Annals of Software Engineering 13, 203–230, 2002
© 2002 Kluwer Academic Publishers. Manufactured in The Netherlands.

Engineering Web-Based Systems with UML Assets

GRANT LARSEN glarsen@rational.com
Rational Software Corporation, 10632 W. Ontario Ave., Littleton, CO 80127, USA

JIM CONALLEN jconallen@rational.com
Rational Software Corporation, 181 Washington St. Suite 525, Conshohocken, PA 19428, USA

Abstract. Software development is a product of evolution. It builds on the experiences of the individual and the community, continually refining every aspect of process. Web centric architectures are no different. Their ever-increasing levels of functionality and complexity are evident in the latest generations of web applications. In this paper we discuss some standards that enable teams to build, communicate and apply their experiences in the art of web application software development. These standards appropriate various levels of abstraction, and bring the focus of web application development back to the team. Combining UML representation of web-based systems with techniques for packaging these solutions for reuse can positively impact development timelines and efficiencies.

1. Introduction

Several challenges that continue to face software engineers developing web-based systems include:

(a) the increasing complexity of software solutions;

(b) the increasing demand for such systems; and,

(c) the high cost of delivering those solutions.

These issues clearly make web application development a team activity. Since web application development at its highest level is no different than any other type of software development it is clear that in order to make teams effective, standards need to be in place for the communication, exchange and overall understanding of the artifacts and issues involved in the development of software with complex web centric architectures. It is a fundamental assertion of this paper that an adoption of two such standards; the Unified Modeling Language (UML), with the Web Application Extension (WAE) profile for expressing designs, and the Reusable Asset Specification (RAS) for the exchange reusable bundles of design and code can accelerate teams building such systems.

The first standard, the UML allows us to express complex software structures and behaviors visually and in a way that makes it easier to understand. The UML is already the defacto standard in the design community. Together with the WAE, a profile for the UML that addresses the modeling of some web application specific issues, the UML/WAE can be effectively used to express web application architectures and designs

at various levels of abstraction and detail. The second standard, RAS defines how we can package bundles of software artifacts that as a unit represent something greater than the sum of its parts; an Asset. Assets represent a higher level of artifact that can be reused not only in the same software project but also across multiple projects.

Collectively these two standards and their supporting technology and sub-standards enable teams to express, communicate and exchange software assets and components. This article introduces the standards themselves and discusses the workflows involved in putting them to use.

This article assumes a working knowledge of the UML and a fundamental understanding of web centric system designs. Some of the discussions in this article may require more detailed knowledge of web architectural elements and of XML. Upon reading this article you will have a high-level understanding of how to organize reusable solutions for web-based system using the UML.

2. Web-based systems and the UML

Web applications have been receiving a lot of attention the past few years. They offer some pretty significant architectural advantages that include but are not limited to:

- Execution in an unknown and largely heterogeneous environment.
- Deployment typically doesn't require client side activities.
- Massively scalable.
- Fault tolerant client and server communications.

Of course selecting a web centric architecture in itself doesn't guarantee the above advantages will be met. Even web applications can be poorly architected, or fail for unknown technical reasons [Earls 1999].

Looking at a high level of abstraction, web based systems are architecturally just specializations of client–server systems. The key elements remain the client, the server and the connecting network. The differences only appear a lower levels of detail and typically are related to the principal communications protocol HTTP, and the vendor specific extensions to the browser client software. Additional non-technical issues include expectations in the user interface paradigms, content management and in general the technology churn so rampant in Internet-based web applications.

When the industry made its painful move from mainframe based systems to client–server systems, a lot of lessons were learned about the very nature of software development. This era of computing saw the development of ever increasingly complex software systems, and the ubiquitous use of computers and software in nearly every aspect of business life. To manage this complexity the industry collectively identified a set of best practices. These best practices vary slightly from one organization to another and from one vendor to another, but for the most part include the following principal elements [Jacobson 1999; Kruchten 2000]:

- Iterative development.

- Component architectures.
- Use Case (User Centered) requirements.
- Visual modeling.

An iterative software development process identifies and recognizes some things that developers have know for along time. People make mistakes, and sometimes mistakes or missing information isn't known until you're already downstream in the process. An iterative development process just acknowledges that artifacts and assumptions created early on might need to be changed as a result of newly discovered information.

A component-based architecture simply embodies the principals of encapsulation and modularity. Component models (i.e., COM, JavaBeans, SOAP/Web Services) provide the standards and infrastructure to make component-based architectures easier to build and deploy. Practical component reuse is a direct result of these component interoperability standards. Such reuse will be broader and will have greater impact on your projects if there is a standard for describing the nature and structure of reusable items.

Use Cases and User Centered are important to nearly all client–server systems, especially web-based ones. Many web-based systems have anonymous users, who have never had the benefit of attending a training seminar as the new application is deployed. With the reach of computer systems extending farther and farther in to society, the need for intuitive and predictable user interfaces is even more critical. Use Cases keep us focused on delivering meaningful and understandable systems directly to the user.

People have been building complex things for years and techniques for managing complexity have not changed much over the years. Two principal strategies for managing complexity are to:

- divide and conquer;
- model (visually).

The use of component architectures is an example of how divide and conquer and can help manage the creation of complex software, but even most components these days are individually too complex for a human to comprehend in its entirety.

Modeling things lets us understand them at different levels of abstraction and levels of detail. A model is representation and a simplification something. It describes a thing from a particular viewpoint and at a particular level of abstraction and detail. Visual models do so with pictures and graphical representations. The Unified Modeling Language (UML) [Booch 1999; Rumbaugh 1999] is a visual modeling language that is presently the de-facto standard for modeling software intensive systems. It is the result of a consolidation of the efforts of Grady Booch (Booch Method), Jim Rumbaugh (Object Modeling Technique, OMT) and Ivar Jacobson (Object Oriented Software Engineering and Use Cases), and is now managed by the Object Management Group (OMG).

The UML is a modeling language that was born out of the object oriented analysis and design community. It specifies how systems can be visually represented in structural diagrams (class, component, deployment, object) and in behavioral diagrams (use case, sequence, collaboration, statechart, activity).

Its creators had standard object oriented languages in mind, but were wise enough to provide a mechanism to extend the language to meet the special needs of future systems. This extension mechanism is used to add new semantics to the core language, that are suitable to modeling different types of systems at appropriate levels of abstraction and detail [Alhir 1999].

This flexibility must be used carefully. Extending any language with new semantics requires careful thought, and to be successful must be internally consistent, and ideally it must be interoperable with the core semantics, and with other extensions that are likely to be used in describing any one system.

The key to successful modeling is to do so at appropriate levels of abstraction and detail from a specific viewpoint. Abstractions that are too high or too low confuse our understanding of what is important. Inappropriate levels of detail can either misguide us into underestimating the level of effort, or into building rigid and intolerant systems. Above all, models are to help us understand something. If our models do not make that easier then we are modeling the wrong things or levels of detail.

3. User experience-level representation for web based systems

Some of the earliest models of a system are the use case models. These give us a view into the system from the user's point of view and capture the system's functional requirements in a formal way. Each use case describes the functionality of the system in terms of a dialog and a scenario. Each scenario represents an expected path through the system as a dialog of activity that describes the user's actions and the system's responses.

One of the first analysis and design activities in a system is to elaborate the use case model in to objects. In addition to representing classic server tier objects, it is apparent that in web centric applications that the individual web screen is a first class object, each with their own unique states and behaviors. This is where some of the influences of a web centric architecture begin to present themselves. The popularity of Internet based web applications has given rise to the development of common user interface paradigms. The field of Information Architecture (IA) is an example of new specialization of development team member whose focus extends the traditional Human Interaction (HI) skills and branches out to include the entirety of the "user experience".

A term User Experience (UX) is receiving considerable attention in web application development circles. It is used to describe the team and activities of those specialists responsible for keeping the user interface consistent with current paradigms and most importantly appropriate for the context in which the system is expected to run in. The UX team and in particular the Information Architect is responsible for creating the look and feel of the application, determining principal navigational routes through the system's web pages, and for managing/organizing the structure of the content in the pages. The artifacts that the UX team produces includes, in addition to others:

- Screens and Content.
- Storyboard scenarios.

- Navigational paths through the screens.

A Screen is something that is presented to the user. The term *screen* is used instead of the more common term *page*, since a screen of information may contain multiple and independent pages of information. A page in this sense maps roughly to a URL, and it is independently requestable. A screen on the other hand is simply the end result of combining a number of pages into a consistent and coherent user interface.

In addition to any user interface infrastructure (menus, controls, etc.) a screen contains business relevant content. Content is the generic term for business information that appears in a web page. It is a combination of static content (field names, titles, text and images that are constant for each user of the system), and dynamic content (selected products, personal information, current status and other computed information).

One important distinction between screens and the mechanisms that build and produce screens, is that a screen is strictly what is presented to the user. How it got there is not an inherent property of a screen. Usually a screen is built and presented to the user by server side mechanisms (Java Server Pages, Servlets, Active Server Pages, etc.). These mechanisms often interact with server side components that produce the dynamic content in the screens. The static content is provided by templates and usually resides on the server's file system. This combination of template and dynamic content is what builds screens. It is also important to understand that whether a screen is produced as the result of a JSP's processing, or dynamically assembled on the client from an XML document the resulting user interface experienced by the user is still a screen filled with content.

When screens are combined into scenarios, they express mini stories of the application's usage. In a given "story" any particular screen may be visited many times (where each time it may have a new set of dynamic data). Each scenario is an expression of a very specific use of the system. There are no conditional expressions in scenarios, and actual business and domain terms and realistic phrases are used in the documentation. The whole goal of a storyboard scenario is to express a typical use of the system through the eyes of the user. Early on in the development process, the screens might be simple HTML prototypes, or even hand drawn diagrams. As the process continues these artifacts evolve into higher fidelity mockups or actual HTML files.

Eventually these HTML files, or templates make their way into the actual application and are delivered by web servers and embedded with actual dynamic content. This integration of the UX presentation artifacts with the business presentation artifacts needs to be done often and periodically. It is generally efficient to have the UX and engineering teams work independently, however without periodic and frequent sync points, the integration events might become troublesome.

One of the most architecturally important artifacts that the UX team produces is the navigational path map. This diagram(s) expresses the structure of the screens in an application with their potential navigational pathways. It can be thought of as a road map of the application's screens. An important characteristic of this diagram is that it expresses all of the legal paths through the system. The influence of the browser's Back button, or the caching of previously navigated to pages does not belong in this diagram.

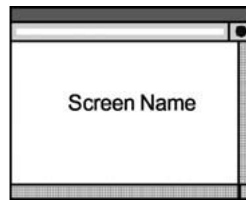

Figure 1. «Screen» class stereotype icon.

They are important issues to be considered when designing and architecting the system, but not something that belongs in the navigational map.

Capturing the UX design in UML is done via an extension to the UML (i.e., stereotypes, tag values, constraints). What follows is a description of this extension. It is important to note that this extension and the models its produces are not at a level of detail or abstraction that lend themselves to automated forward or reverse engineering. They allow us to model the architecturally significant UX team's contributions at a relatively high level. It is fully expected that the UX model will be referenced by more detailed design models, and that they will form, in part, a path of traceability from detailed design models to the even higher level use case model.

3.1. UX model extension

A screen can be represented in a UML model with a «screen» stereotyped class.[1] It is also possible to give this new stereotyped class its own icon (figure 1). When working with screens one of the pieces of information that is important to both the UX team and the engineering team is the dynamic content in the screen. This dynamic content is best expressed as attributes of the screen class. Figure 2 shows an alternative way of expressing a «screen» element with its attributes exposed. Since the screen class is a requirements or analysis level modeling element, expressing the dynamic content of the screen can be relatively free formed. Strict data types and valid identifier names are not necessary to just convey the dynamic content of the screen, that's stuff better expressed in the detailed design models. In this model, it is sufficient to identify the dynamic content by name, and perhaps with a short description.

In addition to the dynamic content expressed as attributes in a «screen» stereotyped class, operations can be defined and expressed in the model. Operations of a «screen» indicate things that a user can do in and with this screen. These user actions or gestures are a way of capturing some of the responsibilities and expected actions of users of the system. When an operation is tagged as being static (i.e., underlined) then this operation represents something that the system does, typically during the creation of the screen instance.

Additional adornments can be added to «screen» elements in the model to express additional properties of a screen. For example the sample screens provided in this article

[1] The extensions to the UML presented in this article are part of the next version of the Web Application Extension profile to the UML.

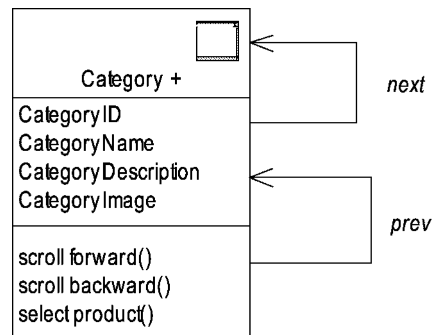

Figure 2. «Screen» UML element with attributes shown.

use an exclamation sign postfix (!) to indicate that the screen is navigable from any other page in the system (most likely from a common menu bar in the application user interface). Another adornment that is useful is the plus sign (+), which when appended to the screen name indicates that the page is a Paged List. Paged Lists are screens that allow user to navigate through large lists of information where there is a limit to the number of items that can be displayed on any one page. Results from Internet search engines are examples this type of mechanism. Other adornments can be used to express secure pages (SSL/TLC/SET). Officially these adornments are captured in the UML as tagged values, however at this level of abstraction it is often convenient to adorn the class icon or class name with the information to make the diagrams easier to understand. Using color is another useful way to impart this information in diagrams.

In addition to the «screen» stereotype, a «screen form» class stereotype is used to identify HTML forms that are contained in a screen. A separate class is required to model HTML forms because in any given screen there is the possibility that multiple forms might exist, each submitting themselves to a different next screen. Separate «screen form» classes are also useful for clearly identifying related groups if input fields. Figure 3 shows how a Cart screen contains an embedded form called LineItemsForm. This form contains an array of RemoveProduct checkboxes and an array of Quantity text input fields. The form itself points back to the Cart screen since it represents posted data that when processed by the server will result in the return of the updated Cart screen. This diagram implies that in a given Cart screen there are a number of checkboxes that can be submitted to server, where they are processed and the Cart page returned. The practical result of this is that each line item in the cart will have a checkbox that when checked will remove the item from the cart.

It is also important to note that there is no submit button indicated in the form. This level of detail is not really required at this level of abstraction.[2] In an actual design model, there would most likely be an HTML submit button, and possibly a number of other input fields in the form, but from the UX team's point of view these are not required

[2] This of course is an arbitrary distinction, and depending upon the development team, inclusion of submit buttons and even hidden fields may be appropriate for the given situation.

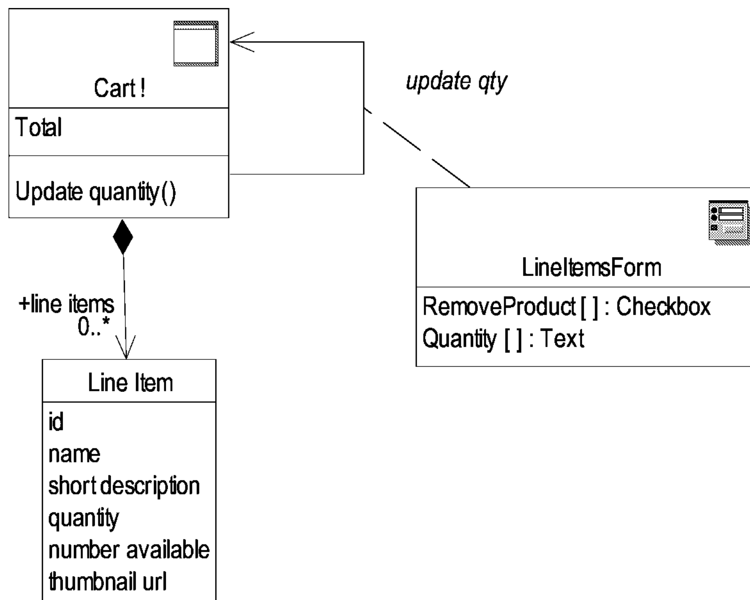

Figure 3. A screen that contains an HTML form.

to convey the important aspects of the user interface design. There is an additional class encapsulating a single line item of dynamic data. Since the Cart screen's dynamic data contains a variable number of line item data in addition to a single total price value, it is convenient to model the data bundled in a single class that as a group is optionally repeated in the screen.

A proper system model should explicitly specify mappings of screens and analysis/design elements (figure 4). In this figure the analysis objects are represented with stereotyped elements suggested in Jacobson's [1992] book. This link in the chain of traceability identifies the actual classes of objects that are responsible for delivering the screens in the runtime system. Farther down the process there might be similar mappings or linkages that connect these analysis classes to detailed design classes and components.

The UX team's storyboards are screens linked together to provide the stepping-stones through which a use case scenario is realized. This is expressed in the UML model as a collaboration diagram. A UML collaboration diagram is a diagram of object instances. Each instance in this diagram represents a separate and distinct object. Even though the same «screen» may appear multiple times in the same scenario, each and every time it appears it represents a new "object" instance, one with a potentially different state. In a web application the typical screen object's lifetime is short. It is created when first requested, and is terminated when the next screen is requested.

When creating a «screen» scenario diagram it is important to use actual instance names. For example in the scenario of figure 5 many of the screens have instance names in them (the part before the colon). These instance names should represent real life names, and help to bring to life the nature of the scenario. Numbered message indicators

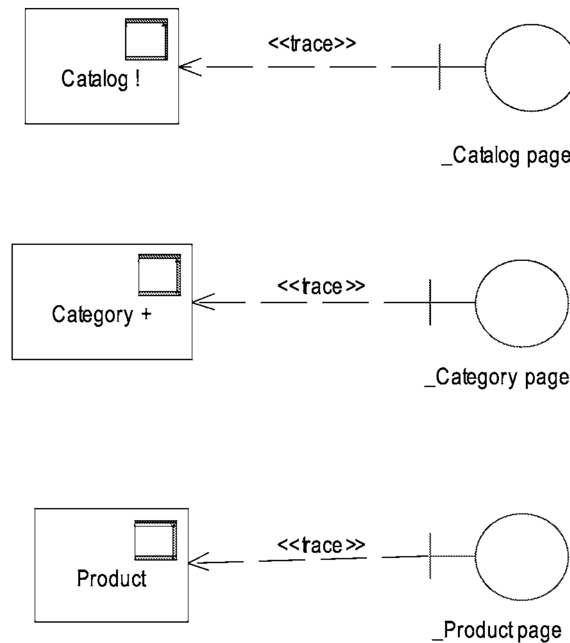

Figure 4. Sample «screen» and analysis element mappings.

indicate the order of flow in the scenario. In this particular diagram the objects were spatially organized from left to right to indicate specific tiers or classes of screens, and vertically from top to bottom in chronological order.

The third piece of information that the IA is responsible for, and of interest to the system's designers is the navigational path map. For most systems this diagram might actually require multiple diagrams to express clearly, however small systems (number of pages less than 30) can express them in a single diagram easily. This diagram is akin to a class diagram found in the analysis and design models. It expresses the «screen» classes and their principal navigational paths to the other screens in the system. Its primary purpose is to show the types of screens and their relationships with each other. Figure 6 shows a simple navigational map diagram for an eRetail application.

Color is used in this diagram to help express some inherent properties of some of the pages (navigable from anywhere and scrollable). This diagram also shows only the normal navigational paths. It does not attempt to address the use of the back and forward browser buttons, nor of the use of locally cached pages. It should be possible to print off copies of this diagram and with a highlighter draw paths through the system that correspond to required use case scenarios.

More detailed diagrams expressing navigational flow can be created that include screen attributes (figure 7). In these diagrams a combination of navigational flow, and dynamic content are expressed. Diagrams like these provide a contract of sorts between the UX team and the engineering team. In this diagram the screen names, and their dynamic content are clearly identified. The UX team can use this information to evolve

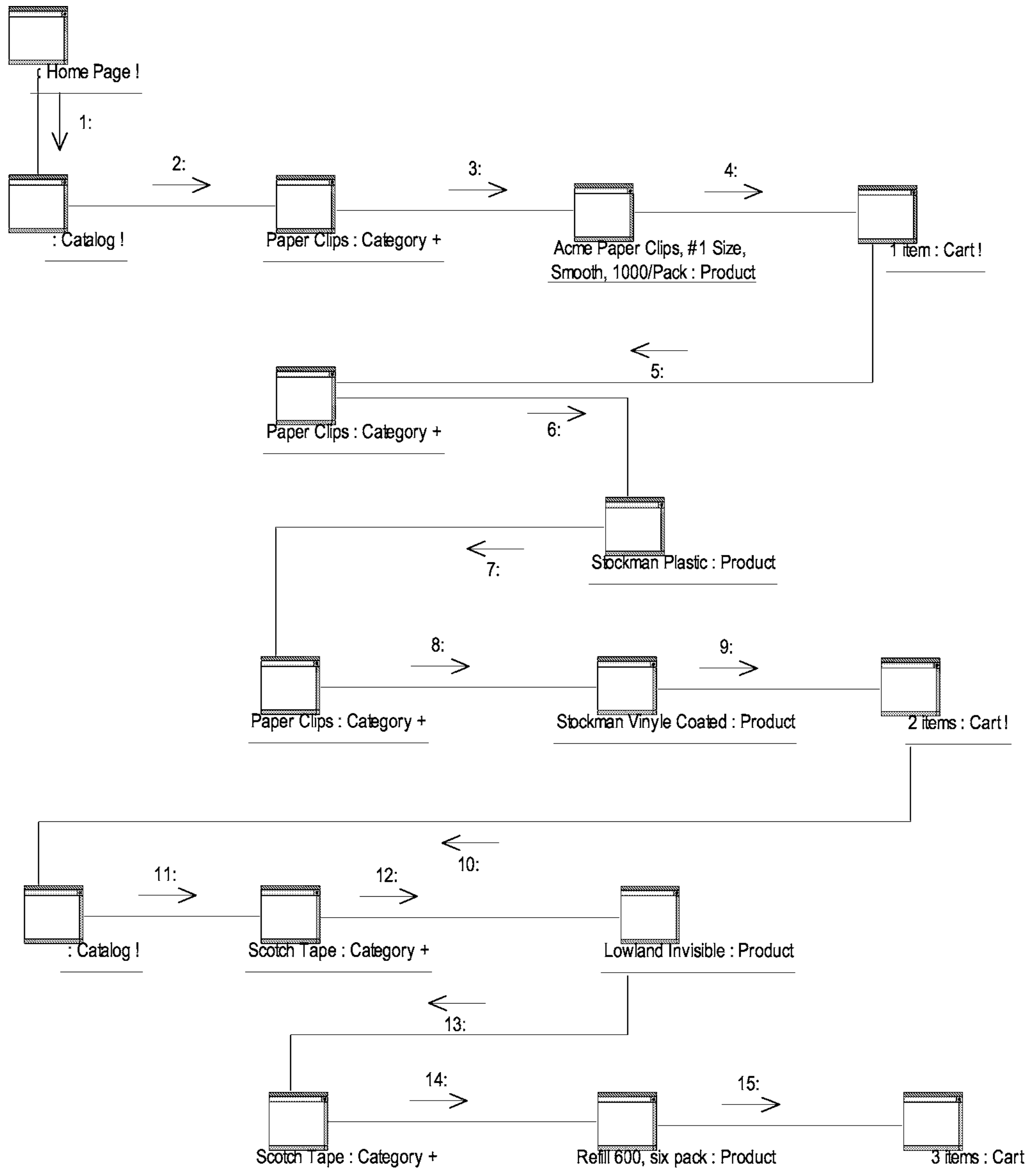

Figure 5. Storyboard scenario expressed as a UML collaboration diagram.

the screen designs and prototypes by stubbing the actual values of the dynamic content with dummy values. The engineering team can use this diagram to stub out the HTML templates so that each page just displays a list of the dynamic content expected in the page, and any associated forms. This temporary HTML is used during the debugging of program logic, freeing the developer from the overhead and potential confusion of presentation details. If each team agrees to uphold the spirit of this diagram and contract,

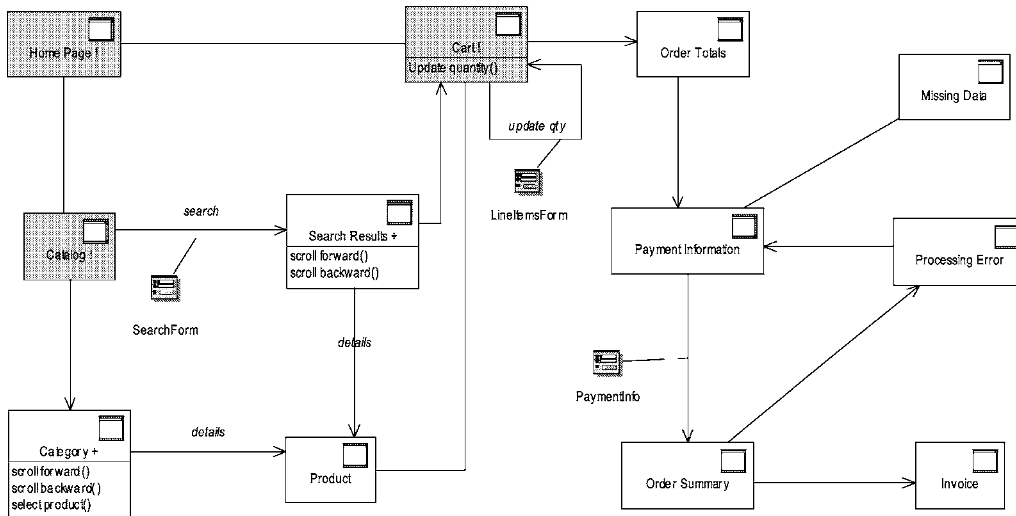

Figure 6. Web application navigation map using stereotyped classes.

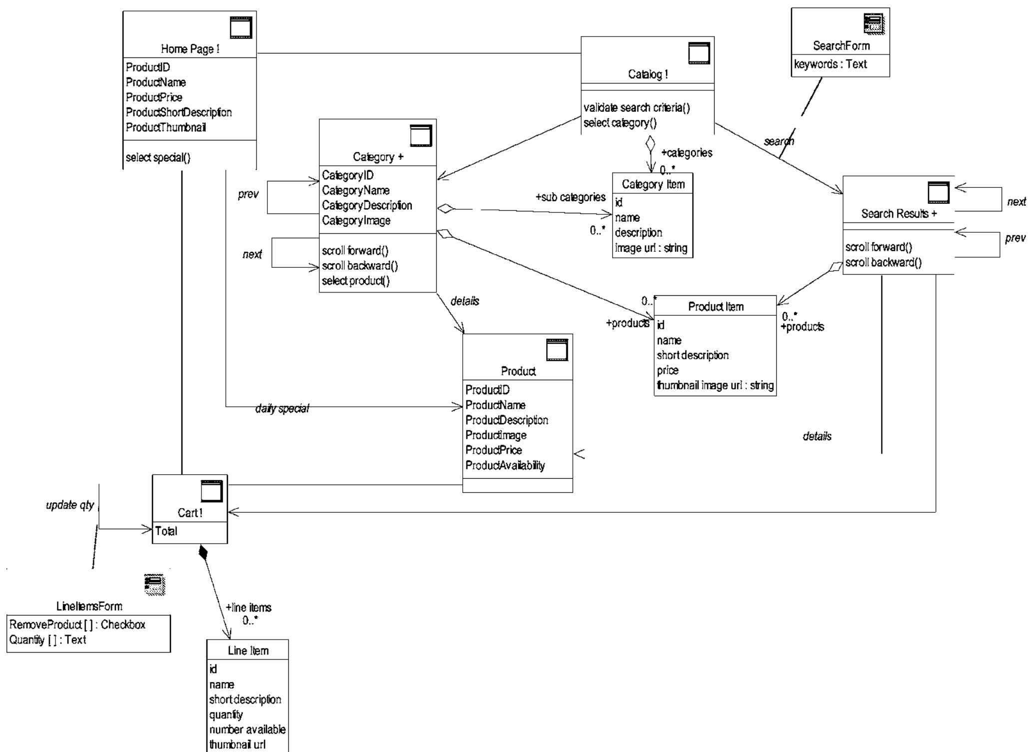

Figure 7. Navigational map with screen attributes.

the integration events, which should be happening at periodic and frequent intervals, should be much easier.

3.2. Design-level representation for web-based systems

Modeling web-based systems is no different than modeling any system. Success is defined by its understandability and correctness with the respect to the things it is modeling. The key to modeling web centric architectures is therefore to do so at the appropriate level of abstraction and detail. The Web Application Extension (WAE) is a formal extension to the UML that defines a set of UML extension elements[3] that allow designers to model web based systems [Conallen 2000]. In addition to the WAE other extensions have been proposed to express design model details of web-based systems [Baumeister 1999] and for more general user interfaces [Anderson 2000]. The former example tends to focus on the client side elements and collaborations while the latter is useful for more generalized interfaces and does not provide as much detail design information. The WAE on the other hand tends to focus on the page creation aspects of the web tier and their connections with the middle tiers of the system. This paper overviews the most fundamental elements of the WAE, its motivations and mappings to actual implementations.

One of the things that make the web application unique is the nature of client and server tier communications. When client and server tier elements collaborate to accomplish some business task, a dialog of communication is exchanged between the client and server. In traditional client–server systems this is usually done over a semi-persistent communication channel. The messages that are sent back and forth vary widely, and are typically very context and order sensitive. On the contrary most communication in web-based systems are fragmented and over temporary communication channels. The nature of communications is not message oriented, but rather resource oriented [Berners 1999].

When a web client communicates with a web server it is always in the form of a request for a web page resource. The request often includes parameters, or submitted fields of data supplied by the user. The server responds to a resource request by accepting the supplied parameters and user input, processing it, and returning with a new instance of a resource page. Each page is an instance of a HTML formatted document.[4] HTML provides a means to express a user interface that can be rendered and presented to a user on the client. It also provides a mechanism for a user agent[5] to accept simple input, execute scripts, applets, controls and other client side resources.[6]

Deciding what is important to model in web-based systems, and at what levels of abstraction depends on how designers think of their systems, and what it is that im-

[3] Profile is the current term for a set of UML extension elements.

[4] For web applications HTML is the predominant language used for resources, however similar systems may utilize Wireless Markup Language (WML) or XML depending upon the application and user agent.

[5] The technical term for a web browser.

[6] Depending upon the client configuration, user agents might invoke special applications on the client to render or process special types of information sent to the client. For example, many browser installations are configured to run external applications to render PDF files or play multimedia clips.

plementers actually build. For most web-based systems the principal concepts in the presentation tier are web pages, and the hyperlinks that connect them. These two key concepts in web centric architectures are often what distinguish a web design from any other client–server design.

At higher levels of abstraction (requirements and analysis), the concept of a web page is a singular one (i.e., UX Screen). The web page is thought of as a single element that can be requested and rendered in a browser. While in the browser it might also execute some behavior (via scripts or embedded applets). Designers, however, have a more detailed view of a web page resource.

The construction and life cycle of a typical web page is complex. When a user selects a hyperlink or submits a web form a request for a resource is made to the server. The server accepts the request and identifies the resource. If the resource requires server side processing, as, for example, a web page template might, then the template is loaded and its processing instructions are executed.

Web page templates are things like Java Server Pages (JSP), Active Server Pages (ASP), Cold Fusion Markup (CFM), PHP, etc. They are a combination of presentation and rendering instructions and business logic coordination. They are the glue that connects the business logic tier to the presentation tier of the system. Templates dynamically build HTML content. They invoke operations on business tier objects and use the results to build and stream out unique HTML to each client.

Once on the client, the page takes on a new life. The HTML rendering instructions sometimes contain scripts or reference applets that execute on the client. These scripts often contribute to the overall business logic processing of the system by performing form validations, or by providing navigational assistance to the user. If they do indeed contribute to the business goal of the system, then they need to be part of the system's design and hence the design models of the system.

The designer is faced with problem of how to model a web page, and its hyperlinks in UML. During analysis web pages are conveniently modeled with a single class or classifier element. A design model, however, contains much more detail and is a closer abstraction to the real mechanisms and components that define a web page in the system (JSP, ASP, PHP, ...).

The solution to modeling web page designs is to represent a web page resource with two class level abstractions, one to represent its server side life, and another to represent its client side life. The two, however, are closely tied with a special relationship.

Using the UML's extension mechanism, stereotypes are defined that can be placed on common modeling elements and that denote new semantics to be applied to the element. Two class stereotypes and one association stereotype describe this special relationship between the design modeling elements that make up a web page.

The class stereotype «server page» is applied to a class in the model that represents that part of the web page that exists on the server. Its attributes and operations exist while the server is processing the page. This class has relationships to server side objects. When a «server page» object is created it does two things:

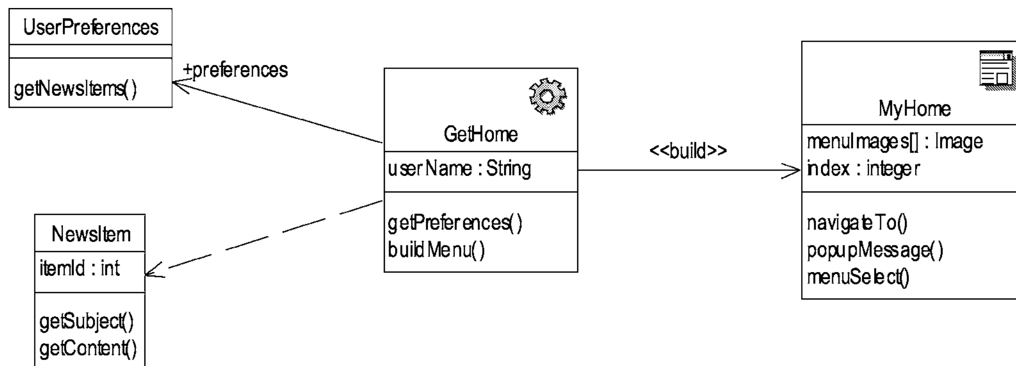

Figure 8. Simple design model of a web page.

(1) process any incoming parameters or form data supplied by the user;

(2) produce HTML output, or redirect itself to another «server page» resource that does.

The class stereotype «client page» represents the web page as it appears on the client. Its attributes are typically JavaScript variables, and its operations are JavaScript functions. It has relationships to embedded applets, and other client side resources. The relationship between a «server page» and the «client page» that it is responsible for building is a «build» stereotyped directional relationship.

In figure 8 the «server page»[7] `GetHome` class models the server side aspect of a Java Server Page that delivers a user's personalized home page. In this example the class `GetHome` defines the String attribute `username`, and three operations; `getPreferences()`, `buildMenu()` and `news()`. It also has an association with the Java class `UserPreferences`, and an identified dependency on the Java class `NewsItem`.

In the JSP code these appear in special declaration blocks,[8]

```
<%!  String username; %>
<%!  UserPreferences preferences; %>
<%!  public UserPreferences GetPreferences() {
        HttpSession session = request.getSession();
        return (UserPreferences) session.get(``UserPref'');
      }
      public void buildMenu() { ... }
      public void news() { ... }
%>
```

Operations defined in the JSP source can create and invoke methods on server side Java objects. Through these data from the main application server can be obtained and used in the construction of the HTML output. Further down in the JSP source there is likely to be combinations of HTML statements and JSP statements. For example, in the

[7] The UML extension mechanism allows stereotyped elements to be rendered with special icons. The full set of icons and details of this extension are captured in the WAE.

[8] The full source of the actual operations have been omitted for simplicity.

following fragment we see how the HTML template contains embedded Java scriptlets that execute on the server. Each scriptlet is bound by `<%` and `%>` tokens in the JSP source.

```
<h2>Welcome <%=username%></h2>
<p>Your personalized news items:</p>
<%
  NewsItem items[] = preferences.getNewsItems();
  for( int i=0; i<items.length; i++ ) {
    NewsItem item = items[i];
    String subject = item.getSubject();
    String content = item.getContent(); %>
    <h3><%=subject%></h3>
    <p><%=content%></p>
<% } %>
```

The end effect is a combination of HTML code and server side Java code that when processed by a JSP container produces HTML formatted output with appropriate dynamic content. The HTML part is modeled with the «client page» stereotyped class `MyHome`. In addition to having dynamic content embedded in the HTML, this class also indicates the use of client side JavaScript. Two variables and three functions are defined. They map directly to an HTML `<script>` element:[9]

```
<html>
<script language=''JavaScript''><!----
var menuImages; // array of images for the menu
var index;
function navigateTo( url ) {
  window.location = url;
}
function popUpMessage( msg ) { ... }
function menuSelect( item ) { ... }
// --></script>
<head>
  <title>Your Home</title>
</head>
<body>
<h2>Welcome jimc</h2>
<p>Your personalized news items:</p>
<h3>Elvis Spotted in Conshocken</h3>
<p>April 1, 2001 Elvis was spotted again in
Cunninghams Pub in West Conshohocken... </p>
</body>
</html>
```

Each «server page» and «client page» are ultimately realized by a components. It is the component that maps directly to a URL. In the example above both `GetHome` and `MyHome` are realized by the same JSP component. This JSP component not only provides the mechanism by which the logical classes can resolve to URLs, but it also

[9] The full HTML source is not shown for simplicity.

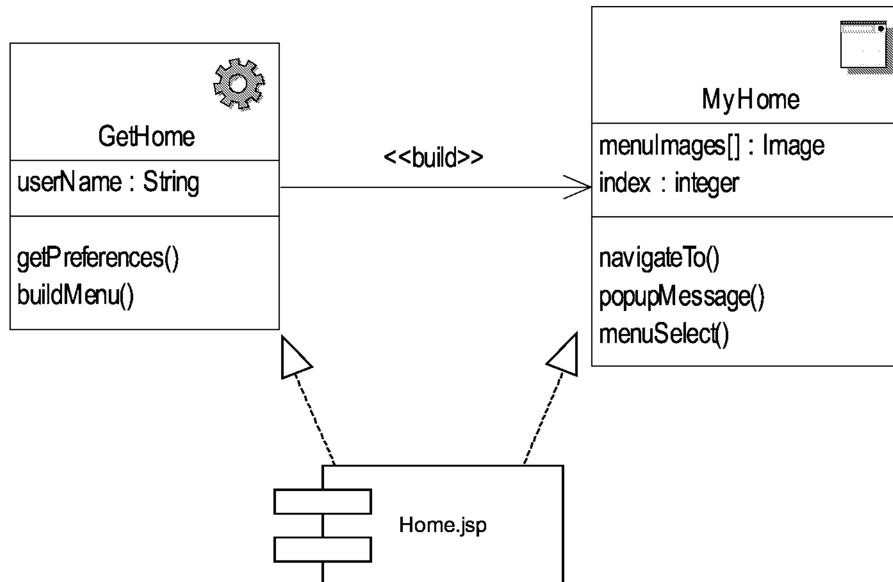

Figure 9. JSP components realize logical client and server pages.

identifies the JSP source code file that provides the implementation of the classes. The component is how the two logical abstractions of the web page are actually managed as one web page, and one source code component (figure 9).

Additional conventions and stereotypes are used to further refine how URL paths are captured and expressed in the model. Package hierarchies roughly map to URL paths, with the top most package stereotyped «virtual root» mapping to the server part of the URL.

Another important part of the design model is the ability to express hyperlinks to other pages in the system. The most natural way to express this is with a «link» stereotyped association. A «link» association can connect two «client pages» or a «client page» to a «server page». Links can only originate from a «client page» since they are essentially abstractions of the HTML anchor element (<a>). When a URL passes along parameters, these are captured as tagged values,[10] or optionally with a link class.[11]

The last major element type that needs to be modeled is the HTML form. This element defines a combination of user input field types and a URL to submit their values to for processing. In the design model the class stereotype «form» is used to capture this collection of elements. The principal reason forms are modeled with separate classes is because according to the HTML specification, it is possible for any given HTML formatted page to contain multiple forms, each submitting themselves to a different server

[10] A UML extension element that allows additional values to be associated with model elements. In class diagrams tagged values are rendered between curly braces.

[11] The attributes of a link class can be used to define more exactly the set of expected URL parameters are passed along with the request for the resource.

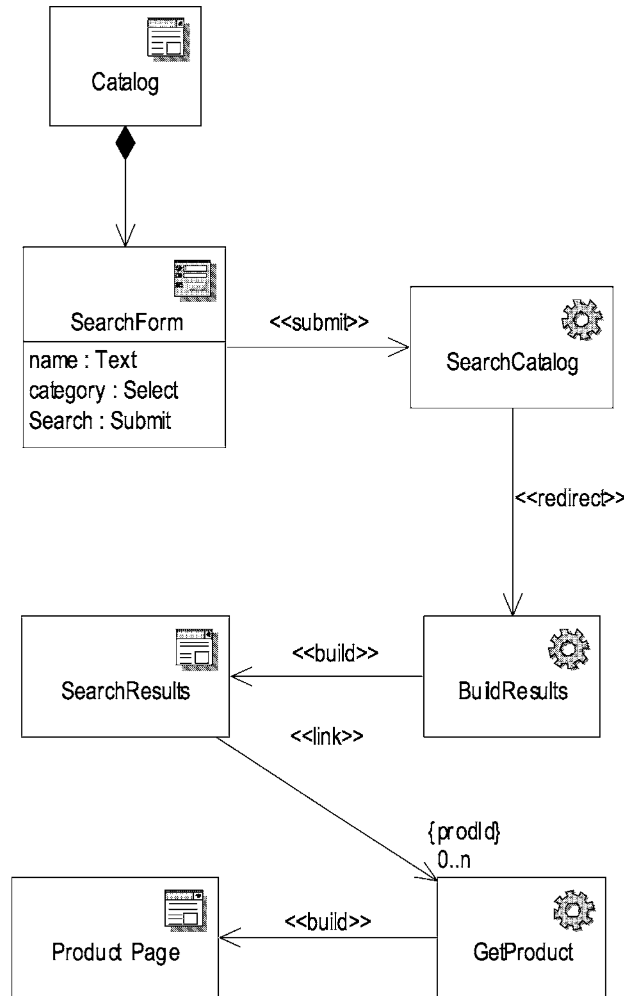

Figure 10. Logical model of a searchable product catalog.

side URL. Forms are always contained by client pages, since an HTML form does not operate outside the context of a an HTML element.

In the logical design model a stereotyped «submit» association is drawn from the form to a «server page» stereotyped class. Figure 10 shows a class diagram with several stereotyped classes that make up part of a searchable product catalog.

In this diagram a `Catalog` «client page» contains a search form. This form contains three input fields, one text box, one selection list and a submit button. The when the user presses the submit button, the values are passed along with the resource request to the `SearchCatalog` «server page». In this particular model, the `SearchCatalog` page just searches the catalog and when finished passes control over to the `Build-Results` «server page». The `BuildResults` page is responsible for building the `SearchResults` «client page». In the JSP Model 2 architecture, the `SearchCat-`

`alog` class is implemented as a Servlet, and `BuildResults` as a JSP. The determination of how any given «server page» or «client page» is implemented is made by the type of component that realizes it.

This discussion of modeling web-based system designs with UML is just a broad overview, and addresses only the highlights of the technique. It's guided by the need to express detailed designs of web applications at the appropriate level of abstraction and detail. This detail is best expressed by extending the UML with the Web Application Extension (WAE), where the logical structure of the web pages is clearly captured in their client and server side life cycles. Although logically any given web page may be modeled with many classes, in the component view it is modeled as a single component. It is the component, which realizes the logical classes that «client page», and «server page» classes resolve to.

4. Reusing web solutions

The ability to model and communicate at various levels of abstraction and detail (use case, UX, analysis, design, implementation, etc.) with UML is only the first step in enabling teams to efficiently build web applications. UML is a language for expressing the structure and behavior of systems and parts of systems. It is apparent to the experienced practitioner that the vast majority of software systems share common mechanisms, frameworks, components and patterns. Like the art of software development itself software architectures and systems themselves are evolutionary not revolutionary. Each new generation of systems is built upon the success (or failure) of preceding generations.

Reuse in the software industry has a speckled past. The earliest successful efforts in organized reuse came about with the use of compiled object libraries. Although successful to an extent (and still used today) this form of reuse has its drawbacks in platform transparency and often imposes significant design constraints on its clients. Early object-oriented advocates lauded the promises of "classes" of objects to be reused and specialized. In practice it has been found that the natural unit of reuse is really at the collaboration or package level, where groups of tightly coupled classes collectively represent a unit of reuse. The first real success story for practical and wide spread reuse in the industry came with the introduction of the Visual Basic Control (VBX). For the first time thousands of third party components were being produced that could be effortlessly incorporated into Visual Basic projects.

What made the VBX control even more successful than previous attempts at reuse was an accepted standard. This standard allowed Visual Basic developers to produce high quality components that had a good chance of integrating with existing projects, without having the component developers deal with the issues of producing many versions of the component for different architectures. This success clearly helped to promote today's most prominent component models (COM and JavaBeans).

Both COM and JavaBeans like the UML are accepted standards. They describe actual component models (the details of object-to-object communication), and design notations. What is missing is the standard that describes how these can be organized

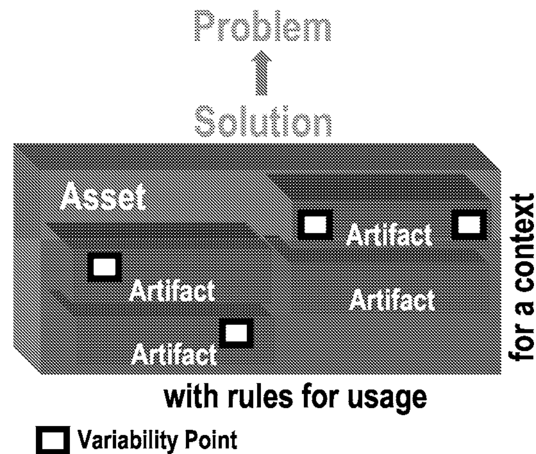

Figure 11. Assets are solutions to problems.

into a practical reusable unit. This reusable unit contains all the designs, code and other artifacts that implement a pattern. A pattern considered here is a solution to a problem in a context. The problem is most often a recurring problem that is evident across similar (or even dissimilar) systems. Solutions with supporting artifacts (models, code, etc.) that are applied to multiple projects are considered reusable software assets. What is described in remainder of this article is just such a standard for the packing and presentation of reusable software assets for web-based systems (or any kinds of systems).

Any practical software reuse solution is reified as a collection of artifacts that can be included directly in the target project's artifacts or with minimal effort. An asset is a package of relevant artifacts that provides a solution to a recurring problem, for a given context. An asset has one or more artifacts and these are classified and have descriptions of how to apply and use them. Figure 11 illustrates the major concepts of assets.

An asset is created or harvested with an explicit purpose of applying it (repeatedly) in subsequent development efforts. Assets can be of different granularity and may allow different degrees of customization (or variability) and can be applied (or targeted) at different phases of software development. Variability Points, as defined in the RAS, are locations within an asset that may be customized or where concrete elements must be provided when the asset is applied or reused.

Raising the abstractions with which software engineers engage problems, systems, and solutions, has an effect that is similar to the way an automobile user may approach a car. Rather than focusing on the timing belt and the mixture of air and fuel in the engine, the automobile user is faced with a higher level pattern including participants such as the steering column, mirrors, the accelerator and so forth. Focusing on this level of abstractions increases the effectiveness of the automobile user in achieving the target destination.

Assets therefore are a named collection of relevant artifacts that provide a solution to a problem for a given context and a description of how to use and apply the artifacts. And, the reusable artifacts in an asset, such as those describing a pattern, may include

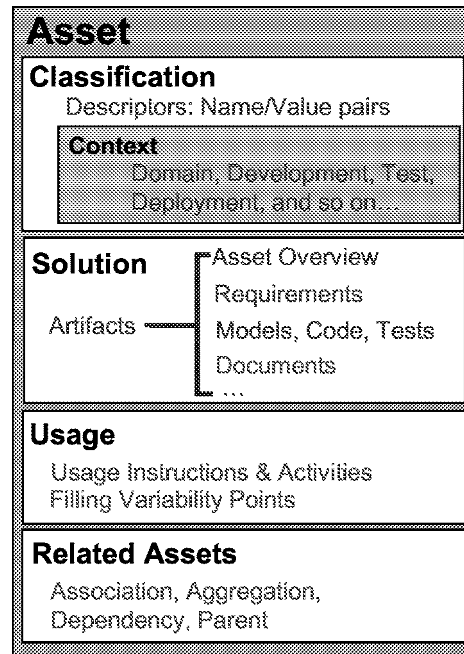

Figure 12. RAS-based asset structure.

not only the source code, but also may include the requirements, the models and designs, as well as the relevant testing artifacts.

The Reusable Asset Specification (RAS) is being created to standardize the approach to organizing and packaging assets for reuse. This specification describes the asset's structure and the approach for reducing the friction in reuse transactions. This enables the consumer of the asset to rely on a certain paradigm each time they reuse an asset. Much like the automobile user who can rely on a certain set of participants, such as the steering column, and so forth. The RAS is just a specification. It describes how to bundle a varied collection of artifacts in a consistent package.

Briefly, the RAS describes the structure of an asset by saying that it must have a *usage* section, a *classification* section, a *solution* section, and a *related-assets* section. The usage section contains the documents describing how to apply the asset for a given context. The classification section provides descriptors of the asset and describes the contexts that are relevant to the asset. The solution section contains the artifacts that comprise the solution of the asset. The related-assets section contains references or pointers to other assets. There are several kinds of asset relationships defined in RAS. Figure 12 highlights the top-level RAS sections.

At the technical level the asset package is just a collection of files in a hierarchal structure, with a single XML based descriptor file that acts as a manifest. The structure illustrated in figure 12 is the structure of the XML descriptor file. The package is

typically “zipped” together like Java JAR files. The XML descriptor file is located in a fixed position in the zipped RAS file, and contains relative links to each of the individual artifact files.

Assets that are based on the RAS can participate within reuse scenarios such as harvest asset, package asset, publish asset, measure asset, rate asset, search asset, browse asset, and install/customize asset. See `www.rational.com/rda` for more information on the RAS.

The final details of the official specification are still being discussed, however the general approach has been agreed upon. Some tool vendors have embedded RAS functionality into their offerings.

An example might best illustrate an asset for web-based systems using the asset packaging as described in the RAS. Bringing together the representation of web-based systems described earlier in this article with techniques for packaging these solutions for reuse can positively impact development timelines and efficiencies.

An important point to remember is that an asset may include artifacts from all aspects of the software development lifecycle. This includes, requirements, models, code, tests, and so on. The UML plays a critical role in describing individual artifacts in the asset making the asset consumable by those investigating its solution without having to invest the effort in apply it first. In the example below, the focus is to illustrate the kinds of UML models, artifacts, documentation and so on that are relevant to reusable solutions for web-based systems. The asset is not completely documented or described in this article due to space constraints.

5. Asset: paged dynamic list

5.1. Classification section

Context

- *Development.* This asset can be used in any web-development environment that provides user session management capabilities.
- *Test.* This asset has two test configurations; Linux, and Windows 2000.
- *Deployment.* This asset should be deployed on Tomcat and Apache servers.

Descriptors

- *Keywords.* Web search, search result, dynamic list, page dynamic list.
- *Known Uses.* This asset has been used in n-tier web-based systems.
- *Author.* Jim Conallen.
- *Packager.* Grant Larsen.

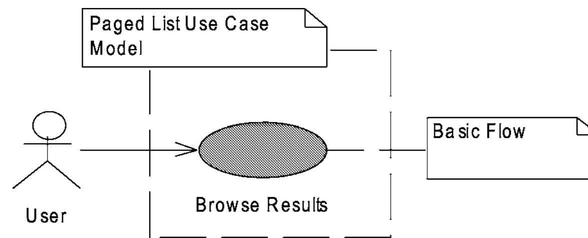

Figure 13. Browse results Use Case diagram.

5.2. Solution section

In this section the artifacts are organized into four sets: the requirements set, the design set, the realization set, and the test set.

Overview. This solution provides a general mechanism for capturing queries and building server result lists and screens for the user to navigate.

Problem description. When navigating the web, users perform searches and must navigate the results of those searches. The total number of results is can be more than can be practically placed in a single web page, therefore the results must be viewed one section at a time.

Requirements Set

Requirements Set Artifact List. Below is a list of the artifacts comprising the requirements set.

pageddynamiclist.mdl [*Use Case Model* package] ⇒ Rational Rose model,
browseresults.uc ⇒ Requisite Pro Use Case document.

The Requirements Set includes the artifacts that comprise the requirements that the solution will support. Below is a small list of requirements.

1. The user must be able to enter searches for items on the web.
2. The search criteria must be managed.
3. The results of the searches must be configurable to a specified number per page.
4. The user must be able to browse the results of the searched; including forward, previous, and specific page browsing.

Figure 13 illustrates the Use Case diagram for this asset. The figure states that the User (the Actor) will browse the results of a search.

The basic flow of the Use Case is illustrated below. Figure 14 outlines the expected behavior that the asset will support. From this figure you can see that the user can issue searches and then will have several options for reviewing the results including navigating to the next page, to the previous page, and to a specific page.

Figures 13 and 14 represent the models describing the requirements set for which the asset should resolve.

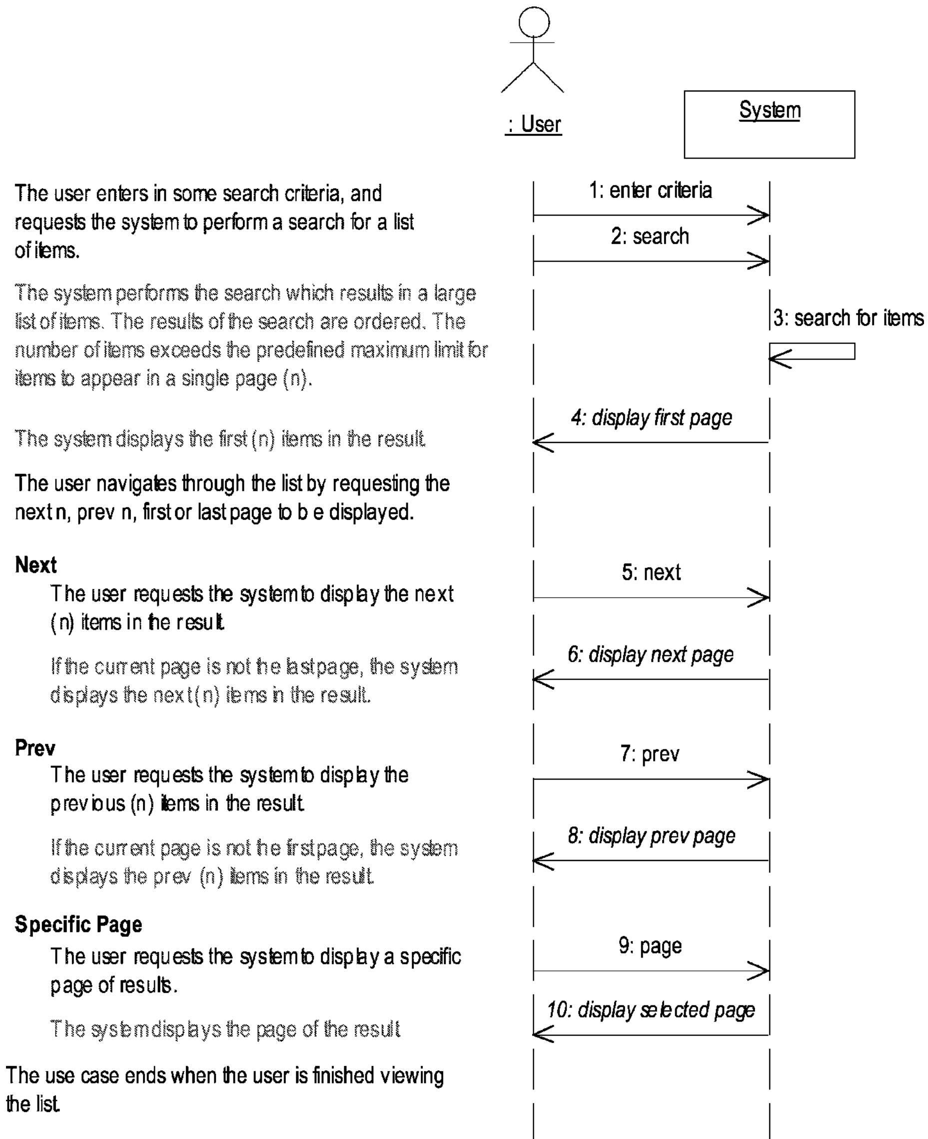

Figure 14. Browse results: basic flow sequence diagram.

Design set

Design Set Artifact List. Below is a list of the artifacts comprising the design set:

- pageddynamiclist.mdl [*Design Model* package] ⇒ Rational Rose model.

The Design Set includes the models and other artifacts that comprise the design of the solution.

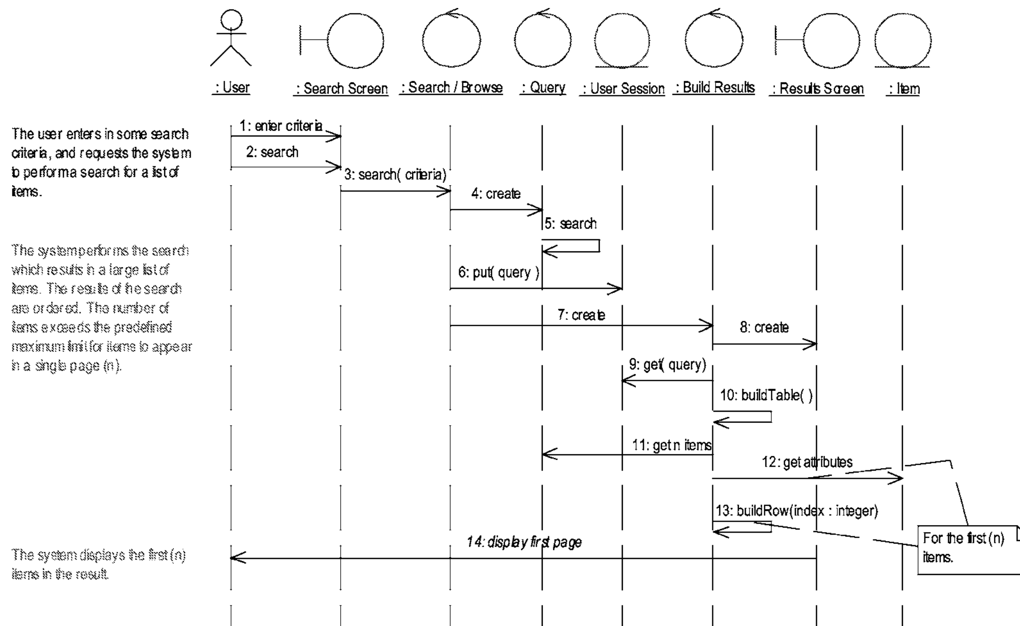

Figure 15. Analysis Model sequence diagram.

The Analysis Model includes the models and other artifacts that comprise the analysis views of the solution. Figure 15 describes *what* the asset does using a UML sequence diagram with «boundary», «control», and «entity» stereotypes [Jacobson 1992]. The model describes the interaction of the Search/Browse controller with the Query and User Session to create the Build Results. These results are then placed on a Results Screen and then displayed to the User.

Figure 16 represents the class diagram from the Design Model. It illustrates the classes, attributes, methods, and relationships necessary to provide the paged dynamic list solution.

The `SearchPage`, `SearchForm`, and `ResultsPage` classes are client side pages that the user will see. The `Search` and `BuildResults` classes are server side pages.

Variability Points, as defined in the RAS, are locations within an asset that may be customized or where concrete elements must be provided when the asset is applied or reused. The `Search` class and the `BuildResults` class have the following variability points.

- `Search` *class Variability Points.*

  Search scope describes the scope for the search, such as database, intranet, Internet.

- `BuildResults` *class Variability Points.*

  Result set size describes the maximum size of the result set.

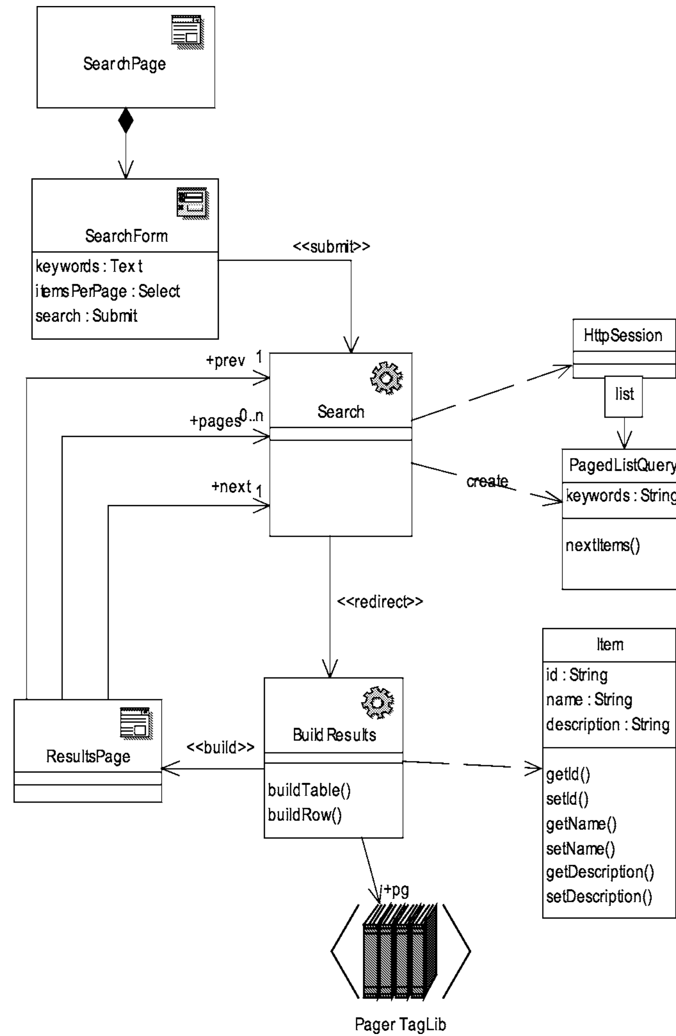

Figure 16. Design Model class diagram.

Realization Set

Realization Set Artifact List. Below is a list of the artifacts in the asset:

- search.jsp ⇒ JSP file;
- buildresults.jsp ⇒ JSP file;
- pagedlistquery.java ⇒ Java file;
- item.java ⇒ Java file.

The Realization Set contains artifacts that are the models and implementation files of the solutions. This set typically contains UML Component diagrams as well as the

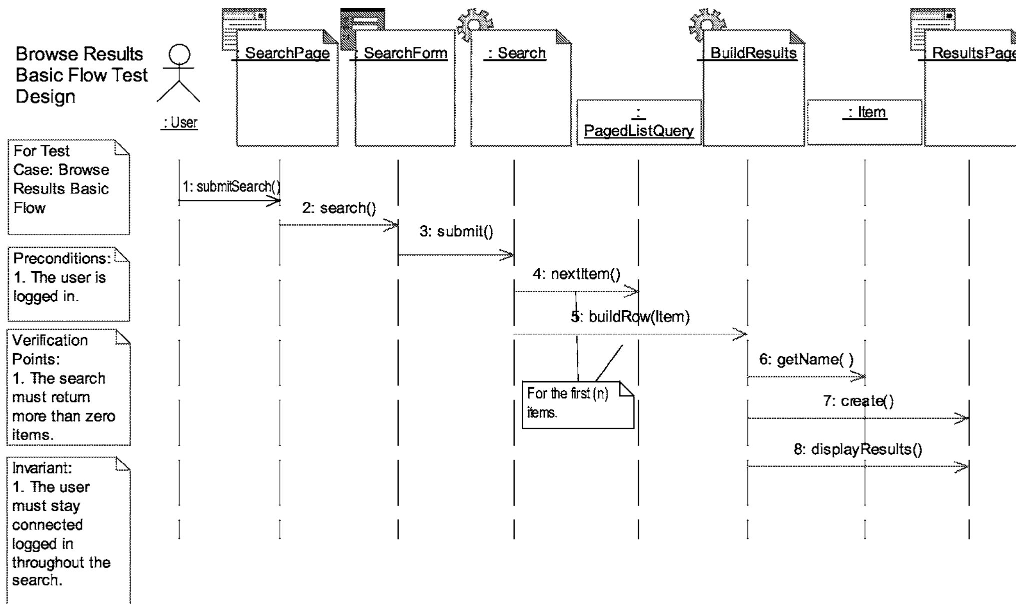

Figure 17. Browse results basic flow test design sequence chart.

actual Java, JSP, C++, C#, and other relevant files. There are no models illustrated for this set in this example.

Test Set

Test Set Artifact List. Below is a list of the artifacts in the test set:

- pageddynamiclist.mdl [*Test Model* package] ⇒ Rational Rose model;
- pageddynamiclist.rsp ⇒ Test Manager project.

The Test Set contains the models and tests that validate and verify the solution. Tests may be located not only in this set but also may be located close to the actual artifacts themselves.

Figure 17 illustrates the nature of a test sequence chart for a test design, which may include a description of the verification points as well as the pre-/post-conditions and data pool(s) and other relevant test design information.

In addition to these models are a series of tests, test cases, and so on handled by the Rational Test Manager.

5.3. Usage section

This section illustrates how to apply the asset by describing the activities that should be performed.

Activities

(i) *Verify Tomcat & Apache servers are operational.* Make sure these servers are installed.

(ii) *Inject the Requisite Pro document into the project workspace*.

(iii) *Inject the Rational Rose model into the project workspace*. Assign concrete values to the Variability Points on the `Search` class and the `BuildResults` class.

(iv) *Inject the Java and JSP files into the project workspace*.

(v) *Regenerate the source code from the Rose model to update with the concrete values added to the* `Search` *class and the* `BuildResults` *class*.

(vi) *Inject the Test Manager project into the project workspace*. Update the test cases and test designs for the `Search` and `BuildResults` tests with the concrete values chosen for the `Search` class and `BuildResults` class.

(vii) *Recompile test scripts and harnesses*.

(viii) *Compile the JSP and Java classes into the project workspace*.

5.4. Related assets section

This section identifies those assets on which this asset may be dependent or may contain or may point to previous versions of this asset.

Related asset name	Reference	Relationship
Pager Tag Library	`http://jsptags.com/tags/navigation/pager/`	Association
Sun J2EE Pattern Value List Handler	`http://java.sun.com/blueprints/patterns/j2ee_patterns/index.html`	Association

6. Summary

Web-based systems can be effectively designed and implemented using the UML. WAE is the UML extension used to represent the items that are considered first-class citizens in web-based systems. As the challenges facing software engineers continue to increase, modeling, packaging, and applying assets for web-based systems will increase the effectiveness of software engineers.

The RAS describes some techniques for packaging reusable assets and reducing the friction in reuse transactions. We find that packaging reusable assets may require additional models, artifacts, or documentation that will not typically be included in normal system development. This extra effort is justified by the value these items bring in

making the reusable asset understandable and useful. This article illustrated using the RAS to partially package an asset for solving problems relative to developing web-based systems.

References

Alhir, S.S. (1999), "Extending the Unified Modeling Language (UML)," available at http://home.earthlink.net/~salhir/extendingtheuml.html.

Anderson, D.J. (2000), "Extending UML for UI," A Position Paper for the TUPIS2000 Workshop at UML2000, http://www.uidesign.net/2000/papers/TUPISproposal.html.

Baumeister, H., N. Koch and L. Mandel (1999), "Toward a UML Extension for Hypermedia Design," available at http://www.pst.informatik.uniuenchen.de/projekte/forsoft/pubs/uml99.pdf.

Berners-Lee, T. (1999), "Web Architecture from 50,000 Feet," available at http://www.w3.org/DesignIssues/Architecture.html.

Booch, G., I. Jacobson, and J. Rumbaugh (1999), *The Unified Modeling Language User Guide*, Addison-Wesley, Reading, MA.

Conallen, J. (2000), *Building Web Applications with UML*, Addison-Wesley, Reading, MA.

Earls, A. (1999), "True Test of the Web," available at http://www.informationweek.com/718/18iutst.htm.

Jacobson, I. (1992), *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, Reading, MA.

Jacobson, I., G. Booch, and J. Rumbaugh (1999), *The Unified Software Development Process*, Addison-Wesley, Reading, MA.

Kruchten, P. (2000), *The Rational Unified Process, an Introduction*, Second Edition, Addison-Wesley, Reading, MA.

Rational Software (2001), "Rational Development Accelerators", available at http://www.rational.com/rda.

Rumbaugh, J., G. Booch and I. Jacobson (1999), *The Unified Modeling Language Reference Manual*, Addison-Wesley, Reading, MA.