Architectural Issues and Solutions in the Development of Data-Intensive Web Applications

S. Ceri, P.Fraternali

Dipartimento di Elettronica,
Politecnico di Milano, P.za L. Da Vinci 32,
20123 Milano Italy

R. Acerbis, A. Bongio, S. Butti,
F. Ciapessoni, C. Conserva, R. Elli,
C. Greppi, M. Tagliasacchi, G. Toffetti
WebRatio
P.le Gerbetto 6,
22100 Como, Italy

Abstract

A data–intensive Web application is a Web-enabled software system for the publication and management of large data collections, typically stored in one or more database management systems. Data-intensive Web applications constitute the most diffused class of applications found on the Web today, and their industrial relevance is an established fact. As a consequence, an intense technical and scientific debate is ongoing on the various aspects of their implementation, especially on the software architectures and design process best suited to cope with the peculiar aspects of a data-intensive Web application.

The ideal software process should meet two possibly competing goals: 1) capturing the application requirements in a formal way, so to incorporate them in the development lifecycle and derive the software directly from the functional requirements; 2) delivering a software architecture that meets the non-functional requirements of performance, security, scalability, availability, and maintainability. Such process should also be amenable to automation, to let developers concentrate on functional requirements and optimization, and delegate the repetitive tasks to software tools.

The goal of this paper is to report on an experience of applying a novel software development process to data-intensive Web applications, and to discuss the problems, design choices, and trade-offs that led to the conception of WebRatio, an innovative technology for Web application development.

1 Introduction

Data-intensive Web applications are the predominant kind of applications found on the Web, and therefore their effective specification, design, implementation, and maintenance is crucial; several architectures, design tools, and implementation practices are being developed to better serve their needs. In particular, this paper focuses on WebRatio, a software tool representative of a particular approach to the development of Web applications, called the "model-driven approach", which claims that more and more effort should be spent on the application modeling, and re-usable implementations should be automatically or semi-automatically produced from high-level models.

The distinguishing feature of WebRatio [WebRatio02] is the adoption of formal graphical languages for the specification of data intensive Web applications, and the semi-automatic generation of code from such specifications. Web applications are specified using the Entity-Relationship (ER) model for the data requirements, and the Web Modelling Language (WebML) [CF+02, WebML02] for the functional requirements.

The supported ER model is quite conventional, with a few limitations that make the ER schema easier to map onto a standard relational schema; this standard schema is then used by the WebRatio implementation as either the schema of a newly designed database supporting the Web application, or as a reference for mapping to pre-existing data sources.

WebML is a visual language for expressing the hypertextual front-end of a data-intensive Web application, i.e., the interfaces presented to the users for content browsing and management. WebML includes primitives for modelling such aspects as:

- The structuring of the application into different hypertexts (called site views) targeted to different user groups or access devices.
- The hierarchical organization of a site view into areas.
- The pages that constitute the actual application interface, the content units contained in each page, with their relationship to the elements of the data model (entities and relationship)
- The operations and services that can be activated from the application pages.

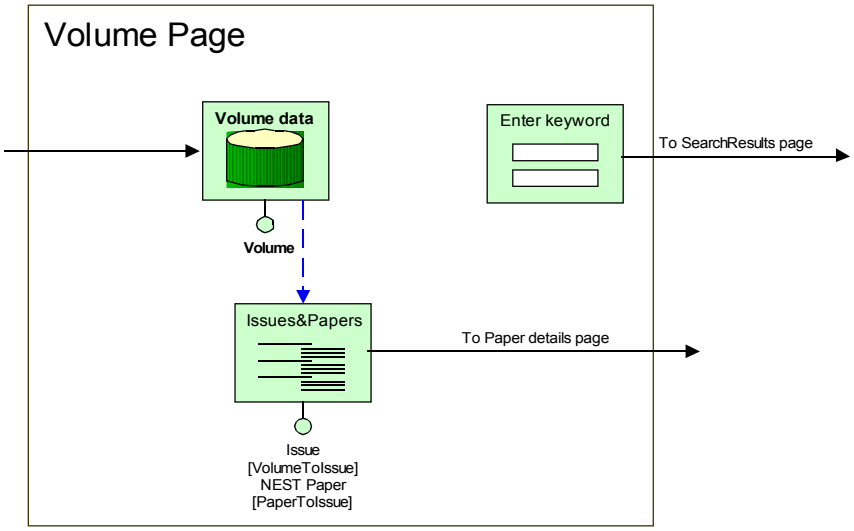

Figure 1 - Example of WebML page specification

- The links that connect pages, content units, and operations to provide users with suitable interactions on the browsers (e.g., anchors, radio buttons, forms for data entry).
- Session-level information and personalization aspects.

Figure 1 shows an example of WebML hypertext diagram, in which page are represented as white rectangles, units as labelled icons inside pages, and links as arrows between pages or units.

The model of Figure 1 represents a real page taken from the ACM Digital library Web site, which displays the details of an ACM TODS volume (Figure 2). The page includes a data unit (Volume Data) constructed on entity Volume, which displays the detail of a TODS volume selected in a previous page, as indicate by the link pointing to the unit, which implicitly transports the identifier of the volume.

The data unit is associated by a transport link (shown as a dashed arrow) to a hierarchical index unit, which displays the volume issues and papers.

The hierarchical index unit receives from the data unit the identifier of the volume, and constructs the hierarchy of issues and papers using the entities Issue and Paper, and the relationship VolumeToIssue and IssueToPaper. The outgoing link from the index unit points to a separate page (not shown in Figure 2), where the details of the paper are published. The page also contains an entry unit, whereby the user can insert search keywords, to be matched against the volume papers.

A WebML specification may also contain operations, which are services callable from within pages, which execute some processing and then display a result page. WebML offers several built-in operations, and a mechanism for adding user-defined content units and operations.

The WebRatio development architecture includes a graphic interface for editing ER and WebML schemas, and customisable code generators for transforming ER specifications into relational table definitions for any JDBC or ODBC compliant data source, and WebML specifications into page templates for the Java2EE and .NET architectures.

The design of the WebRatio development process and tools has faced a number of challenges, in part inherent to any CASE environment, in part specific of the Web application development context.

- The code generator should be based on a modular software architecture, where each aspect of the application logic is as isolated as possible. In particular, the presentation, business, and data extraction and manipulation logic should remain separated and be independently evolvable.
- The part of the presentation dealing with the graphical aspects of the pages (like the overall page layout, static texts and images, CSS styles, and so on) and client-side processing (like input validation) should be factored out of the code generation process, and editable by a non-technical graphic designer.
- The data extraction and manipulation queries should also be factored out of the code generation process and from the implementation code, so that the data expert should be able to override the system-generated queries, both in the design stage and after the application is deployed.

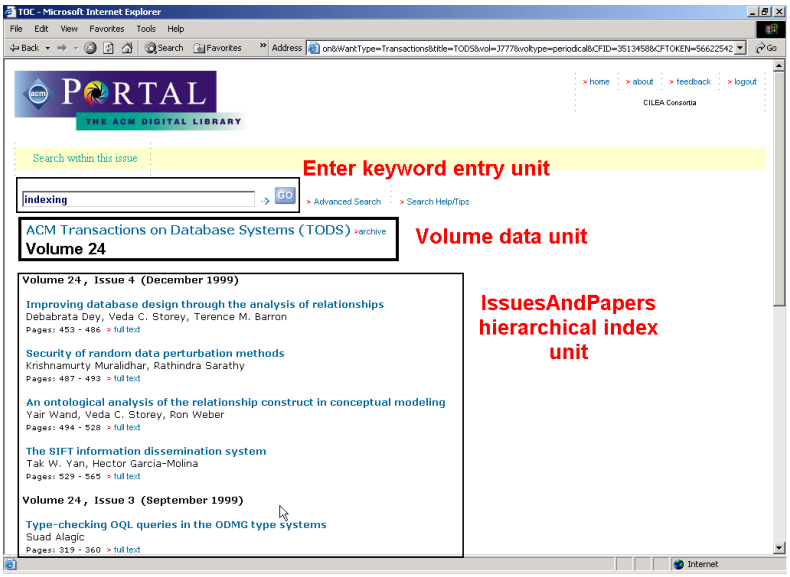

Figure 2 - A page from the ACM Digital Library modelled in Figure 1

- The design and code generation process should scale to thousands of dynamic page templates and hundred of thousands database queries. In these contexts, applying presentation styles and client-side logic to page templates manually is unfeasible, and these features should be applied automatically in a bulk manner.
- The generated code should perform and scale well, and comply with the requirements of Web caching architectures, especially those for caching dynamic page templates.

In the remainder of this paper, we address each of these problems and the solution adopted in WebRatio; the design principles that will be discussed are indeed very general, as they apply to arbitrarily designed Web applications (including those which are manually coded) and to any Web development environment supporting a clear separation between data extraction, business logic, and presentation.

2 Software Architecture for data-intensive Web Applications

The simplest way of organizing the architecture of a data-intensive Web application goes under the denomination of **template-based approach**. Each page of the application that publishes dynamic content is mapped to one page template, which includes the static markup of the page and server side scripting instructions, which typically perform three tasks:

- The decoding of the parameters of the HTTP request.
- The preparation and execution of the database queries or component calls necessary to retrieve content or execute operations.
- The generation of the dynamic part of the page from the database content.

In the template-based approach, each client request is addressed to one page template, which accomplishes all the functions required for building the response, possibly interacting with business components.

The template-based approach is simple to master, because of the one-to-one mapping between pages and templates, but suffers from several problems that make it impractical for large applications:

1. The source code of the page template is overloaded with too many functions, which are heterogeneous in nature (request decoding, data extraction, interaction with business logic, markup generation).
2. The control logic is scattered through the templates and hard-wired; each template embeds the URLs pointing to the other templates callable from that page, and thus any change in the hypertext topology or control logic of operations (e.g., to which page redirect the user in case of operation failure) requires intervention on the code of the template.
3. Functions embedded inside templates are not reusable in other templates.
4. If server-side scripting languages are used, source code maintenance is problematic, because HTML markup is mixed with programming instructions.

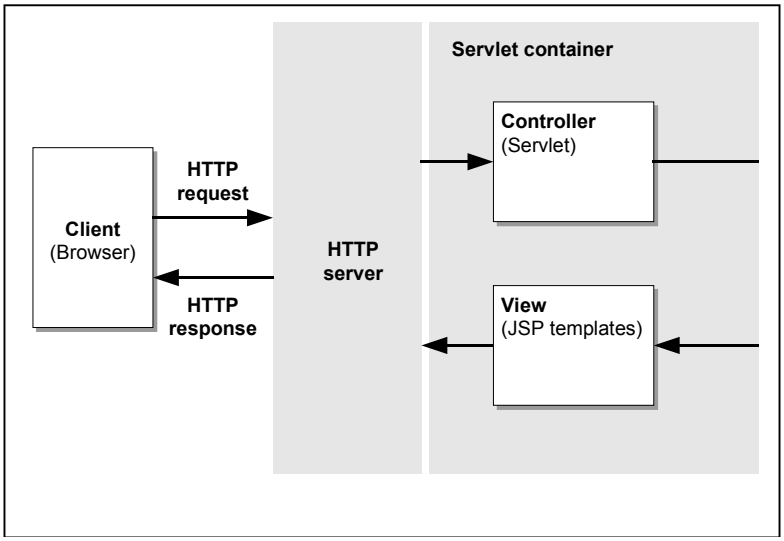

Figure 3 - The MVC architecture applied to Web applications

5. Many presentation aspects are hardwired to the source code of the template, both in the static HTML and in the programming instructions for building the dynamic content.

These problems are in part alleviated by the use of **server-side tags** instead of server-side scripting languages, as advocated by the JSP 1.1 and ASP.NET architectures. A server side tag is an XML tag inserted into the page template, which hides the presence of a server-side component for dynamically producing content. The use of server-side tags cleans up the page template from most server-side scripting instructions, making the source code more easily editable by the graphic designer, and encapsulates into tags frequently used functions, like the production of markup from the result sets of queries and the validation of user input, into components that can be reused in multiple templates. However, server-side tags are not a panacea: the global control logic of the application remains hard wired to the templates, and server-side tags still mix the presentation and data extraction logic.

An improvement to the template-based architecture is granted by the so-called **presentation frameworks**, which are software architectures that apply to the presentation layer of Web applications the soundest design patterns of modern software engineering. One of the most powerful presentation frameworks is the so-called *Model-View-Controller* (*MVC* for short). The MVC is conceived to better separate and insulate the three essential functions of an interactive application:

- The business logic of the application (*the Model*).
- The interface presented to the user (*the View*).
- The control of the interaction triggered by the user's actions (*the Controller*).

In the MVC architecture, the computation is activated by a user's request for some content or service. The request is intercepted by the Controller, which is responsible of deciding which action should be performed for servicing it. The Controller dispatches the request, in the form of a "request for action", to the suitable component of the Model. The Model incorporates the business logic for performing the action, and executes such logic, which updates the state of the application and produces a result to be communicated to the user. The change in the Model triggers the most appropriate View, which builds the presentation of the response. Such presentation typically embodies interaction objects, whereby the user may pose a new request and reactivate the computation process.

In the Web context, the original MVC is adapted to take into account the peculiarity of HTTP as a client-server protocol. Figure 3 shows the adaptation of the classical MVC architecture to the Web context, using Java as a reference platform. The illustrated scheme is sometimes called *MVC 2 architecture*.

The emitter of service requests in the MVC 2 architecture is the Web browser. When the user clicks on a hyperlink in the HTML page, an HTTP requests is addressed to the HTTP server, which may route it to the servlet container, where a program acting as the Controller intercepts it. The Controller decides the course of action necessary to service each request. The possible actions are contained in the Model in the form of object-oriented components (sometimes called **action classes**). The Controller maps the HTTP request to the suitable action, by creating an object of the action class and calling one of its functions.

Each **action class** is a Java class wrapping a particular application function, operating on the state of the application. Example of actions could be execution of a database query, the sending of e-mail, or the authentication

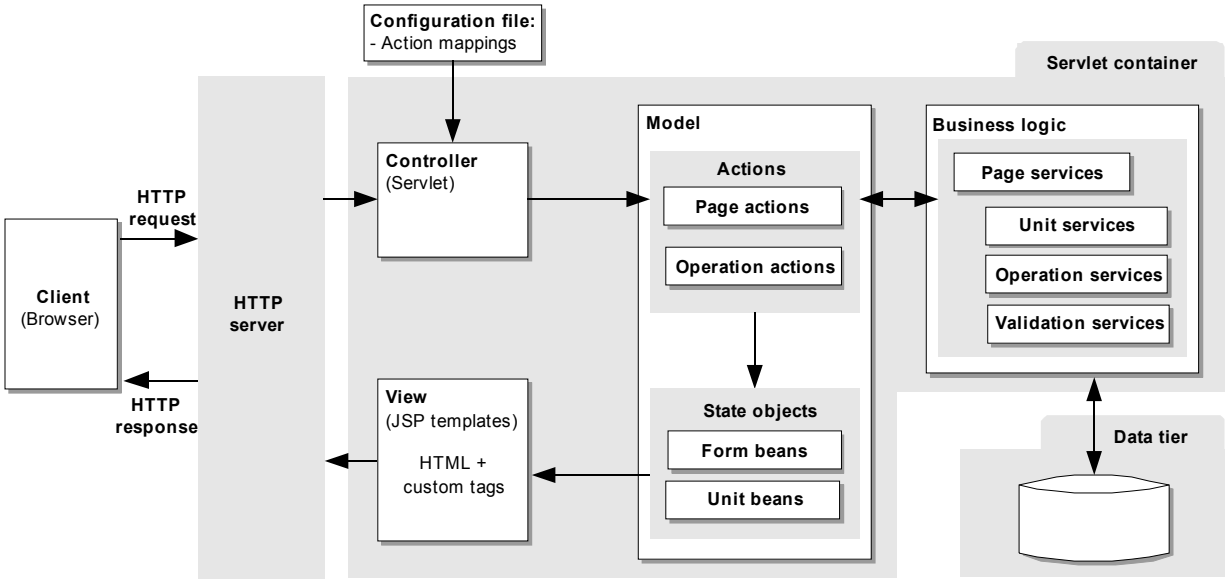

Figure 4 - Mapping WebML concepts to the MVC architecture

of the user. If the invoked action needs to update the state of the application, it may create or modify appropriate objects of the Model, called **state objects**, which represent the state of the application. State objects may last just the time needed for servicing the request, or persist between consecutive requests; for example, they may store the result of a data retrieval query, or the trolley items of the user. After completion, the action communicates the outcome of execution to the Controller, which decides what to do next.

In the typical flow of control of a Web MVC application, after an action completes, the Controller invokes a JSP page template, which is part of the View. The JSP template is responsible of presenting the updated state of the application to the user; for doing so, it accesses the state objects of the Model, where the current state of the application is stored, and builds the HTML page, which is sent back to the browser. Examples of views built after the execution of an action could be the display of the result of a database query, the notification that e-mail has been sent, and the home page of the Web site after the successful login of the user.

3 The MVC architecture of WebRatio

WebRatio adopts an MVC-based organization, in which the components produced by the code generators fit into a well-established framework. The key aspect of the WebRatio architecture is the mapping of the various hypertext primitives of WebML (pages, content units, and operations) into the boxes of the MVC 2 architecture. This mapping is schematically illustrated in Figure 4.

Each **WebML page** is mapped into four elements: 1) a *page action* in the Model, 2) a *page service* in the business tier, 3) a *JSP template* in the View, and 4) a *page action mapping* in the Controller's configuration file.

- The **page action** is an instance of an action class: it extracts the input from the HTTP request and calls the page service in the business tier, passing to it the needed parameters. When the invoked page service terminates, the page action notifies the Controller of the outcome of page computation.
- The **page service** is a business function supporting the computation of a page. It exposes a single function `computePage()`, invoked to carry out the parameter propagation and unit computation process. The page service updates the state objects in the Model: at the end of the page service execution, all the JavaBeans storing the result of the data retrieval queries of the page units (called **unit beans**) are available to the View.
- The **page template** in the view computes the HTML page to be sent to the user, based on the content of the Model. The page template contains the static HTML needed to define the layout where the units are positioned, and **custom tags** implementing the rendition of WebML units.
- The **action mapping** is a declaration placed in the Controller's configuration file that ties together the user's request, the page action, and the page view.

Each **WebML unit** maps into two components of the MVC2 architecture: a unit service in the business layer, and a custom tag in the View. Note that units do not contribute actions in the Model, because the Controller knows only

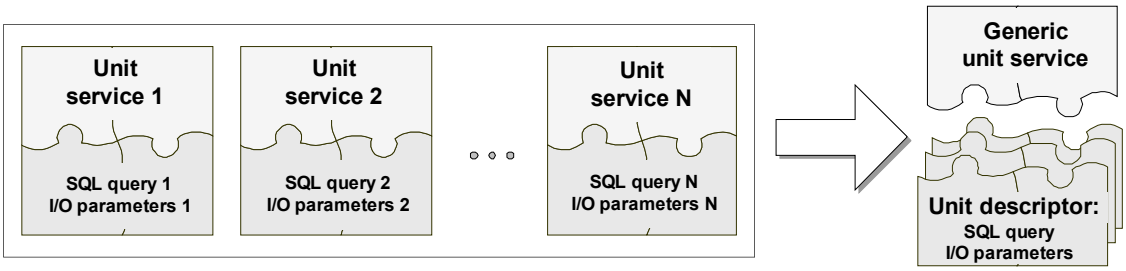

Figure 5 - Unit-level services versus generic unit service plus descriptor

about pages, and is unaware of the units contained in them, which are not exposed as individually callable actions.

- A **unit service** is a Java class, which is responsible for computing the unit's content and producing a collection of **unit beans**, which are JavaBeans objects belonging to the Model, holding the content of each unit. The class encapsulates the instructions needed to assemble the data retrieval query, execute it, and package the results into suitable unit beans.
- In the View, content units map to **custom tags** transforming the content stored in the unit beans into HTML. Such tags could be generic tags taken from a standard tag library, or WebML-aware tags, defined on purpose to match the features of WebML units.

Each **WebML or user-defined operation** maps into two components of the MVC2 architecture: an operation service in the business layer, and an action mapping in the Controller's configuration file, which dictates the flow of control after the operation is executed. Note that operations do not contribute templates to the View, because they do not directly display content.

The MVC2 architecture of WebRatio solves two of the key issues of template-based architectures:

- It factors out of the page templates the control logic, which is centralized in the Controller's configuration file.
- It separates presentation from business logic, by associating the former to the View templates, and the latter to the Model and business classes.

In particular, changing the business logic and the data retrieval logic no longer impacts the building of the dynamic markup, as long as the objects used to represent the application state in the Model maintain their interface.

4 Scaling the software architecture to large applications

The MVC architecture is a big step forwards in the direction of facilitating the maintenance of data-intensive Web applications. However, when the application is very large, the MVC architecture does not alleviate the problems associated with the size of the application:

- Every unit and operation requires a **dedicated service** in the business tier. If units are many, a very large number of services must be developed and maintained. All the services of individual units of the same kind (for instance, index units or create units) are very similar, because they differ only for the details of the data retrieval or update query, and possibly for the properties of the data bean storing the query result. However, this similarity is not exploited to reduce the amount of code to build and maintain.
- Every page requires a **distinct page service**. These services are numerous and all similar, because they differ only for the parameters fetched from the HTTP request and for the sequence in which unit services are invoked, and parameters are passed from one query to another one. Again, similarities are not factored out.
- The business services are implemented as **programs executed inside the servlet container.** It would be more appropriate to implement them as full-fledged business components living in the application server, using a distributed object technology like Enterprise JavaBeans.
- The look and feel of the application is **hardwired to the JSP templates**. Changing the presentation style requires manual intervention on a large number of files. For example, updating the graphic style of all index units, for instance adding a mouse-over JavaScript effect, requires locating and manually updating the relevant mark-up in all pages.

To avoid the proliferation of page and unit services, WebRatio exploits *genericity*, a classical principle of software design. Unit services can be re-organised according to the pattern shown in Figure 5.

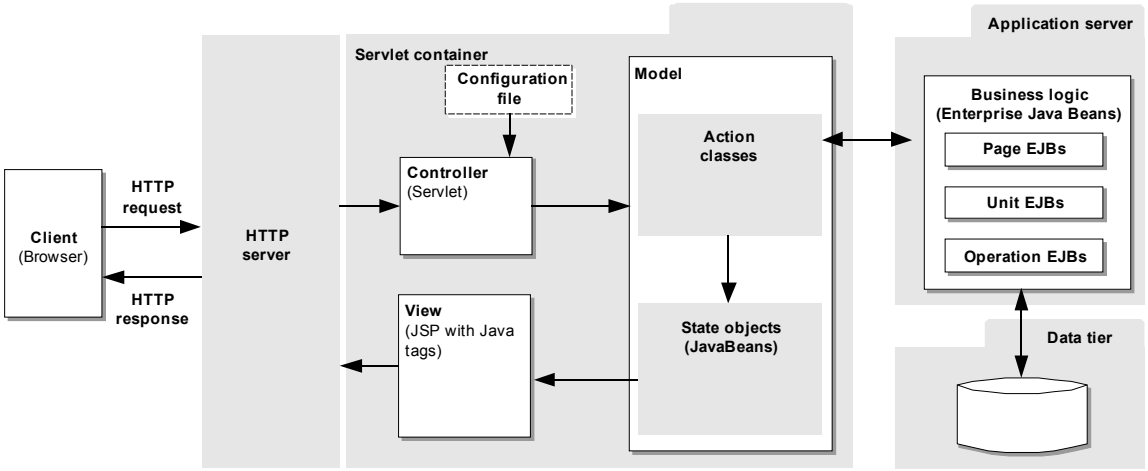

Figure 6 - The MVC 2 architecture embedded in the application server architecture

For each type of unit, a single *generic service* is designed, which factors out the commonalities of unit-specific services. This generic service is parametric with respect to the features of individual units, like the SQL query to perform, the input parameters of such a query, and the properties of the output data bean produced by the query. The unit-specific information can be stored in a **descriptor file**, for instance written in XML, used at runtime to instantiate the generic service into a concrete, unit-specific service.

The same design practice is applied to page services, but in this case the descriptor associated to an individual page is more complex, because it describes the topology of the page units and links, which is needed for computing units in the proper order and with the correct input parameters.

A second improvement concerns the reusability of the service in the business layer. In the MVC architecture described in Figure 4, the business logic components are implemented as Java classes executed inside the servlet container. This approach impose several limitations to the scalability and reusability of the implementation:

- Page and unit services live in the servlet container and cannot be called by other applications, for example by a non-Web application needing the same services. Therefore, non-Web applications do not share the business logic with Web applications, and must re-implement it, which introduces duplications, opens the way to errors and misalignments, and impairs maintenance.
- Cloning the machine where the servlet container resides duplicates also all the services of the application. The number of clones must be decided statically, and cannot be adapted at runtime. If the traffic of a certain application reduces, the objects implementing its services remain in main memory and occupy resources, potentially impacting other applications running on the same server.

A better software organization is obtained by splitting the business logic into the servlet engine and an application server, as shown in Figure 6. In particular, the role of the Model can be shared between the action classes living in the servlet container and business components implementing the page and unit services, deployed in the application server. In this case, the action classes call the appropriate business objects, which implement the actual application functions.

Figure 6 shows a concrete realization of the application server architecture, fitting the Java2EE platform. In this context, the business components are implemented as **Enterprise JavaBeans (EJB)**, an open standard for building server-side distributed components in the Java programming language. EJBs are deployed into the application server, which is called *EJB container*, and can be accessed by Web applications and other enterprise applications.

5 Managing presentation

Another fundamental issue in the development of large applications is the reduction of the effort necessary for updating the look and feel of the application across a large number of pages. Dealing with presentation requires addressing two distinct concerns, *graphic properties* and *layout*.

The effective management of graphic properties requires some care in the use of HTML: graphic properties should not be coded as tag attributes in the HTML mark-up, but should be factored out into *Cascading Style Sheets*, stored in separate files. A good practice in the definition of Cascading Style Sheets for WebML applications is to leverage the conceptual model to modularise the CSS rules. A set of rules can be designed for each WebML unit, by identifying the different graphic elements needed to present a certain kind of unit (labels of various kinds, cell

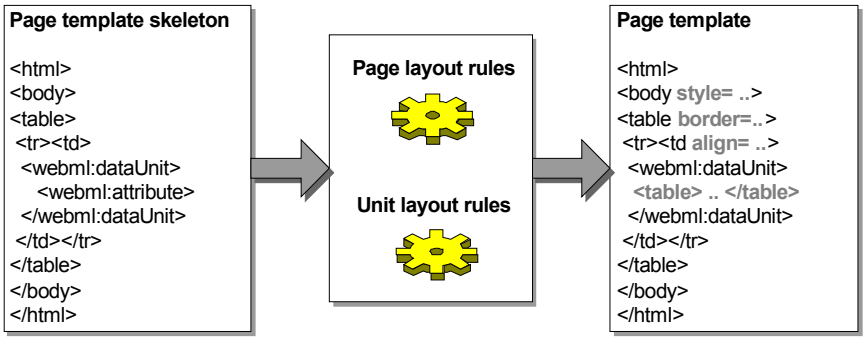

Figure 7 - Factoring out page layout rules using XSLT

backgrounds, and so on) and assigning to each element the proper graphic attributes using CSS.

Factoring out the layout from the JSP template of a page is more difficult, but can be done. An extremely effective technique exploits XSLT for defining layout rules for pages and units. The fundamental idea is to define the layout of the page and of the different kinds of units separately from the JSP templates, as illustrated in Figure 7:

- Producing a *page template skeleton*, which includes all the custom tags corresponding to the units of the page, but only the minimal HTML mark-up needed to define the layout grid of the page and the position of the various units in such a grid.
- Using *XSLT presentation rules* for transforming the template skeleton into the final page template, embodying the real presentation mark-up.

There are two kinds of XSLT rules: *page rules* and *unit rules*.

- **Page rules** match the outermost part of the skeleton's layout (for example, the top-level HTML table) and transform it into the actual grid of the page, which may include multiple frames, images, static texts, and other kinds of embellishments. For facilitating the writing of page rules, page layouts could be classified into general categories (for instance, multi-frame pages, two-columns pages, three-columns pages, and so on), and different rule sets could be designed for each category of layout.

- **Unit rules** match a class of units (for instance, index units, or data units) and produce the markup for their presentation. The produced markup includes the static HTML for laying out the content of the unit, the custom tags that actually produce the dynamic content of the unit from the unit beans, and the CSS attributes defining the unit look & feel.

A further benefit of presentation management with XSLT lays in the possibility of applying the presentation rules either at compile time or at runtime:

- Applying the rules at compile time yields a set of page templates embodying the final look and feel of the application; this approach is more efficient, because no template transformation is required at runtime.
- Presentation rules can be applied also at runtime, by publishing in the application server the template skeletons and transforming them on the fly, when the HTTP request arrives. This approach is more expensive in terms of execution time, because XSLT processing takes place at runtime, but is more flexible and may be very effective for multi-device applications. Different XSL rules can be designed addressing the presentation requirements of alternative devices; then, the most appropriate rules can be dynamically applied at runtime, based on the user agent declared in the HTTP request. In this way, the actual pages seen by the user have a presentation dynamically adapted to the access device, and the template skeletons plus the different XSLT rules serve the needs of a broad spectrum of access devices.

6 Optimisation and caching

A difficult aspect of automatic code generation is addressing performance optimization, an area where no software tool can replace human expertise. The architecture of WebRatio addresses these issues in a pragmatic way: rather than attempting complex optimizations, the code generator lets developers integrate their own optimized code in the software generation process. Developers can force the use of their code in two ways:

- By customizing the XML descriptor of units, where the data access queries produced by the code generator are stored. Replacing the automatically generated query with an optimized one and marking the descriptor as optimized forces the code generator to use the provided query.

- By customizing the unit service: each descriptor refers to the business component to use for filling the content of a unit; this component can be completely overridden by a user-supplied one, which may implement any required optimization policy.

Another subtle aspect of the MVC2 architecture applied to Web application is its relationship with Web caching architectures, which are a low-cost solution frequently used by developers to down-scale the server-side hardware necessary for meeting the performance requirements. First-generation caching solutions cached entire pages, and were inadequate for complex interactive and personalized Web applications, with pages composed of different content elements with different caching requirements. Last-generation cache technologies, like the Edge Side Include (ESI) initiative [ESI], apply more sophisticated caching strategies, based on the capability of marking fragments of the page template, which can be cached individually and with different policies.

However, the MVC architecture partly reduces the benefits of template-level caching, because the HTTP request does not invoke the page template directly, but an action class, which performs all the costly data queries *before the page template is parsed and executed*. In other words, caching fragments of the page template may spare only the computation of markup from query results, not the execution of the data extraction queries.

WebRatio solves this issue by adopting a two-level cache architecture: on one side, developers can use their favorite template caching product, for instance an ESI compliant one; in addition, developers can tag any WebML content unit in the conceptual model of the application as cached, and specify the associate cache invalidation policy. Then, WebRatio caches the data beans produced by the action invocations, which typically include the result of data access queries, and make them reusable by multiple requests. Moreover, since a conceptual model of the application is available, which clearly exposes the Entity or Relationship on which the content of a unit depends, and the operations that may act on such content, the implementation of operations automatically invalidates the affected cached objects, sparing to the developer the need of managing a business-tier cache in his application code.

7 Evaluation

WebRatio demonstrates the feasibility of applying CASE tools and automatic code generation to data intensive Web applications, preserving the peculiar aspect of Web development, like the production of high-quality presentation and the integration with state-of-the-art presentation frameworks and Web caches, and granting the scale-up to very large applications.

The interesting aspect of this experience has been the beneficial interplay of modeling and architectural issues. The positive results achieved in the development of applications can be ascribed both to the use of a high-level modeling language and to the careful definition of the software architecture.

- Having a high-level, yet formal language, as the starting point of application development permits one to exploit all the advantages of sophisticated software architectures, without incurring into the effort of managing the detailed and repetitive aspects of implementation. Just as an example, in the MCV architecture the configuration file of the Controller, which centralizes the control logic of the application, quickly becomes unmanageable when the application size increase; in WebRatio, it is automatically generated from the topology of the hypertext in the WebML diagram. The developer re-links the pages in the WebML diagram and the code generator re-builds the new configuration file.

- Implementing the tool on different software architectures helped us in clarifying the deep meaning of each modeling construct, and achieving that critical mix of abstractness and concreteness, which is necessary to ensure that the conceptual model is indeed conceptual, but at the same time can be implemented efficiently. In particular, the porting WebML on the MVC architecture, which was performed during the first quarter of 2002, prompted a revision of several language features and helped to clarify the semantics of the core WebML units and of the different ways in which they can be linked. After the porting, and as an indirect advantage of such experience, we have added to WebRatio the notion of "plug-in units", i.e. of new components, which can be easily plugged into the design and runtime environment, by providing their graphical icon, their unit service and rendition tags and the XSL rules for building their descriptors. Plug-in units are being used for adding to WebRatio content and operation units interacting with Web services and implementing workflow functionalities.

A special mention is deserved by the management of presentation, which has always been considered a killer of any attempt of automating the production of Web front-ends. We think that WebRatio pushed the solution to this problem further ahead, demonstrating that automatic implementation and high-quality interfaces are not incompatible goals.

The use of CSS and XSLT for managing the presentation features of a large application enforces a sound development workflow, which fosters a healthy distinction of responsibilities in the development team:

- *The graphic designer* establishes the categories of page layouts, writes HTML mock-ups for each class of page layout, and produces HTML mock-ups for the different

kinds of units. He defines "examples of presentation" and need not to worry about the actual coding of units or pages.

- *The XSLT programmer* transforms the page and unit mock-ups created by the graphic designer into XSLT style sheets. This activity is not difficult, because XSLT has an XML-like syntax, which blends well with the syntax of the HTML mock-ups. The XSLT programmer needs only to understand the structure of the custom tags representing the different kinds of units, and may ignore the way in which such tags are coded.
- *The application modeller* defines the application pages and the units in each page, and produces the template skeletons from the WebML model of the page, which is quite a trivial task.
- *The programmer* implements the custom tags and the business services behind them.

8 Experience

The WebRatio technology is in use since October 2000 and has been applied to several industrial applications. As an illustrative case, we present the results of an application developed for Acer Corporation, called Acer-Euro. This application[1] serves the customers and internal personnel of the 21 national subsidiaries of the Acer European branch, by organizing, collecting, managing and publishing on the Web content about Acer products. Acer-Euro addresses three categories of users: customers in the various nations, who are offered a mix of centrally administered data, for example product specifications and list prices, and locally produced data, for instance country-specific news and events; product managers, who verify and update data about the products; and marketing and communication managers, who administer marketing materials, like news and event lists. The Acer-Euro application is integrated with an extranet for managing the product distribution channel. The integrated application features 22 site views, 556 page templates, and 3068 units, for a total of over 3000 SQL queries.

All the page templates of the 22 site views have been automatically generated with WebRatio. Overall, less than 5% of the template source code and SQL queries needed manual retouching, essentially for optimizing query execution time or page presentation.

The MVC presentation framework with generic services has greatly reduced the code to maintain. A conventional MVC implementation would requires 556 Java classes for page services and 3068 Java classes for unit services. Using generic services and XML descriptors, only one generic page service is required (accompanied by 556 page descriptors, encoded as XML files) and 11 unit services (for the basic WebML units: data, index, multidata, multi-choice, scroller, entry, create, delete, modify, connect, disconnect), accompanied by 3068 unit descriptors. For each unit, developers can optimize the data extraction query working on the XML descriptor, and deploying the optimized version without interrupting the service.

Another area of substantial benefit is presentation management, where factoring out presentation into CSS and XSL rules has granted a substantial reduction of the presentation management effort: for all the 556 pages the look & feel has been produced by only three XSL style sheets (one for the B2C site views, one for the B2B site views, and one for the internal content management site views). Less than 5% of the HTML code produced by the XSL style has been retouched manually to improve the rendition.

9 Related Work

The MVC software architecture was introduced in [GHJV95]; its specific implementation for the Web context, the so-called MVC 2 architecture, is discussed in several textbooks and technical articles, including [ACM01, Davis01]. An open source implementation of the MVC 2 architecture can be found in [Apache02], together with many technical resources for developing Web applications. Architectural patterns for Web applications implemented in the Java 2 Enterprise Edition platform are discussed in the section of the Sun's Web site blueprint applications [Sun-BluePrint02]. The Enterprise JavaBeans API are specified in [Sun-J2ee02], together with tutorials and technical papers.

The development of Web sites with a model-driven approach has been specifically addressed by two important research projects, namely Araneus [AMM98] and Strudel [FFKLS98]. Both these methods allow the designer to separately define the site's structure and content. In the former, the Entity-Relationship model is used to describe the data structure, whereas a logical model called Araneus Data Model (ADM) is proposed to describe the site structure. A page scheme in ADM may include both atomic attributes (a text, an image, and so on) of a single object, similar to the WebML concept of data unit, and complex nested attributes representing sets of objects, similar to the WebML concept of index. A site is defined as a set of linked page-schemes. In Strudel, both the schema and the content of a site are described by means of queries over a data model for semi-structured information. Web sites are defined in a declarative way, by writing one or more queries over the internal representation of data, using the Strudel query language (StruQL). Some commercial tools [Hyperwawe98, Oracle02] provide a hypertext conceptual model, typically based on an extension of the Entity-Relationship model.

[1] The public part of the Acer-Euro application, i.e., one of the 22 developed site views, is reachable from http://www.acer-euro.com . The remaining site views are accessible only through the corporate VPN.

More information on WebML is available on www.webml.org [WebML02]; more on WebRatio on www.webratio.com [WebRatio02].

10 Conclusions

This paper has discussed an advanced architecture for deploying Web applications onto a Model-View-Controller framework, starting from high-level specifications based on the ER and WebML languages. The illustrated architecture is implemented in WebRatio, a commercial CASE tool for the automatic generation of data-intensive Web applications; this approach has been discussed in the context of the J2EE platform, but is very general, and can be used for implementing Web applications of arbitrary nature. We have shown the benefits of the adopted solutions, especially for what concerns the modularisation of the application code, the maintainability of the different aspects of the application, and the management of presentation. These benefits become more and more relevant as the application size increases.

11 References

[ACM01] D Alur, J Crupi, D Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall, 2001.

[Apache02] http://jakarta.apache.org/struts, 2002.

[AMM98] P. Atzeni, G. Mecca, P. Merialdo, A. Masci, G. Sindoni. The Araneus Web-Base Management System, *Proc. Int. Conf. ACM-SIGMOD 1998*, Seattle, USA , June 1998, pp. 544-546.

[CFP99] S. Ceri, P. Fraternali, S. Paraboschi. Design Principles for Data-Intensive Web Sites. *ACM-SIGMOD Record 28*(1), pp. 84-89, 1999.

[CFM02] S. Ceri, P. Fraternali, M. Matera. Conceptual modeling of data-intensive Web applications. *IEEE-Internet Computing*, 6(**4**), July-August 2002, pp. 20-30.

[CF+02] S. Ceri, P. Fraternali, A. Bongio, S. Comai, M. Brambilla, M. Matera. *Building Data-Intensive Web Applications*, Morgan-Kaufmann, to appear (winter 2002).

[Davis01] M. Davis. Struts, an open-source MVC implementation. February 2001, http://www-106.ibm.com/developerworks/library/j-struts/?n-j-2151,2001.

[FFKLS98] M. F. Fernandez, D. Florescu, J. Kang, A. Y. Levy, and D. Suciu. Overvew of Strudel - A Web-Site Management System. *Networking and Information Systems* 1(**1**), pp. 115-140, 1998.

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissedes. *Design Patterns - Elements of Reusable Object Oriented Software*, Addison Wesley, 1995.

[Hyperwave98] Hyperwave Information Management. Hyperwave User's Guide, Version 4.0. Munich, Germany, 1998.

[Oracle02] Oracle. Oracle9i Designer: Technical Overview. http://www.oracle.com, 2002.

[RAJ01] E. Roman, S. Ambler, T. Jewell. *Mastering Enterprise JavaBeans* (2nd edition). John Wiley & Sons, 2001.

[Sun-Blueprint02] http://java.sun.com/blueprints/patterns/j2ee_patterns/index.html, 2002 .

[Sun-J2ee02] http://java.sun.com/j2ee, 2002.

[WebRatio02] WebRatio Site Development Studio, www.webratio.com, 2002.

[WebML02] WebML Web Site, www.webml.org, 2002.