An Architectural Style for Data-Driven Systems

Reza Mahjourian

Department of Computer and Information Science and Engineering
University of Florida
Gainesville, FL 32611, USA
rezam@ufl.edu

Abstract. Data-driven systems and applications are specialized software solutions for acquisition, management, and presentation of information. These systems are usually developed using the same software tools, technologies, and processes used for creating any other type of software. Not only is this approach inefficient, but also it results in extreme redundancies due to the inherently repetitive nature of these applications. However, data-driven systems exhibit characteristics which can be exploited for extensive reuse across a single application or a family of applications. In this paper, we present XPage, an architectural style which is especially designed for building data-driven systems. We also provide several case studies from real-world deployments of XPage to help evaluate its efficiency and flexibility for developing real-world solutions.

1 Introduction

Data-driven systems are software solutions for information and data management. The two primary functions of these systems are acquisition and presentation of information. Information acquisition is typically performed using data entry forms or via interfacing with external data sources. Information presentation is concerned with retrieval and display of stored information to the user with appropriate navigation and querying facilities. Data-driven systems are also characterized by requiring intensive user interaction both for acquisition and retrieval of information. They are beyond doubt among the most common types of customized software systems in use today. University registration systems, e-commerce applications, content management systems, financial and accounting applications, a personal address book, and an online photo album are a few examples of data-driven applications.

Despite the existence of a consistent demand for development of new data-driven systems, they are mostly developed as one-off projects, with little reuse taking place beyond what is offered by the development technologies and programming languages used. Recently, the software industry has introduced some development frameworks which offer higher-level programming libraries to help with rapid development of data-driven applications. However, these frameworks are not high-level enough to prevent the repetitive nature of data-driven systems from showing up in the final programs as repetitions of nearly identical code segments or constructs. Moreover, none of these industrial frameworks offer an explicit software architecture, and the software engineering decisions behind their designs are buried in their implementations.

Academic research on producing agile techniques or methods for developing data-driven systems is severely lacking as well. Software engineering researchers consider data-driven applications to be in the realm of database research, because of their concentration on information management tasks. In addition, the seemingly primitive nature of "reading and writing structured data" appears to be lacking the necessary complexity to qualify as an interesting software research problem. On the other hand, database researchers have little interest in solving the software engineering challenges involved in streamlining development and managing complexity of software systems. In spite of that, it is quite surprising to know that the very little work done in this area comes from the database research community, and not from the software research community.

In this paper, we present an architectural style [1, 2], which is specifically designed for creating and maintaining data-driven systems. This architectural style has been extracted from a software framework we developed in 2001 and gradually extended afterwards. We have dubbed the architectural style *XPage*, following the name of the original framework.

Before discussing the XPage style in detail, we are going to review the related work in Sect. 2. Section 3 presents the architectural style and its key components and connectors. Section 4 provides three cases studies from real-world applications developed based on this style. Finally, Sect. 5 wraps up the paper with conclusions.

2 Related Work

A comprehensive framework for developing data-driven applications should address a wide array of concerns, from providing efficient data storage and retrieval mechanisms, to handling complex user interactions in the presentation and view layer. To our best knowledge, no other architectural style has been proposed to support development of data-driven systems to this extent. However, there are some solutions proposed by the database research community which focus on related problems.

The most notable example in this category is WebML [3, 4]. WebML is a product which provides a model-based development environment with a database-oriented view. The core of the application is created with a "structural model" which outlines the data model. Special-purpose data-aware "units" or "operations" are provided for data presentation or manipulation tasks. A program is created by associating these special units with the objects defined in the structural model. A "navigation model" is used to establish the links between different pages and content units.

In [5, 6], Vigna proposes a solution based on developing the entire application out of the Extended *ER* (Entity Relationship) model [7]. In their solution, the cardinality constraints on entity relationships in the ER model are used to decide an appropriate presentation and navigation model for the application. Based on an augmented ER model, their software generates SQL statements for creating the required tables. The developer is expected to execute these statements to create the underlying database. User interface forms are also automatically created based on the ER model. Afterwards, the application can be customized directly by modifying the generated forms.

Even though the organization of data-driven applications is mostly influenced by the structure of their underlying data repositories, ER models lack the required expressive

power to specify the structure and behavior of an entire application. In any data-driven system, a key factor in deciding the appropriate navigation and presentation model is the predefined flow of information according to its underlying business processes. This information is not captured in the ER model. A flexible software development framework requires mechanisms for specifying the business logic and view organization of an application independently of its underlying data model. Another undesirable side-effect of using ER models is that since Relational Databases Management Systems (RDBMS) are not directly based on ER models, the ER-based development tools have to assume responsibility for creating and managing the relational database as well. However, this is inflexible and counter-productive, since in many real-world situations there are database experts who prefer to design and fine tune the database independently. A requirement for working with legacy databases poses a similar problem.

Recently, we have witnessed introduction of some industrial software development frameworks which enable web developers to create data-driven applications more efficiently and rapidly. Examples of these frameworks include Ruby on Rails and CakePHP. The core of these frameworks is based on the concept of *Active Records*, which provide a two-way mapping between object classes and database tables. Any instantiation or modification of Active Records is directly reflected on their associated tables. Foreign key relationships are exposed in Active Records by linking attributes of one object to the instances of the referred objects. To implement the logic of an application, these frameworks recommend developers to write "controllers", which are service entry points for user defined operations on the data. However, they do not offer any higher-level components for the view layer of an application. The *scaffolding* technique can be used to rapidly create the view layer code out of the structure of the Active Records. However, the produced artifact is low-level code and the relationship between this code and the original Active Record can be lost with subsequent modifications to the either artifacts. Another source of inefficiency with these frameworks is that the standard mechanism for retrieving data from Active Records involves traversing them row by row to reach individual data objects. This suggests a low-level programming style, which for many data-driven scenarios can be entirely abandoned for a high-level view of the "whole data set".

None of these solutions address the software engineering side of the problem. Although they facilitate implementation of data-driven applications, their lack of an explicit architectural design makes it difficult to analyze these solutions with regard to issues of interest to the software engineering community. Moreover, since the relationship between implementation-level constructs and the architectural components and connectors is not clear, it is not easy to determine their potential for reuse across different domains. Nor can one try to formalize a process for designing, implementing, and maintaining the components needed for these solutions.

Another class of solutions which are extensively employed in creating data-driven systems are various middle-ware technologies such as Enterprise Java Beans [8]. These middle-ware technologies offer standardized interfaces for accessing and manipulating data sources, and include basic services such as concurrency, distribution, security, and component naming and registry. Such technologies can provide the platform for handling the data storage and retrieval tasks in data-driven systems, and thereby answer one

side of the problem. However, they do not offer specialized solutions for the view layer. Another shortcoming of these technologies is that they do not suggest any particular architecture on their surrounding system. The assumption is that developers use "glue" code to instantiate, utilize and maintain these objects whenever necessary. Despite being flexible, this is less in line with the spirit of software architectures, which advocate reuse by formalizing exemplification of good engineering solutions.

In [9, 10] Mattmann et al. present the OODT reference architecture, which is a solution for locating remote data sources and aggregating data from distributed data providers. OODT components and connectors provide the services of data source registry, identification, and querying on top of the industrial middle-ware technologies. Although OODT components and connectors can be employed for creating data-driven systems, like middle-ware technologies, OODT does not offer any solution for the view layer of these systems, mainly because its focus is on a different problem. Abstracting and modeling the interactions in the view layer of data-driven applications is much more complex than modeling the data layer operations, which more or less exhibit a linear input-output model. Lastly, like middle-ware technologies, OODT's solution is "programmer-intensive" [9] as it does not employ a high-level description language.

3 The Architectural Style

In this section, we introduce the XPage architectural style and its accompanying development framework. First we provide an overview of the style and its key characteristics. We then proceed to introduce some of the individual components and connectors.

3.1 Overview

Overall Architecture An XPage application is comprised of a set of interconnected *View Pages*. A View Page can be regarded as an abstraction of a web page, or a desktop form. Each View Page, in turn, contains one or more *View Forms*. The View Forms are data-aware components which can directly interact with the end-user. XPage offers different types of View Forms for common information acquisition, manipulations, and presentation tasks. Each View Form is connected to one or more *Data Sources*. A Data Source abstracts the data model of the underlying data source or destination. Data Sources, in turn, are associated with *Data Adapters*, which are connectors whose function is to provide a consistent interface over different types of data repositories available to the application. View Pages and View Forms constitute the view layer of an XPage application, while Data Sources and Data Adapters constitute its data layer. Figure 1 shows the overall architecture of XPage and its key components and connectors.

XPage components and connectors rely on a predefined initialization and launch protocol for their operation. Upon receiving a request for a specific View Page from the end-user, a *Coordinator* connector locates the corresponding XML file and instantiates the View Page component. This process is repeated for the View Forms in the loaded View Page and for any other components and connectors referenced in them. Once the component and connector hierarchy is loaded, a sequence of events are propagated in the hierarchy starting at the root View Page component. Some of the key events are

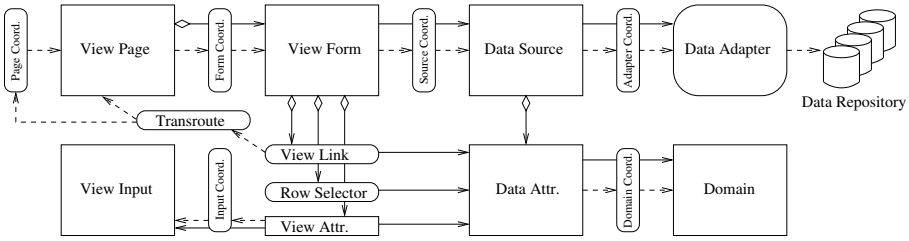

Fig. 1. The overall XPage architecture

load, init, register, process_read, process_write, and *commit*. User input and preferences are also passed to the components in the form of *Message* objects at various points in this sequence. Depending on its function, each component and connector may do a different task upon receiving the events.

Component and Connector Granularity The components and connectors of XPage are divided into two distinct groups, based on their granularity. The *coarse-grained* components and connectors, are first-level players in the architecture of an application. They bundle considerable amount of functionality to make them capable of handling significant data-driven responsibilities in a data-driven system. However, the XPage style defines these coarse-grained components and connectors in terms of a number of common *fine-grained* components and connectors. The fine-grained components and connectors are generalizations of the common structural and behavioral elements which constitute the coarse-grained components.

Communication In the data layer, components communicate with direct synchronous messages. In the view layer, communications take place by sending asynchronous messages which are carried by *Message* objects. Some particular interactions are so frequently used in data-driven systems that they demand for special treatment. For instance, in many cases, components rely on receiving foreign key parameters to determine what data item to display or manipulate. On a data entry form, a foreign key parameter must be received to establish a relationship between the newly created entity and its related entities in the database. Other frequent scenarios include requesting particular sort orders or filters on the presented information. In order to facilitate these interactions, XPage offers a Message type hierarchy, which covers various user interface events as well as inter-component communications.

Messages can be private or global. Private Messages have a particular recipient address, while Global Messages carry parameter-like values and are available to the entire component hierarchy. The private and global messages allow components to implement "push-", or "pull-"based communications, which are both handy in data-driven systems. In the view layer, all Messages are handled by a universal connector called the *Transroute* connector, which locates message recipients by their registered addresses. The Transroute connector is also responsible for processing user interface transitions such as submission of forms or loading of a new View Page.

The XPage Language All the coarse-grained and fine-grained components of XPage are configured using an XML-based domain-specific language. A set of all such XML files is enough to describe an XPage application. At run time, XML files are loaded to instantiate and initialize the components and connectors upon request. The XPage framework employs an object caching mechanism to increase the performance of the application.

Extensibility Clearly, complex applications have requirements which cannot be satisfied with the generic functionalities embedded in XPage components and connectors. Most components and connectors feature a number of *extension points* to let developers customize their behavior. The extension points are usually associated with the predefined events. There are two extension points for every event. For example, corresponding to the *process_write* event, there are two extension points called *before_process_write* and *after_process_write*. Developers can plug in custom code in these extension points to directly control the behavior of the components. For example, a data entry form can use the *before_process_write* extension to perform additional validations and potentially prevent the component from storing the data by canceling the *process_write* event.

In the following sections, we describe the XPage components and connectors in more detail.

3.2 Data Layer

Data Adapter – This coarse-grained connector is used to abstract away the heterogeneous interfaces of different types of data repositories. Whether the data repository is an RDBMS, an XML file, or a gateway to a remote web service, appropriate Data Adapters make them available to the application through a consistent interface which allows for data retrieval and data manipulation. Data Adapters translate the service requests into a language understandable by the underlying data repository. For instance, a request can be translated into a SQL query, an XPath query, or a web service invocation message. Data Adapters also offer transactional services to maintain the integrity of data repositories when multiple components need to collaborate for a single data operation.

Data Source – This coarse-grained component works on top of a Data Adapter. Data Sources are used to elevate the flat interface provided by Data Adapters to a hierarchical object model suitable for complicated interactions that view layer components need. Like Data Adapters, Data Sources provide data retrieval and manipulation interfaces, however in a more structured manner. Users of a Data Source work with individual *Data Attributes* which correspond to the columns in its data source or target. In addition, Data Sources can enforce various integrity constraints by collaborating with other Data Sources on related entities.

Data Attribute – In its simplest form, this fine-grained component corresponds to a column in a query definition. Data Attributes are associated with Data Sources. When the Data Source is retrieving data, its Data Attributes receive values for the corresponding columns. After a user requests the Data Source to retrieve a row of data, he is expected to contact its Data Attributes to get the retrieved values. Likewise, for

storing and manipulating data, the user is expected to populate the Data Attributes with desired values before asking the Data Source to perform the operation. In addition, Data Attributes respond to a number of Messages for filtering the data source or requesting a particular sort order. They pass these requests up to their parent Data Source.

Data Attributes have a type hierarchy which determines their features and capabilities. Two of the important Data Attribute types are *Primary Key Data Attribute*, and *Foreign Key Data Attribute*. Primary Key and Foreign Key Data Attributes are required for data operations like create, update and delete. They let the Data Source know which set of the data values populated in the Data Attributes should be used to locate the affected data items, and which set should be used to provide the new or updated data. They also guide the Data Source to enforce various integrity constraints. The type of a Data Attribute also determines to which requests that Data Attributes can respond. For example, it determines whether the attribute is updateable or searchable. More complex Data Attributes like the *Derived Data Attribute* can interface with auxiliary Data Sources to automatically calculate derived and aggregated values.

Domain – This fine-grained component is used to help guarantee the validity of data handled by Data Attributes. If a Data Attribute is associated with a particular Domain component, all requests for writing to or reading from that Data Attribute pass through the associated Domain component for validation. Each Domain component provides two services of *read* and *write*. In addition to checking validity of values, these two services can also convert between internal and view-level representations of data values. For example, thousands separators can be automatically added and removed for numbers upon reading and writing of the data.

Figure 2 depicts the exchanged messages for an example data retrieval scenario. An external entity first configures the Data Attributes of a Data Source and then retrieves one row of information. Some internal messages are not shown.

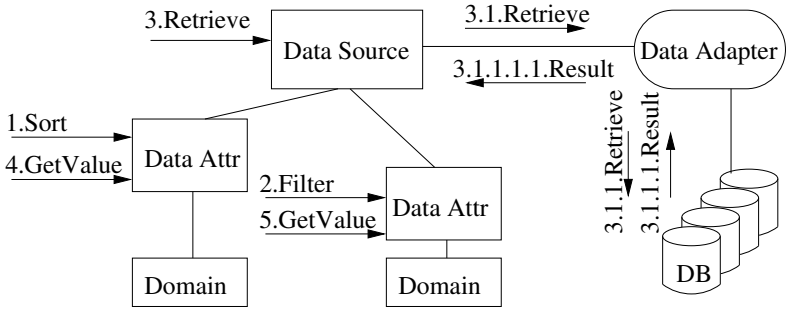

Fig. 2. Example data layer interaction

3.3 View Layer

View Page – An XPage application is implemented as a set of View Pages, which are coarse-grained components. View Pages contain the data-aware View Forms, as well as some presentation-only components like *Icons* and *Headers*.

View Form – View Forms are coarse-grained components with specialized functionalities, yet similar architectures. Typically, each View Form is associated with a Data Source which serves as the source and/or destination of data. Currently, there are five types of View Forms, corresponding to the five primitive operations on data: *Create Form, Read Form, Update Form, Delete Form,* and *Search Form*. These components are packed with common services which are usually required in implementing a data-driven application. For example a Read Form can automatically present the data in an underlying Data Source in grid format or itemized format. It can paginate the data rows, and automatically change the sort order if the end-user clicks on one of the grid columns. It also allows users to download its data set as a file. A Search Form can present the end-user with a data entry form for specifying filter criteria and then pass the filters to an associated Read Form for displaying the search results. A Create Form can automatically validate user input and warn the user if required entries are missing or invalid. A Delete Form can consult the integrity constraints in the data model to check for validity of a delete operation before attempting it. All these services are either provided by default or specified in configuration files at the conceptual level. These services are realized by collaboration of a number of common finer-grained components in the View Forms, which are described below.

View Attribute – These fine-grained components correspond to the individual data-aware "element types" on a View Form. Each View Form has a number of View Attributes. Typically, each View Attributes is connected to some Data Attribute from the View Form's associated Data Source. The connection between View Attributes and Data Attributes is established during component initialization. The type of collaboration between the View Attribute and the Data Attribute depends in part on the containing View Form. For example, on a Read Form, View Attributes receive the retrieved data from corresponding Data Attributes, but on a Create Form, View Attributes send user input value to Data Attributes for storage. View Attributes also respond to some specific Messages. For example, upon receiving a Filter Message, the View Attribute sends a filter request to its associated Data Attribute, which in turn is routed to its parent Data Source. Although View Attributes are more concerned with the logic of data operation, they also carry some presentational semantics based on their types. For instance, on a Read Form, View Attributes end up appearing as the header of data columns in the data grid shown to the user.

View Cell – These are fine-grained components which represent the individual data-aware elements. A View Cell may represent an individual data entry field on a form, or an individual value in a grid of displayed data. View Cells are not defined in the configuration files. Rather they are produced at run time during the operation of the View Forms. For example, on a Read Form, for each retrieved data row the View Attributes instantiate new View Cells. After retrieving all the rows, a grid of View Cells is formed which is displayed to the end-user. On data entry and manipulation forms, View Cells are instantiated during component initialization and are represented as individual data

entry fields on the GUI. All View Cells maintain links to the original View Attributes that instantiated them to pass the messages they receive.

View Input – These are fine-grained components which represent the user interface widgets. Every View Cell which represents a data entry input is associated with a View Input. The type of View Input determines how that View Cell is represented on the user interface. Example View Inputs are text fields, multi-line text fields, drop down lists, checkboxes, etc. Since drop downs are heavily used in data-driven applications, they receive special treatment in the XPage style. By default, any View Attributes linked to a Foreign Key Data Attribute is represented as a drop down input on the user interface. As its parameter, the drop down View Input receives the Data Source component matching the target entity of the foreign key relationship. This enables the drop down input to display appropriate values from the referenced entity.

View Row Selector – View Row Selectors are fine-grained connectors whose purpose is to receive special filter requests via *Select Messages* and relay the filter to their associated Data Attribute. Any View Form can have a number of Row Selectors in addition to its View Attributes. The effect of sending a Select Message to a Row Selector is almost the same as sending a Filter Message to an ordinary View Attribute. They both result in limiting the data rows which are retrieved, or manipulated. However, the semantic difference is that Row Selectors are used when the View Form's operation relies on receiving the message. For example, an Update Form usually needs to work with an individual data row and refuses to operate if it does not receive a proper Select Message, because otherwise it may affect unintended data rows.

View Link – View Links are fine-grained connectors which link different View Pages in an XPage application. Depending on the containing View Form, a View Link may appear as a hyperlink taking the user from one page to the other, or as a form submit button. In either case, each View Link carries a number of Message objects. All communications between components on different View Pages take place through View Links, and are routed by the Transroute connector. For example, in a book list page, a View Link can be placed next to each row to allow the end-user to go to a book update page to modify that book. In this case, a foreign key value is sent to a target Row Selector in the book update page.

Figure 3 shows some of the view components and connectors involved in this example. Notice how the same View Attributes, View Cells, and View Links take different forms on the two types of View Forms. The View Link appears as a hyperlink on the Read Form, and as a submit button on the Update Form. When the View Link is on a data entry form, it also automatically carries the user input as a number of *Input Messages*.

View Template – Contrary to what their name suggests, none of the view layer components and connectors mentioned so far are concerned with their presentation. Instead, each visible XPage component or connector is associated with some View Template component, which is able to "draw" it on the user interface. Standard View Templates are provided in the XPage framework for all coarse-grained and fine-grained components. Developers can customize the presentation of an individual component without having to provide custom presentations for its contained elements.

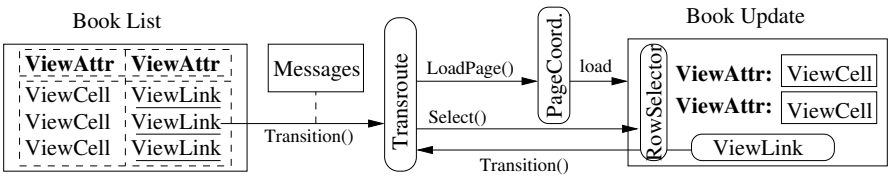

Fig. 3. Example view layer interaction

3.4 Support Components

Coordinator connectors are responsible for locating and loading other components during initialization of the component hierarchy. Based on the type of the requested components, Coordinators can decide intelligently whether they can reuse previously loaded components or not. This is particularly important for sharing Data Adapters for ensuring transactional integrity, as in order to successfully rollback transactions all related operations on a View Page must be handled by a single Data Adapter. Coordinators can also facilitate deploying the XPage style in a distributed environment, as they can transparently return stubs for remote components and connectors. Other XPage components facilitate user authentication, access authorization, navigation, and internationalization.

4 Case Studies

In this section, we briefly discuss some real-world deployments of the XPage style.

4.1 Squash

In [11], Esfahbod et al. use XPage to implement a web-based front-end for configuring an organizational gateway, which gives users controlled access to a set of internally administered infobases. The front-end allows the end-user to define available infobases together with a hierarchy of organizational users in different levels and groups. Access rights can be assigned to individual users or to organizational levels. From the information gathered in the database, appropriate configuration files are generated for a squid web proxy, which acts as the gateway to the actual infobases.

Despite unfamiliarity of this group with the XPage architecture and its framework, it took their three member team only six person-days to implement Squash. At that time they used an earlier version of XPage. In that version only the Create Forms offered drop down View Inputs, and this form of user input was not available in Search Forms. Since they needed such a feature on their search forms, and they did not want to bother modifying the XPage framework, they used the extension points as a workaround. They used a Create Form in place of the Search Form, however they overrode the *before_process_write* extension to cancel the create operation and instead route the user input messages to the Search Form for being processed as filter criteria.

4.2 BibIS

In 2006, XPage was used to develop a Bibliographic database for the Database Research Group at the Department of Computer Science at University of Florida. The core functionality of BibIS is to manage bibliographic data on publications. It allows end-users to manage publication types, enter publication information, and optionally upload article files. In addition to browsing and searching the publications, BiBIS allows users to generate BibTeX entries for any set of selected publications. Up to this point, the requirements could be satisfied using nothing but default XPage components and connectors.

However, the interesting requirement in BibIS was that all the publication attributes and publication types needed to be dynamically definable by the end-user. We used the extension points to satisfy this requirement. For the publication entity, an "empty" Data Source was defined with no Data Attributes. In the *before_load* extension point of this Data Source, we included custom code to load an auxiliary Data Source on the table containing publication meta-data. The custom code dynamically added Data Attributes to the publication Data Source to match the stored meta-data. Using this approach greatly reduced the complexity of this solution, since all other View Forms which worked with the publication entity were developed as if the publication table was a static table. Other extension points were used to update the structure of the actual publication table as the end-users updated the publication meta-data attributes.

4.3 Ringtone Vending Website

Our last case study is from the deployment of XPage on an Internet website for selling cell phone ringtones and logos. This system worked on three geographically distributed servers. A catalog server on the content provider's site offered web services for getting information on available content for sale. The web server presented the catalog to the Internet users and accepted orders. Received orders were sent to a GSM server, which communicated with the content provider to get the ringtone and then send it to end-user's cell phone.

Although there is no intrinsic support for web services in XPage, we used the extension points on two virtual Data Sources to provide the connection from the web server to the catalog server and the GSM server. The first virtual Data Source made a web service request to the catalog server upon a read request to get the catalog information. The second virtual Data Source was used as if it was saving user orders. However, instead it activated local scripts which sent order parameters to the remote GSM server.

A common theme that is seen in all the above experiences is an invariable need for extensibility in the architectural style. All these systems had requirements which were not predicted when XPage was designed. This confirms the importance of extensibility as a key requirement for such a generic software engineering solution.

5 Conclusions

We have presented the XPage architectural style for creating data-driven systems. XPage facilitates reuse at the code level by offering a conceptual domain-specific language.

However, more importantly, it facilitates reuse at the architectural level by providing an efficient break down of responsibilities in the generic coarse-grained and fine-grained components. As part of future work, we are looking forward to presenting the software engineering challenges that we faced when designing XPage and the guidelines that we followed to address them.

Our successful experience with XPage shows that it is possible to streamline many activities involved in design and development of data-driven systems. However, we believe data-driven systems have much more capacity for reuse and we are looking forward to seeing more research devoted to discovering techniques and methods for exploiting this potential.

References

1. Perry, D.E., Wolf, A.L.: Foundation for the study of software architecture. Software Engineering Notes **17**(2) (1992) 40–52
2. Garlan, D., Shaw, M.: An introduction to software architecture. Advances in Software Engineering and Knowledge Engineering **2** (1993) 1–39
3. Ceri, S., Fraternali, P., Bongio, A.: Web modeling language (webml): a modeling language for designing web sites. In: Proceedings of the 9th international World Wide Web conference. (2000) 137–157
4. Ceri, S., Fraternali, P., Matera, M.: Conceptual modeling of data-intensive web applications. IEEE Internet Computing **6**(4) (2002) 20–30
5. Vigna, S.: Erw: Entities and relationships on the web. Poster Proc. of Eleventh International World Wide Web Conference (2002)
6. Vigna, S.: Automatic generation of content management systems from eer-based specifications. ASE **00** (2003) 259
7. Chen, P.P.S.S.: The entity-relationship model: Toward a unified view of data. ACM Transactions on Database Systems **1**(1) (1976) 9–36
8. Sun-Microsystems: Enterprise java beans. http://java.sun.com/ejb/
9. Mattmann, C.A., Crichton, D.J., Hughes, J.S., Kelly, S.C., Ramirez, P.M.: Software architecture for large-scale, distributed, data-intensive systems. In: WICSA '04. (2004) 255
10. Mattmann, C.A., Crichton, D.J., Medvidovic, N., Hughes, S.: A software architecture-based framework for highly distributed and data intensive scientific applications. In: ICSE '06. (2006) 721–730
11. Esfahbod, B., Safy-Allah, H.: Squash: Design and implementation of a large scale http gateway and masqurader. Internet draft: http://behdad.org/ download/ Publications/ squashdoc/ squash.pdf (2003)