Investigating Model-Driven Architecture for Web-based Interactive Systems

Mohamed Taleb¹,², Ahmed Seffah¹, Alain Abran²

¹ Department of Computer Science and Software Engineering,
Concordia University, Montreal, QC, Canada
mtaleb AT encs.concordia.ca, seffah AT cse.concordia.ca,
² Software Engineering and Information Technology Department,
Ecole de technologie supérieure, Montreal, QC, Canada
alain.abran AT etsmtl.ca

Abstract Decoupling the various aspects of Web applications (for example, business logic, the user interface, navigation and information architecture) and isolating platform specifics from the concerns common to all Web applications are some examples of Web engineering challenges. This paper describes how the principles of the Object Management Group’s Model-Driven Architecture can be used to develop Web application, while at the same time ensuring their cross-platform portability and usability. In the Model-Driven Architecture proposed for Web applications, a set of 5 models is identified to provide a pool of proven solutions to these problems. The models span several levels of abstraction, such as Domain, Task, Dialog, Presentation, and Layout. Our proposed architecture shows how these individual models can be combined at different levels into heterogeneous structures which can be used as building blocks in the development of Web applications.

1 Introduction

In software engineering, an interactive system is a program with which the user engages in conversation (dialog) in order to accomplish tasks. It consists of two parts: the software part, which is referred to as the interactive application; and the hardware part, which supports the execution of the software. In turn, the software can be divided into two sub parts: the user interface, and the algorithmic part, which contains the semantics of the interactive application. The hardware in an interactive system consists of input and output devices, as well as various managers (device drivers), which provide the physical support required for the execution of the interactive application.

Over the past two decades, research on interactive systems and UI engineering has resulted in several architectural models. These constitute a
major contribution not only in facilitating the development and maintenance of interactive systems, but also in promoting the standardization, portability, and ergonomic "usability" (ease of use) of the interactive systems being developed. Such architectures provide: (a) a precise definition of the UI, which is aimed at: (i) presenting the output to the user; (ii) gathering user entries to transmit them to the interactive system procedures that will treat them; and (iii) handling the dialog sequence; (b) the separation of concerns, especially the decoupling of the UI from the system semantics; (c) the definition of reusable and standardized UI components; (d) the decentralization of dialog management, help, and errors across the various components of an interactive system; and (e) programming driven by events.

The Web infrastructure, including scripting languages and content management tools, offers major opportunities for developing advanced Web software systems, the latest of which are highly interactive and platform-independent, and feature various types of UI. As our reliance on Web-based applications continues to increase and the technologies supporting these applications become more complex, there is a growing concern about the manner in which the Web-based systems/applications are developed.

Web-based system development has generally been ad hoc, resulting in a variety of problems and poor quality [10]. The way these problems are addressed is critical to deploying successful large-scale Web applications. Furthermore, in developing these applications, technology is secondary to the information and the services that use and create the information. The goal should be to provide an environment in which those who produce individual Web services and information can do so in a way that is as independent of specific Web service implementation technologies as possible. The purpose of such an approach is to ensure the interoperability of Web applications as the underlying technologies change [3, 4].

Web engineering [7] emerges as a new branch of software engineering. In particular, it focuses on models, architectures, and tools which can be used as foundations for complex Web application design, development, evolution, and evaluation. More specifically, Web engineering is aimed at addressing the issues of scalability, maintainability, usability, configuration management, and other non-technical aspects. Examples of problems addressed specifically by Web engineering include: (1) decoupling the various aspects of Web applications, such as business logic, the UI, navigation, and information architecture; and (2) isolating platform-specific issues from the concerns common to all Web applications.

Model-Based Engineering (MBE) approach [8] has the potential to provide a comprehensive solution to some of these problems. MBE advocates the use of models as the key artifacts in all phases of development, from system specification and analysis to design and testing. Each model usually addresses a single concern among the issues involved in constructing the system. Thus, the basic functionality of the system can be separated
from its final implementation, the business logic can be separated from the underlying platform technology, etc. The transformations between models [5] provide a path that enables the automated implementation of a system to be derived from the various models defined for it.

However, existing Model-Based Web Engineering approaches [8] currently provide only high-level solutions for the design and development of Web applications. They do not address the specific concerns related to using separate models (navigation, presentation, data, etc.), and they are not supported by compilers that produce most of the application’s Web pages based on these models. Such approaches have some limitations, especially when it comes to modeling concerns such as interoperability. Furthermore, current Web applications need to interoperate with other external systems, which require integration with third-party Web services and portals, and also with legacy systems.

In the Model-Driven Architecture (MDA) proposed for Web applications, a set of five models is identified to provide a pool of proven solutions to these problems. The models span several levels of abstraction, such as Domain, Task, Dialog, Presentation, and Layout. They are each instantiated, transformed, and combined at various levels of abstraction into heterogeneous structures, and are considered as building blocks in the development of these applications.

The remainder of this paper is organized as follows: section 2 introduces the evolution and challenges of Web applications. Section 3 introduces related work on MDA in general. Section 4 describes the proposed five-tier MDA. Section 5 primarily describes the MDA taxonomy proposed here, as well as some models that we have identified and formalized. Finally, section 6 presents a summary and directions for future work.

2 Web applications: evolution and challenges

There are two views of a Web-based application. In the first view (first generation), a Web application is seen as a series of HTML pages with limited possibilities in terms of interaction. Later, Web technologies led to applications combining existing objects and components (for example, the CORBA, Java, or COM constructs) to create Web services that were more interactive. These Web technologies generate the required WSDL, SOAP, and other XML documents. They also generate code (such as Java code) which binds the Web services to the intermediate and backend tiers. These technologies relieve the programmer of much tedious work, and they meet the requirement of avoiding the labor-intensive hand coding of technology-specific artifacts. Programmers ”turn the crank,” so to speak, and the tool generates a great deal of the necessary XML, WSDL, SOAP, and Java artifacts.

However, Web applications are (and must be) coarser grained than many of the currently existing objects and components, because the busi-
ness functions we need to present to other businesses generally contain much more application logic than there is in any single existing object or component. At the same time, Web applications have to present abstractions of finer grained functions that already exist; in other words, Web applications will be compositions of more primitive functionality.

The second generation of Web applications (second view) entails highly interactive services and distributed information (databases, online repositories, data intensive applications). The design is driven by business requirements and by the need to minimize network traffic from fine-grained interactions. The following questions have to be answered: What is the information that needs to be manipulated? What is the functionality that must be provided? The answers can be used to generate the artifacts that implement the services over some set of technologies. The Web application design vocabulary should describe the information and services in ways that are entirely independent of XML, WSDL, SOAP, UDDI, Java, and other Web service implementation technologies.

Tools are also needed to allow fine-tuning by those who are intimately familiar with these Web service technologies. One approach to devising them is to provide configurable generators presenting a range of options as to how to generate the necessary artifacts. Another approach, which is controversial, is to allow some freedom to modify the artifacts generated. The latter approach requires that tools be smart enough not to overwrite the engineer’s modifications when the design is enhanced and the generator executes again.

The principle of raising the level of abstraction above that of third-generation language code is already well established for some types of systems. We would never consider going back to hand-coding enterprise database systems and GUIs, because we rely on the productivity benefits of being able to produce most or all our GUI and database systems from higher-level models.

As will be highlighted in the next section, the MDA approach offers similar benefits for Web service development. Raising the level of abstraction also reduces disruptions to the system, or prevents them altogether, in the face of change. As well, MDA can be applied to Web applications to improve the resilience of implementations as Web service technologies evolve.

3 Related work

3.1 Overview

In this paper, we have adopted the definition of software architecture from [1]: "The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the re-
relations among them.” The architecture defines the components (such as modules, objects, processes, subsystems, compilation units) and the relevant relations (such as “calls”, “sends data to”, “synchronizes with”, “uses”, “depends on”, “instantiates”). It is the result of early design decisions that are necessary before a group of people can collaboratively build a software system.

A model should be precise enough to at least enable unambiguous analysis, and abstract enough to focus attention and provide insights. It should be simpler to understand than the thing it represents. If they are well-structured, they can make complex interactive systems understandable. Modeling helps users achieve consensus about what exists or what can be built, since it provides a focus on which they can agree or disagree. A good model does not have to be executable, but it must be readily validated against examples.

Models are commonly used to represent the flexibility of complex interactive systems. They can be viewed at many levels of abstraction, and complementary model views can be combined to provide a more intelligible and more accurate view of a system than a single model alone. Meservy and Fensternacher [2] claim that many software development experts have long advocated using models to understand the problem that a system seeks to address, and yet development teams commonly employ models only in the early stages of modeling. Often, once construction begins, the teams leave these models behind and never update them to reflect their design changes during the project.

Most software developers would agree that modeling should play a central role in every project [2]. However, there is no clear consensus on what that role should be, how modeling should be integrated with other development activities or who should participate in the modeling process [2]. The same questions arise when the model concept moves from “contemplative” status (i.e. interpreted by the human) to “productive” status (i.e. interpreted by computer processors). Moreover, where models were once only used in the design phase, and more recently at the construction stage, they must from now on be embedded in the software to allow automated development and dynamic evolution.

In 2001, the Object Management Group (OMG) introduced the MDA initiative [2, 5, 6, 9, 11, 12, 13] as an approach to system specification and interoperability based on the use of formal (i.e. definite and formalized) models. The main idea behind MDA is twofold: (i) to specify the system independently of the target platforms that support it; and (ii) to transform a platform-independent system specification into a specification for a particular platform and context. MDA separates the fundamental logic behind a specification from the specifics of the particular middleware that implements it. In other words, the MDA approach distinguishes between the specification of the operations of a system and the manner in which that system uses the capabilities of its platform. This architectural separation of concerns constitutes the foundation of MDA, with the aim of
achieving three main goals: portability, interoperability, and reusability [3, 4].

Indeed, a model is a formal description of some key aspects of an interactive system from a specific viewpoint. As such, a model always presents an abstraction of the "real" thing, by ignoring or deliberately suppressing those aspects that would not be of interest to a user of that model. In other words, the model is the main element of the system. Modeling constructs differ by ignoring certain of these aspects [9]. For example, an architectural model of a complex interactive system might focus on its concurrency aspects, while a financial model of a business might focus on projected revenues. Model syntax includes graphical or tabular notations and text.

Key opportunities and modeling challenges have been identified in [9], with an illustration of how the "Model" and the "Architecture" can be used to enable large-scale model-driven integration. The advantages of the models are the following: (a) validation of the correctness of a model is made easier; (b) production implementations on multiple platforms are made easier; (c) integration/interoperability across platforms is better defined; (d) generic mappings/patterns can be shared by many designs; and (e) models constitute an interactive system of tool-supported solutions.

Building the MDA comprises three main steps: (1) specification of the system independently of the platform that supports it (PIM); (2) specification of target platforms (PSM); and (3) transformation of the system specification (PIM) into a specification for a particular platform (PSM). This architecture represents the MDA and its transformation from a PIM to a PSM, and eventually to the implementation code of the interactive system. There is an other step that can be integrated into MDA process development: the business model (the Computation-Independent Model, or CIM), sometimes called a Domain model.

However, the MDA has some weaknesses as well: ((a) it does not provide a standard for the specification of mappings: different mapping implementations can generate very different codes and models, which can create dependencies between the interactive system and the mapping solution used; it must take into account a diversity of platforms exhibiting drastically different capabilities; for example, Personal Digital Assistants (PDAs) use a pen-based input mechanism and have an average screen size in the range of 3 inches; and (c) the architectural models must be located and compared to the life cycle of the UI; in particular, difficulties may arise related to the problem analysis (analyzing user needs), expressed generally in terms of tasks and interaction sequences, and to the concepts proposed by these architectures (agents, presentation components and dialog components).
3.2 New generation of platforms in interactive systems

In recent years, interactive systems have matured from offering simple interface functionality to providing intricate processes, such as end-to-end financial transactions. Users have been given more sophisticated techniques to interact with available services and information using different types of computers. Different kinds of computers and devices (including, but not limited to, traditional office desktops, laptops, palmtops, PDAs with and without keyboards, mobile telephones, and interactive televisions) are used for interacting with such systems. One of the major characteristics of such cross-platform interactive systems is that they allow a user to interact with the server-side services and content in various ways. Interactive systems for small and mobile devices are resource-constrained, and cannot support a full range of interactive system features and interactivity because of a lack of screen space or low bandwidth.

The mosaic of interactive systems and multiple platforms has led to the emergence of interactive systems as a sub discipline of software engineering with some specific challenges. One important question is how to develop and deploy the same system for different platforms without "architecturing" or writing code specifically for each platform, for learning different programming languages and the many interactive systems design guidelines that are available for each platform.

3.3 Why models?

In an attempt to segment the various aspects of interactive system architecture and isolate specific platforms from remaining issues, the interactive systems industry has adopted a layered approach. As with other multi-tiered architectures, such as the client-server architecture, there is a common information repository at the core of the architecture. The repository is accessed strictly through this layer, which, in addition to the functions listed, also enables decoupling of the data from the device-specific interfaces. In this way, device interactive systems need only deal with the standardized middleware interface, rather than the multitude of APIs put forth by database repository manufacturers.

Segmenting the architecture and reducing coupling to stringent specifications makes it possible to quickly understand how changes made to a particular component affect the remaining interactive system, since achieving these goals requires a consistent approach to applying both cognitive and social factors to UI design, and requires independent to coordinate their activities.

Interactive systems can also be much more efficient in managing heterogeneous environments. This is critical, as more and more systems will need to interact with very different platforms and devices, resulting in computing devices that exhibit drastically different capabilities: from the
PDA’s pen-based input mechanism and 3 inches average screen size to the typical PC, with a full-sized keyboard, a mouse, and an average screen size of 17 inches. Coping with such diversity requires much more than mere layout changes. Pen-based input mechanisms are slower than traditional keyboards, and are inappropriate for systems such as word processing, which call for intensive user input. Similarly, the small screens available on many PDAs provide only coarse graphics capabilities and would be ill-suited for photo editing applications, for example.

Another system engineering challenge is the diversity in computing platforms, ranging from the traditional desktop to the mobile phone via PDA. Certain form factors are better suited to particular contexts. For example, while walking down the street, a user may use his mobile telephone’s Internet browser to view a stock quote. However, it is highly unlikely that this user will review the latest changes made to a document using this device. Rather, it would probably be more logical, and definitely more practical, to use a full-sized computer for that task. This leads us to believe that the context of use is determined by a combination of internal and external factors, the internal ones primarily relating to the user’s attention while performing a task. In some cases, the user may be totally focused, while at others highly distracted by concurrent tasks. For example, a user driving a car can operate a PDA to search for a telephone number. Thus, external factors are determined to a large extent by the device’s physical characteristics. It is not possible to use a traditional PC while walking down the street, a practice quite common with a mobile telephone. The challenge for a system architect is, therefore, to match the design of a particular device’s UI with the set of constraints imposed by the corresponding context of use.

However, these guidelines differ from one platform or device to another. When designing a multi-device application, this can be the source of a number of inconsistencies. The Java ”look and feel” developed by Sun Microsystems is a set of cross-platform guidelines that can correct such problems. However, these guidelines do not take into account the particular features of a specific device, especially platform constraints and capabilities. This can create problems for a user requiring different types of devices to interact with the server side services and information of a system. Furthermore, for a novice designer or a software engineer who is not familiar with this mosaic of guidelines, it is difficult to remember all the design guidelines and how to use them effectively. It is sometimes difficult to make trade-offs among these principles when they come into conflict. The best solution is often arrived at through guesswork, or by resorting to other means.
4 The proposed five-tier MDA architecture

An important goal of the traditional three-tier architecture is to improve and facilitate the design of interactive systems. However, even though the principle of separating an interactive system into components has its design merits, it can also be a source of serious adaptability and usability problems in systems which provide fast, frequent, and intensive semantic feedback: the communication between the view and the model makes the interactive system highly coupled and complex.

Among the weaknesses of this architecture are the following: (1) no guidance is provided to encourage the designer to cope with the various aspects of the dialog, such as assistance or error-handling; and (2) the architectural models are poorly located in relation to the life cycle of the UI, which can lead, in particular, to difficulties concerning the passage of the problem analysis (analysis of user needs), expressed generally in terms of tasks and interaction sequences, and to the concepts put forward by this architecture (agents, presentation components, dialog components).

To alleviate some of these weaknesses, a set of concepts for this model proposes a five-tier architecture, model-driven generic classification schema for distributed systems (Figure 1). These five tiers constitute an evolution of the familiar three-tier distributed system architecture. The five-tier architecture has a front end that is concerned with presentation, a back end that is concerned with data and legacy systems, and a middle tier that provides business functions.

The difference between the traditional three-tier architecture and the proposed architecture is shown in Figure 1: the presentation tier is divided into a user tier and a workspace tier. This is done for reasons of architectural separation; the device-dependent aspects of presentation are placed in the user tier, and the device-independent aspects are placed in the workspace tier.

Another important point illustrated in Figure 1 is that business services, which are composed from lower level, finer-grained business functions and information entities, are separated in terms of how they are presented to end-users and how the user interacts with them. That is, the same business service might be presented as a Web service, as a Web page, as an application screen, or as a message to a wireless hand-held device. To reuse these composed business services in these different contexts, their core logic in the enterprise tier (which we used to call the "middle tier") should be decoupled from the other components. This core logic is independent of the implementation technologies used to present the service, such as WSDL, HTML, WAP (Wireless Access Protocol), and so on.
A number of models have been suggested; for example, the OMG’s Model-Driven Architecture [2, 5, 6, 9], Paternò’s Model-Based Design and Evaluation of Interactive Applications [3], Vanderdonckt’s Task Modeling in Multiple Contexts of Use [4], and Seffah’s Model-Based User Interface Engineering with Design Patterns [8].

In our work, we investigate how these existing collections of models can be used as building blocks in the context of the proposed five-tier architecture. Which models at which level solve which problem? The research has identified at least five types of Web model that can be used to create a model-driven Web software architecture for interactive systems. Our research project focuses on a subset of the proposed models in this categorization, consisting of the following: (1) Domain model; (2) Task model; (3) Dialog model; (4) Presentation model; and (5) Layout model, all of which are described below. Examples of models are also presented to illustrate the need to combine several types of model to provide solutions to complex problems at the six architectural levels.

A simplified prototype of the Environmental Management Interactive System is developed, in which the five models representing the interactive system are illustrated on a laptop platform. The interactive system and corresponding models will not be tailored to different platforms. This prototype illustrates the various models, as well as the transformation from one model into another, which is performed manually at this time, respecting the transformation rules described in section 5.6.
5.1 Domain model

The Domain model is sometimes, called a business model. Within the scope of user interface development, it defines the objects and functionalities accessed by the user via the interface. Such a model is generally developed using the information collected during the business and functional requirements stage. It defines the list of data and features or operations to be performed in different manners, i.e., by different users in different platforms.

The first Model-Based approaches used a Domain model to drive the user interface at runtime. In this context, the Domain model would describe the interactive system in general, and include some specific information for the UI. These Domain models describe the application in general and include some specific information for the user interface.

For example, a structure of the Login interface, which enables the user to identify himself or herself in order to access secure or protected data and/or to perform authorized operations. An implementation of the login interface of the interactive system for a laptop platform. It allows users to connect to the interactive system to engage in a conversation in order to accomplish their tasks.

Consequently, the only real way to integrate UI and system development is the simultaneous use of the data model. This is why recent model-based approaches include a Domain model known from system engineering methods. Four other models: Task, Dialog, Presentation, and Layout, have the Domain model as an input.

5.2 Task model

This model enables us to describe how activities can be performed to achieve the user’s goals when using an interactive system [4]. The use of Task models can develop integrated descriptions of the system from a functional and interactive point of view. Task models are typically hierarchical decompositions of tasks and subtasks into atomic actions [4]. In addition, the relationships between tasks are described taking into account the execution order or dependencies between peer tasks. The tasks may contain attributes about their importance, duration of execution, and frequency of use.

After establishing the Domain model for the system in this case study, the Task model can be interactively defined. Only high-level tasks and their relationships are portrayed. The overall structure and behavior of the interactive system are given. The structure provided is relatively unique for an Environmental Management Interactive System; the concrete “realization” of high-level tasks has been omitted.

A large part of many interactive systems can be developed from a fixed set of reusable components. In the case of the Task model, the more those high-level tasks are decomposed, the easier it is to use the reusable
task structures that have been gained or captured from other projects or systems. In this case study, these reusable task structures are documented in the form of patterns. This approach ensures an even greater degree of reuse, since each pattern can be adapted to the current use context.

The main characteristics of the environmental management system, modeled by the task structure can be outlined as follows: The interactive system’s main functionality is accessed by logging into the system (the login task enables the management task). The key features are ‘adding a guest’, which is accomplished by entering the guest’s personal information and by ‘selecting an environment task or subtask’ for a specific guest. The two tasks can be performed in any order. The selection process consists of four consecutively performed subtasks (related through 'Enabling with Information Exchange' operators: (1) Selecting Data Source to use; (2) Selecting Task or Subtask : (a) Data management, (b) Indicator management, (c) Presentation tool management, (d) Environmental pattern management.

Consequently, the development of the Task model and the Domain model is interrelated. One of the goals of model-based approaches is to support user-centered interface design. Therefore, they must enable the UI designer to create the various Task models. Three other models (Dialog, Presentation, and Layout) have the Domain and Task models as inputs.

5.3 Dialog Model

The Dialog model enables us to provide dialog styles to perform tasks and to provide proven techniques for the dialog view design. The Dialog model defines the navigational structure of the UI. It is a more specific model and can be derived in large part from the more abstract Task and Domain models.

A Dialog model is used to describe human-computer interactions. It specifies when the end-user can invoke commands, functions, and interaction media; when the end-user can select or specify inputs; and when the computer can query the end-user and present information [1]. In other words, the Dialog model describes the sequencing of input tokens and output tokens, and their interleaving. It also describes the syntactical structure of human-computer interactions. The input and output tokens are lexical elements. Therefore, in particular, this model specifies the user commands, interaction techniques, interface responses, and command sequences permitted by the interface during user sessions. Two other models, Presentation and Layout, have the Domain, Task, and Dialog models as inputs.

For example, the Wizard dialog pattern [14] emerges as the best choice for implementation. It suggests a dialog structure where a set of dialog views is arranged sequentially, and the "last" task of each dialog view initiates the transition to the subsequent dialog view. The Wizard dialog
pattern’s suggested graph structure. However, the sequential structure of
the subtask process must be slightly modified in order to enable the user
to view the details of multiple subtasks at the same time.

5.4 Presentation model

The Presentation model describes how the UIs are graphically and vi-
sually organized. This model exists at two levels: the abstract and the
concrete, which define the appearance and the form of the presentation of
the application on the Web page. This model provides solutions on how
the contents or the related services can be visually organized into work-
ing surfaces, the effective layout of multiple information spaces, and the
relationships between them. They define the physical and logical layout
suitable for specific Web pages such as home pages, lists, and tables.

A Presentation model also describes the constructs that can appear
on an end-user’s display, their layout characteristics, and the visual de-
pendencies among them. The displays of most applications consist of
a static part and a dynamic part. The static part includes the presen-
tation of the standard widgets, such as buttons, menus, and list boxes.
Typically, the static part remains fixed during run-time of the interactive
system, except for state changes like enable/disable, visible/invisible. The
dynamic part displays application-dependent data that typically change
during run-time (e.g. the application generates output; the end-user con-
structs application-specific data).

5.5 Layout model

The Layout model is realized as a instance of an interface. This model
consists of a series of UI components that define the visual layout of the
UI, the detailed dialogs for a specific platform, and their context of use.
There may be many instances of a layout model that can be derived from
the Presentation and Dialog models.

This Layout model makes it possible to provide conceptual models
and architectures for organizing the underlying content across multiple
pages, servers, databases, and computers. This model is concerned with
the ”look and feel” of Web applications and with the construction of a
general drawing area (e.g. canvas widget). All outputs inside a canvas,
which provides an easy and powerful way to draw structured graphics for
UIs, must be programmed using a general-purpose programming language
and a low-level graphics library. This model is then derived from these
four models: Domain, Task, Dialog, and Presentation.

5.6 Transformation Rules

Model transformation is the process of converting one or more models,
called source models, to an output model the target model of the same
system. Transformations may combine elements of different source models in order to build a target model. Transformation rules apply to all the types of model listed above. The following steps correspond to the transformation rules suggested by [2], and are considered part of our architecture:

- Maintain tracking structures of all class instances where needed;
- Maintain tracking structures for Association populations where needed;
- Support state-machine semantics;
- Enforce Event ordering;
- Preserve Action atomicity;
- Provide a mapping for all analysis elements, including:
  - Domain, Domain service;
  - Class, Attribute, Association, Inheritance, Associative class, Class service;
  - State, Event, Transition, Superstate, Substate;
  - All Action modeling elements.

6 A concluding remark

This paper has focused on a Model-Driven Architecture and on the elicitation of models for the development of Web-based interactive systems. We identified and proposed five categories of model for Web-based interactive applications to resolve some challenging problems, such as: (1) decoupling the various aspects of Web applications, such as business logic, UI, navigation, and information architecture; (2) isolating platform-specific problems from the concerns common to all Web applications.

Current limitations of the proposed five-tier MDA architecture include the following: (a) a need to define measures to objectively assess the applicability of the models that could be used in our architecture; (b) no encouragement to consider other aspects of the dialog which are very important to the user (help function or error-handling); and (c) no consideration of other quality attributes, such as communicability, learnability, maintainability, and usability.

The targets of the proposed five-tier MDA include the following: (a) facilitation of the use of models by beginners as well as experts; (b) support for the automation of both the model-driven approaches to design; (c) support for the communication and reuse of individual expertise regarding good design practices; and (d) integration of all the various new
Investigating Model-Driven Architecture for Web-based Interactive Systems

technologies, including, but not limited to, traditional office desktops, laptops, palmtops, PDAs with and without keyboards, mobile telephones, and interactive televisions.

Further work is needed to specify and formalize the proposed models using UML and XML, for example. Next, some transformation rules will have to be defined in concrete manner and some relationships will have to be defined between models, so that they can be combined to define the new architecture for developing interactive systems based on the architectural level and categories of patterns that we have defined in our previous research in this project and the various models proposed and defined in this paper.

References


Mohamed Taleb, Ahmed Seffah, Alain Abran


