail page of the product specified by the productId parameter.

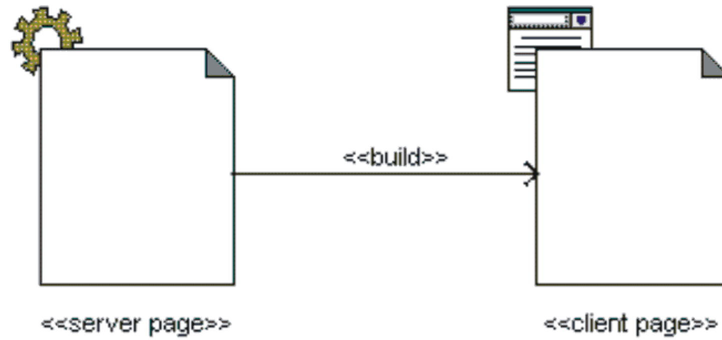

Figure 1. Server pages build client pages.

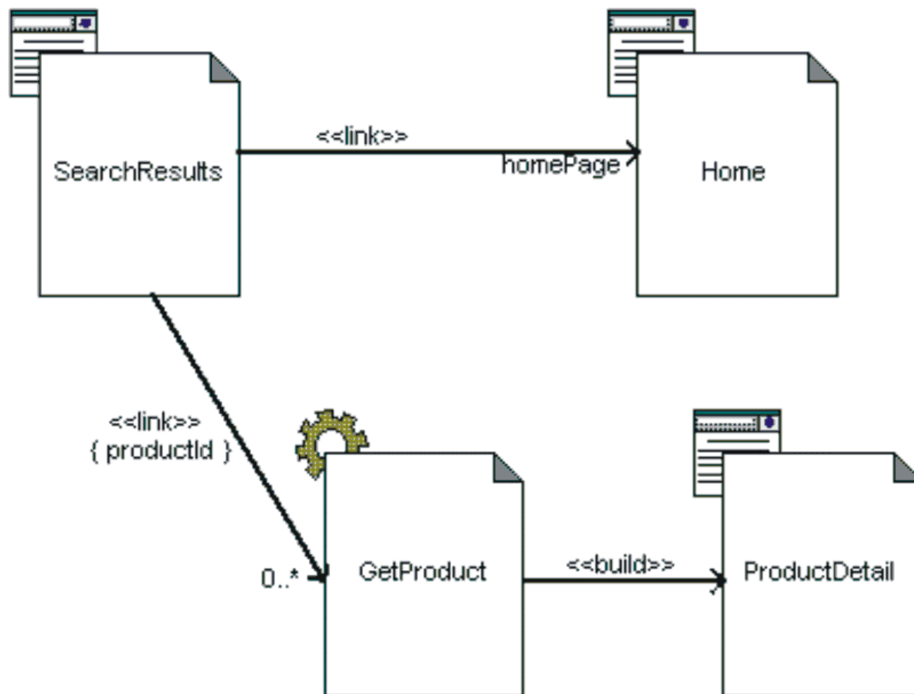

Figure 2. Using hyperlink parameters.

Using these stereotypes, it makes it easier to model a page's scripts and relationships. The «server page» class' operations become functions in the page's server side scripts, and its attributes become page-scoped variables (globally accessible by the page's functions). The «client page» class' operations and attributes likewise become functions and variables visible on the client. The key advantage of separating the server and client side aspects of a page into different classes is in the relationships between pages and other classes of the system. Client pages are modeled with relationships to client side resources: DOM, Java Applets, ActiveX controls, and plug-ins (Figure 3). Server pages are modeled with relationships to server side resources—middle tier components, database access components, server operating system, and so forth, illustrated in Figure 4.

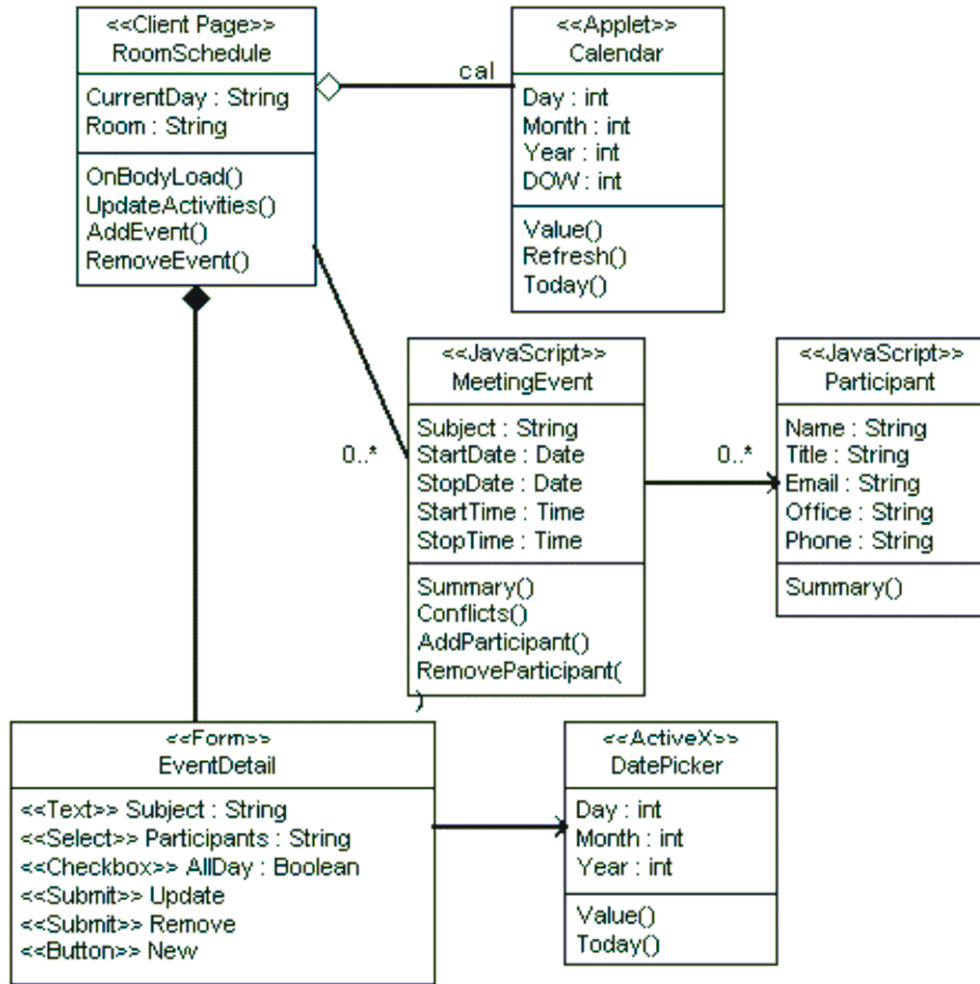

Figure 3. Client Collaborations

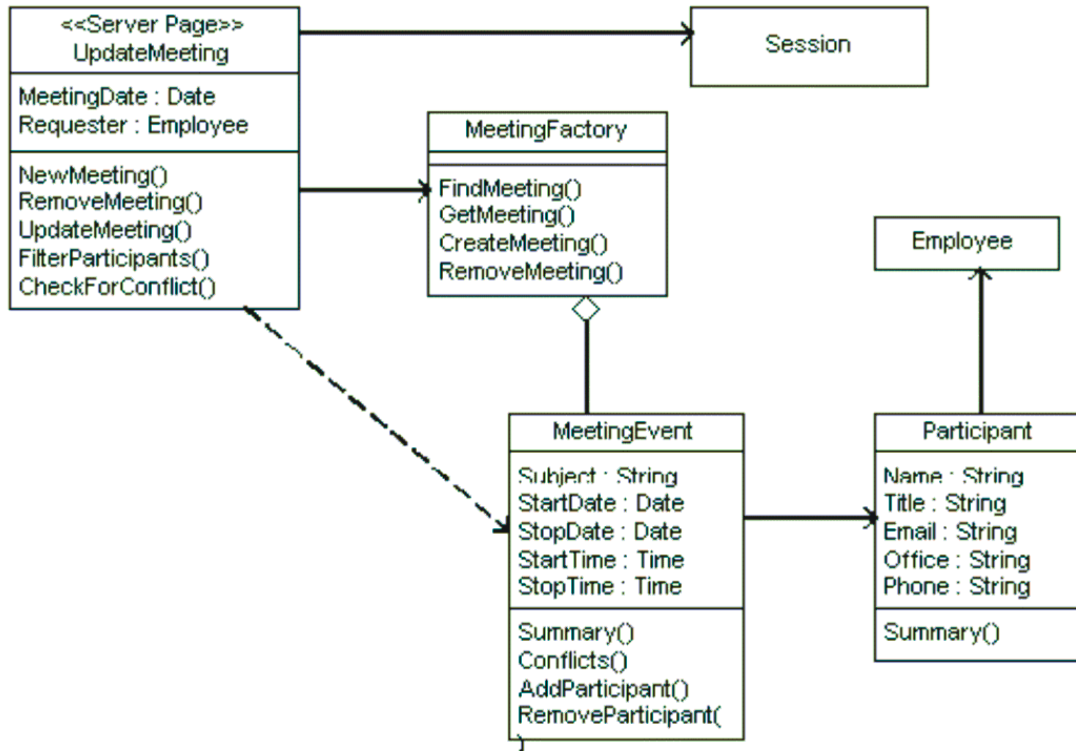

Figure 4. Server Collaborations

One of the biggest advantages of using class stereotypes to model the logical behaviors of web pages is that their collaborations with the server side components can be expressed in much the same way as any other server side collaborations. The «server page» is simply another class that participates in the business logic of the system. At a more conceptual level, server pages typically take on the role of controllers, orchestrating the necessary business object activity to accomplish the business goals initiated by the browser's page request.

On the client side, collaborations can get a little complicated. This is due in part to the variety of technologies that can be employed. A client page, at its simplest, is an HTML document that contains both content and presentation information. Browsers render HTML pages using the formatting instructions in the page, sometimes with separate style sheets. In the logical model, this relationship can be expressed with a dependency from a client page to a «Style Sheet» stereotyped class. Style sheets, however, are principally a presentation issue and are often left out of the ADM

Forms

The principal data entry mechanism for web pages is the Form. Forms are defined in an HTML document with <form> tags. Each form specifies the page to which it is to submit itself. A form contains a number of input elements, all expressed as HTML tags. The most common tags are the <input>, <select>, and <textarea> tags. The input tag is somewhat overloaded since it can be a text field, checkbox, radio button, push button, image, hidden field, as well as a few other less common types. Modeling forms means another class stereotype: «Form». A «Form» has no operations, since any operations that might be defined in a <form> tag are really owned by the client page. A form's input elements are all stereotyped attributes of the «Form» class. A «Form» can have relationships with Applets or ActiveX controls that act as input controls. Each form also has a relationship with a server page—the page that processes the form's submission. This relationship is stereotyped «submit». Since forms are completely contained in an HTML document, they are expressed in a UML diagram with a strong form of aggregation. Figure 5 shows a simple shopping cart page that defines a form and shows the submit relationship to the processing server page.

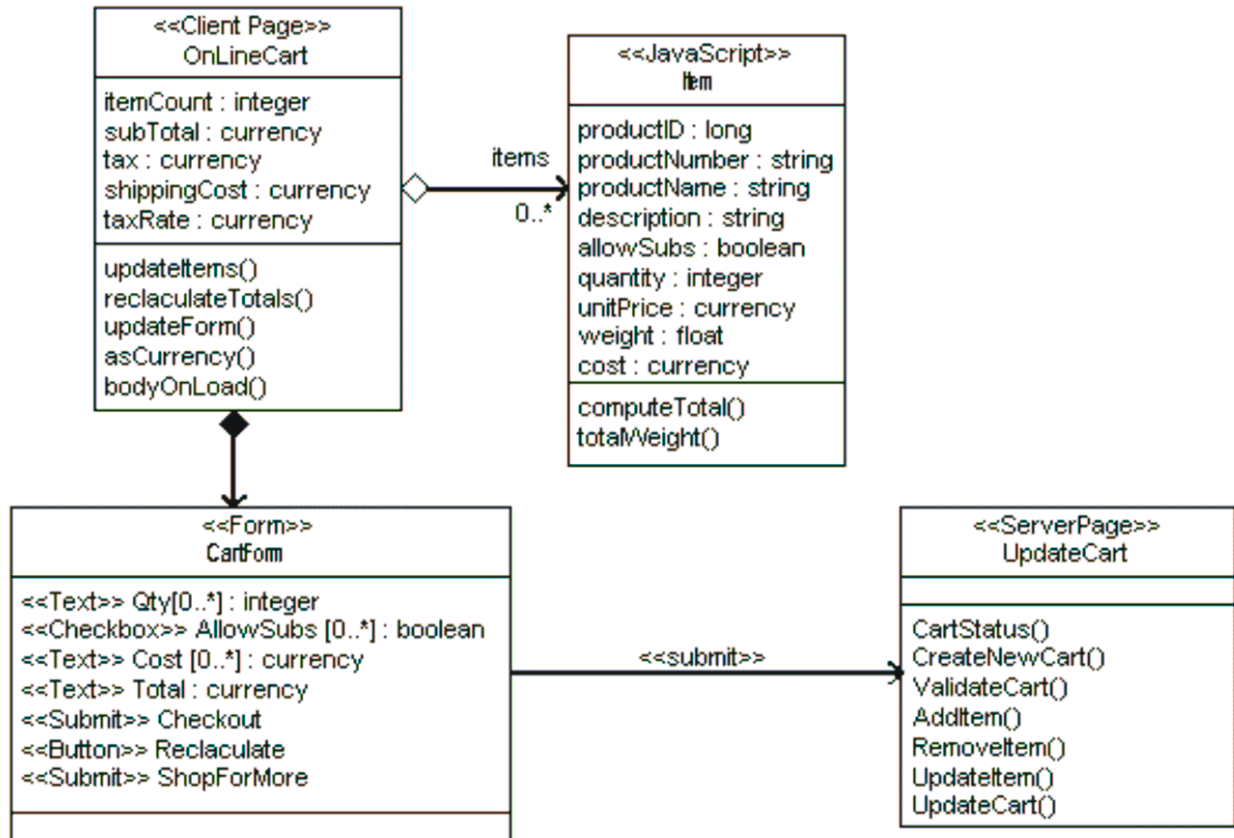

Figure 5. Forms submit to Server Pages.

In Figure 5, the «JavaScript» stereotyped class is an object that represents the items in the shopping cart. Array syntax is used in the description of the form's properties for those fields that have a variable number of instances. In the case of this shopping cart, it means that the cart can have zero to many items, each with a Qty, AllowSubs, Cost, and Total <input> element.

Since all the activity in the client page is executed with JavaScript, and JavaScript is a type-less language, the data types specified for any of these attributes are used only for implementer clarification. When implemented in JavaScript or as HTML input tags, the type is ignored. This also applies to function parameters, which, although not fully displayed in this figure, are part of the model.

Frames

The use of HTML Frames in a web site or application has been a subject of polarized debate since its introduction. Frames allow multiple pages to be active and visible to the user at any given time. The latest feature set for the most common browsers today also allow multiple browser instances to be active on the user's machine. Using Dynamic HTML scripts and components, these pages can interact with each other. The potential for complex interactions on the client is significant, and the need for modeling this even greater.

Whether frames or multiple browser instances are employed in an application, is a decision for the software architect. If it is then, then, for the same reasons as before, the model of this client side behavior needs to be represented in the ADM. To model frame usage, we define two more class stereotypes—«frameset » and «target»—and an association stereotype «targeted link ». A frameset class represents a container object and maps directly to the HTML <frameset> tag. It contains client pages and targets. A target class is a named frame or browser instance that is referenced by other client pages. A targeted link association is a hyperlink to another page, but one that gets rendered in a specific target. In the example shown in Figure 6, a common outline view is presented in a browser using two frames. One frame is named with a target (Content), whereas the other frame simply contains a client page. This client page frame is the book's table of contents (TOC). Hyperlinks in this

page are targeted, so that they render in the Content frame. The effect is a static table of contents on the left-hand side, and the books contents, chapter-by-chapter, on the right-hand page.

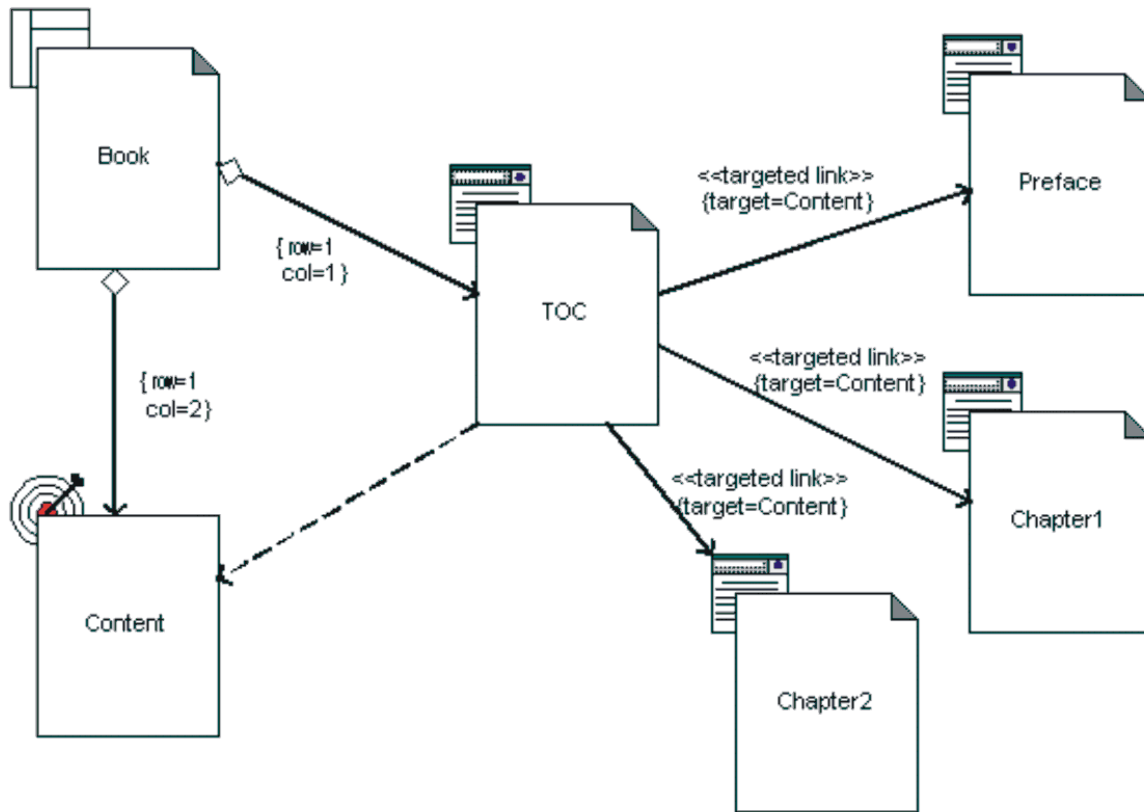

Figure 6. Frames Example

Much of the actual presentation specifics are captured by tagged values in the frameset and the associations. Two tagged values on the aggregation relationship between a frameset and a target, or client page, specify the frameset row and column in which the target, or page, belong. The tagged value "Target" on the targeted link association identifies the «target» where the page should be rendered.

When a target is not aggregated with a frameset, it means that a separate browser instance is used to render the pages. It's important to keep in mind that this notation is expressing a single instance of a client machine. Multiple independent targets are all assumed to be running on the same machine, and the diagram expresses the client side behavior of one client instance. Any other deployment configuration would need to be heavily documented in the model for better understanding.

Conclusion

The ideas and concepts discussed in this paper are an introduction to issues and solutions for modeling web application-specific elements with UML. The goal of this work is to present a coherent and complete way to integrate modeling web specific elements with the rest of the application, such that the levels of detail and abstraction are appropriate for designers, implementers, and architects of web applications. A first version of a formal extension to the UML for web applications is near completion. This extension will provide a common way for architects and designers to express the entirety of their web applications design with UML.

The most recent information on this extension can be found on the Internet in the Rational Rose and UML sections of Rational Software's web site.

Rational®
the e-development company™

Corporate Headquarters
18880 Homestead Road
Cupertino, CA 95014
Toll-free: 800-728-1212
Tel: 408-863-9900
Fax: 408-863-4120
E-mail: info@rational.com
Web: www.rational.com

For International Offices: www.rational.com/worldwide

Rational, the Rational logo, Rational the e-development company and Rational Rose are registered trademarks of Rational Software Corporation in the United States and in other countries. Microsoft, Microsoft Windows, Microsoft Visual Studio, Microsoft Word, Microsoft Project, Visual C++ and Visual Basic are trademarks or registered trademarks of Microsoft Corporation. All other names used for identification purposes only and are trademarks or registered trademarks of their respective companies. ALL RIGHTS RESERVED. Made in the U.S.A.

© Copyright 2000 Rational Software Corporation.

TP-#### /00. Subject to change without notice.