AN ALTERNATIVE APPROACH TO BUILDING WEB-APPLICATIONS

Oleg Rostanin
University of Saarland, chair Siekmann
Stuhlsatzenhausweg, 3, D-66123 Saarbrücken, Germany

Keywords: Web applications, Model-Controller-View, multi-client support, e-learning, XML, database

Abstract: Nowadays in J2EE-world there is a lot of blueprints, articles and books that propose some recommendations, recipes and patterns for producing web-applications in right way. There are also ready decisions like Jakarta Struts or JavaServer Faces that can be taken as a base of a new project development. While developing the DaMiT e-learning system we tried to collect, analyse and implement many of the architectural features being proposed as well as to invent some new mechanisms such as supporting multiple kinds of client software or introducing XML-based interfaces between application tiers.

1 INTRODUCTION

Data Mining Tutor (DaMiT) is an e-learning project sponsored by the German Ministry of Education and Research. The main objective of the project was to create an adaptive e-learning system for teaching and studying of Data Mining in German universities. During the project time it got clear that the system should be developed as a general framework on which e-learning systems in different scientific, educational or industrial areas could be build. Such requirements would imply a higher flexibility and extensibility level of the system (extensible system of user roles, different interface requirements etc. - see (Schulz-Brünken et al., 2002; Grieser et al., 2003)).
On the other hand, at the beginning of the project there was no strict requirements to the user interface (layout, client implementation technology: html, flash, applet or java application etc.) whereas the requirements to the business logic and to the database structures were formulated more definitively.
The next speciality of the project was distributed software development because the project partners that were responsible for development of the different system parts (business logic, database procedures, flash client, exercise engine, certification authority software etc.) worked at least in 6 different German cities. As a rule, the development process of different modules was not 100% synchronized because of lack of human resources.

In such circumstances of fuzzy requirements, poor resources and relatively low communication possibilities between developers, we made some engineering and organisational decisions like:

- customizing the standard J2EE Model-View-Controller implementation for:
  - advanced navigation support;
  - supporting multiple web client types;
- introducing xml interfaces between the application layers;

in order not to waste humans work and to get at the end a high quality product that could be extended, adapted and configured according to the new requirements and application areas.
As a result of the project, we have got not only an universal e-learning system framework that was used only during the project lifetime for building 2 different e-learning systems (DaMiT and IPAC), but also an environment for creating large scalable web applications.
The main features of the DaMiT e-Learning System one can find in (Jantke et al., 2004).

2 SYSTEM ARCHITECTURE

To meet our needs we had to choose such an architecture that would allow us to make the main impact on

Rostanin O. (2004).
AN ALTERNATIVE APPROACH TO BUILDING WEB-APPLICATIONS.
In *Proceedings of the Sixth International Conference on Enterprise Information Systems*, pages 119-124
DOI: 10.5220/0002637601190124

the development of business logic of the application that could be tested with the help of quick-and-dirty html based interface. In such a case the layout could be easily changed or completely replaced. In addition to this, DaMiT was supposed to be a rather complex system. This two facts speak for building the system according to the MVC Model 2 pattern (Alur et al., 2001).
Once having chosen a basic architecture one must decide if it has a sense to use an existent implementation like Jakarta Struts or to design and implement some own framework that would be more powerful in some way. In case of DaMiT we decided for the own implementation that would be able to support new client types (such as Flash, Java applications and applets), provide the possibility of advanced navigation (redirect to the certain page in dependency of some condition etc.) and, last but not least, make the process of web programming as easy as possible in order that less qualified persons like students could participate the development process.

2.1 Classical Model 2 implementation

The classical implementation scheme of MVC-framework by the means of J2EE technologies is illustrated by the fig. 1. An approximate algorithm of work of such a system looks like following:

1. User clicks a link or submits an html form.
2. As a response to the user action, the browser sends an http request to the servlet that was specified by the *href* attribute of the anchor <a> html tag or by the *action* attribute of the <form ..> html tag. Every link or html form must always deliver some special parameters such as *command* or *action*, as well as *session_id* and some others. The names of obligatory parameters and their quantity vary between different Model 2 implementations.
3. The servlet plays the role of a request controller. It parses the http request, figures out the command to be executed and finally runs the *execute()* method of the corresponding *command* class.
4. The *command* class instantiates business-classes, calls their business methods and finally calculates some command dependent as well as obligatory parameters. As an example of an obligatory parameter can serve the flag indicating whether the command was successfully executed or not that allows to make a decision if the system's state must be changed or the system must be returned to the previous state.
5. After the command has been executed, the servlet decides to which url it must redirect the request.
6. If the control was given to a jsp page - the page reads data from the *Session* and the *Request* objects and displays them on the screen.

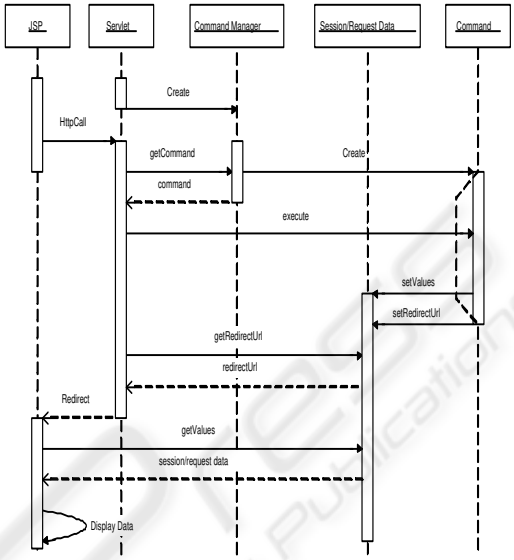

Figure 1: Sequence diagram of a MVC web application.

2.2 Configuration of the Framework

Let us introduce the notion of *MVC framework instance*. We will call *framework instance* a web system that was created on a base of a certain MVC framework. This could be Struts (Husted et al., 2002; Cavaness, 2002), JavaServer Faces (Budi Kurniawan, 2003) or DaMiT frameworks to name a few.
The functionality of such framework instance is defined by the set of so called *actions* or *commands*. In addition to this, the system developer must define all the possible system states (pages) as well as the rules for the system state changing. In other words, these rules determine the navigation paths in the system. Let us call these state changing rules *navigation rules*. Mathematically, navigation rules could be expressed by the formula 2.

$$q = f(c, r)$$

where $q \in Q$ - the resulting system state (page) from the set Q of all the possible system pages, $c \in C$ - the command, and r - the result of command execution.
The result r of the command execution can be calculated as follows:

$$r = f(c, I, S)$$

where I - user input or command parameters, S the state of the user session.
Although, r is a variable, it can take values only from the finite set R of the possible command results.

Formula 2: Navigation Rule

Thus, to design a new system on a base of some MVC framework, the system developer must define:

1. the sets Q of possible states (pages), C of commands, and R of command results;
2. dependencies of the type 2

Usually, these steps are done by filling in the corresponding xml structures that are specific to each MVC framework. We use in the DaMiT framework the structures shown in the listing 3.

```
<commands>
<facade>
<command name="process_login"
allowed_groups="0,1,3,4,5,6,7">
<display name="default" page="START_PAGE"/>
<display name="data_error" page="LOUT_INDEX_PAGE"/>
<command_class>LoginProcessCommand</command_class>
</command>
...
</facade>
<pages>
...
<page name="START_PAGE" save="true">
<client name="html">
<adaptor>de.dfki.damit.servlet.HTMLAdaptor</adaptor>
<conf>StartPageHTMLAdaptor.cnf</conf>
</client>
<client name="java">
<adaptor>de.dfki.damit.servlet.JavaAdaptor</adaptor>
<conf>StartPageJavaAdaptor.cnf</conf>
</client>
</page>
...
```

Listing 3: DaMiT system configuration.

The tag <command> contains the description of the command. The tag <command_class> defines the java command class that will be instantiated to process the request. The tag <display> defines possible alternatives of the system state changing in dependency of the command execution result.
Speaking in terms of our mathematical notation, the list of tags <command> defines our sets C, Q and R as well as the dependency 2.

The main difference between our approach and those proposed by Struts or JavaServer Faces is that we define the system states (pages) abstractly and does not tie them to the certain jsp page or some other action. We call our states *logical pages*. Each *logical page* is characterized by its *name* and the definite set of attributes. We divide the attributes into two main groups:

- session values (S) are computed and collected during the session of user interaction with the system. These values are stored in the *Session* object which lives until the user logs out from the system.
- request values are computed during the current http request and are disposed after the dialog step is finished.

The advantages of our approach are that:

- we can introduce some specific pages like RETURN (see 2.3) that provides the system support of the complex navigation paths;
- our system states are completely independent of the visualisation technology. We define so called *page adaptors* for each pair $\{page, client\}$ so that system states (pages) can be displayed correctly by each of the supported client types - see 2.4.

2.3 Advanced Navigation Support

Such complex web systems as DaMiT contain much functionality that should be easily accessible by the user from his web interface. Otherwise the system becomes unattractive or even unusable for the user. The standard solution of this problem is to provide a *drop-down menu* or *tab control* for accessing these functions.
In the DaMiT project we chose a two-level tab control (the first level for dividing the functionality into groups - *modules*, the second for accessing the functionality itself - *submodules*). Our system has 5 main standard modules as well as some dynamically generated modules:

- Study (contains references to all the learning resources - courses, glossary, search etc.).
- User data (gives access to the user profile - changing identification data and learning preferences).
- Import (allows content provider to import/update learning modules).
- Group management (allows administrator to insert a user to a learning group, so defining user rights for the learning content).
- Dynamical lesson modules (it is very convenient for the user to open each new lesson as a new module - so the learner has always a possibility to return to any previously visited lesson with only one mouse click).

Although such organization of the user interface makes the system comfortable, it bears the problem of the module persistence. We can illustrate this problem, taking the DaMiT system as an example:

1. The DaMiT user is working with a certain lesson (L1).
2. The user decides that he should read some material from another lesson (L2) to understand the material of the L1 and opens the new lesson as a new module.

3. The problem is that when the user finishes reading the L2 he should have a possibility to return to the L1. Having returned to the L1, the user should be on the same place in the L1 on which he leaved it. In other words each module should be able to restore its state.

One way to solve this problem is to invent some module-specific approaches like to store the identifier of the last displayed content for the lesson modules or to regenerate the menu structure according to the state of a certain module but these approaches are complex and often not universal. We propose the following decision:

- We describe a hierarchical application module structure as an xml structure (see listing 4, instantiate it in the application memory as a hierarchy of java *Module* objects (see fig 5) and save in the *Session* object.
- We introduce one more obligatory http parameter - *module* into each http request so that the application always knows in the context of which module the current command is being executed.
- Each module stores the history of http calls (*command call stack*) that were executed within this module during the user session, so that one can return to any of the module states.
- Each top level module knows the submodule that is active at the moment.
- We introduce a command *show_module* as a part of the application framework. This command reproduces the state of the module using the *module call stack*.

```
<modules>
<module name="DaMiT_LOGGED_IN"
allowed_groups="0,1,2,3,4,5,6,7">
<module name="STUDY" allowed_groups="1,2,3,4,5,6,7">
<module name="COURSES" allowed_groups="1,2,3,4,5,6,7"
default_command="goto_strt"/>
</module>
<module name="USER" allowed_groups="1,3,4,5,6,7">
<module name="PREFERENCES" allowed_groups="1,3,4,5,6,7"
default_command="udisplay&amp;mode=pref"/>
</module></module></modules>
```

Listing 4: Application module structure.

Command call stack allows us to build in some advanced navigation features into the framework.

Example 1 Let the command C1 display a universal form for editing user data (page P3) and it could be used for editing of both *learning preferences* (page P1) and *personal data* (page P2) that belong to the different system modules. After submitting the P3 the system saves the changed data into the database and further has to decide where to return: to the page P1 or to the page P2.

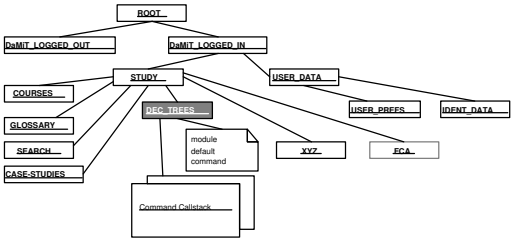

Figure 5: Application module structure.

Example 2 Let the command C1 display a universal page for access to the payment system (page P3) and it could be called practically from the every page of the lesson module (for example from the pages P1 and P2) if the lesson contains some information that is available for some fee. After the payment procedure was completed the system has to decide which content to display - P1 or P2.

We can solve such problems by introducing the logical page RETURN that makes the system to repeat the last command saved in the *module call stack* (fig. 6). It is interesting to notice that in the Example 1 the user sees the same picture that he saw before calling the editing form but in the case of the Example 2 the user sees the lesson content that was previously hidden for him beyond the payment form.

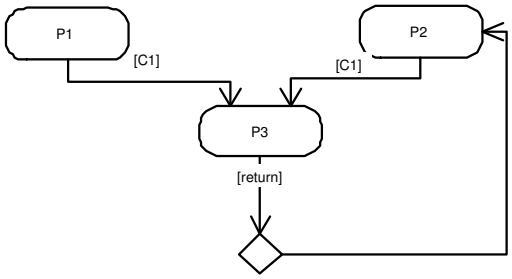

Figure 6: Using RETURN logical page.

2.4 Extension of MVC Model 2

One of the problems of the standard Model 2 frameworks is that it is complex to work with clients others than web browser. These inconveniencies appear in two places on the scheme 1:

1. when the servlet parses the http request;
2. when the servlet makes redirect to a jsp page.

The first problem is that requests coming from different client types have different request format:

- Browser clients send parameters in the form of: *param1=value1*;
- Java/applet client sends usually instantiated serialized objects via http-protocol;
- Flash client can send for example an xml structure that has to be parsed with xml parser to get the information about the request parameter.

Our decision to this problem is to introduce the *HttpRequestFactory* class that analyses the http request and creates an instance of the *HttpRequest* interface that corresponds to the client type. The instances of *HttpRequest* interface provide all the methods for accessing the initial request information. Having introduced the *HttpRequest* interface, we can be sure that the controller component may not care about the initial request format and work uniformly with *HttpRequest* instances.

Thus, to introduce a new supported client type, we need only to create a new implementation of *HttpRequest* interface and to teach the *HttpRequestFactory* to recognize this new type of request.

The second problem is that different clients accept different output formats:

- Browser client expects html code;
- Java/applet client expects to get an object from the input stream;
- Flash client expects as a rule some xml structure that contains all the output parameters.

To solve this problem, we introduce a notion of *client* into description of our logical pages (see 2.1 and listing 3). We also extend our framework by providing the adaptor classes for each client type. Now each logical page can be properly configured for the individual client (see the tag <conf>). The structure of the adaptor configuration file is out of scope of this paper.
Adaptor classes implement the *PageAdaptor* interface and are called by the controller component after the command was executed. The controller component has to deal only with the instance of *PageAdaptor* interface. The name of the adaptor class that will be really instantiated is pointed by the <adaptor> tag (listing 3). In the case of the *START_PAGE* the *HTMLAdaptor* class will be launched in the case of html client, and the *JavaAdaptor* class will be launched in the case of java client. These specific adaptors are able to display the system state in the way required by the corresponding client type (for example, the *HTMLAdaptor* makes redirect to the certain jsp page whereas the *JavaAdaptor* constructs a java object that contains all the necessary output information and writes it into the *ServletOutputStream*).

3 XML INTERFACES BETWEEN THE APPLICATION LAYERS

One more speciality of the DaMiT-System implementation is that we use the xml structures as interfaces between the application layers.

3.1 Application and Database Logic

The DaMiT system uses the database (IBM DB2 UDB) very intensively because all the data like user profiles, learning objects and lessons' structure are stored in the database.

We separated strictly the database logic from the application logic by having implemented it with stored procedures to increase the performance of database operations and not to mix Java und SQL program code to get the system better structured and easier portable between different databases.

We introduced also the following protocol between Java code and stored procedure call:

1. input parameters of the stored procedure that are simple separate values like numbers, strings are given to the procedure in normal way;
2. input parameters that are lists of database identifiers are given to the procedure as a comma separated list of values;
3. output parameters that corresponds to one and only one value are returned as they are;
4. output parameter that could produce multiple rows (cursors) are returned as an xml-structure that is defined for each parameter separately;
5. in the most stored procedures there is always one output parameter that returns the xml-structure containing logical errors that happened while executing the stored procedure.

We benefit from this approach at least in the following:

1. We can combine multiple procedure calls in one larger procedure that returns multiple xml structures. This allows us to decreases the number of database (JDBC) calls;

2. Some xml values returned from the database can be directly given to the application view modules that can make XSLT transformations with these values and display them;
3. We give the application logic developer a possibility to work independently from the database developer: if the stored procedure is not written the business logic developer can use dummy xml structures as if they were received from the database.

3.2 Model and View

As our view component deals mostly with displaying xml structures it has sense to use xml interfaces between the model and the view components as well. It means that if our business logic generates some data (instead of getting them directly from the database, like dynamically changing menu etc.) it should generate xml structures that could be easily displayed.
Like in the case with the database logic such xml interfaces allow us to make the view modules development independent from the application logic development.

4 CONCLUSION

4.1 Summarizing

In this paper we show some achievements of the DaMiT Project in the area of the web applications building. The methods and approaches being described can be extended and used to construct different internet sites. The main results of the research and constructive work in this area are adaptation and extension of the standard J2EE Model-View-Controller implementation in:

- advanced navigation support by introducing the notion of *module* and providing a command calling stack for each module that allows to restore the state of module;
- supporting multiple web client types by introducing the notion of *logical page*;
- providing xml interfaces between the application layers that allows to develop the application layers independent from each other.

4.2 Future work

The world of the web technologies is changing very quickly. Recently Sun Microsystems has proposed its own standards for design and implementation of the web applications (JavaServer Faces). The standard tools and libraries are getting more and more comfortable for the developer. Therefore, one of the main directions of the future work is to adapt the solutions found during the DaMiT project for using them within the JavaServer Faces framework.
Another important direction is to proceed the research in the area of web based applications formalisation and to integrate further navigation schemes (Thalheim and Düsterhöft, 2001).

4.3 Acknowledgements

The author of this paper is grateful to the following scientists for given support and advices: Bernd Tschiedel (Technical university in Cottbus, Germany), Dr. Steffen Lange and Prof. Klaus P. Jantke (both German Research Center of Artificial Intelligence, Germany).

REFERENCES

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

Budi Kurniawan, N. M. (2003). *JavaServer Faces Programming*. McGraw-Hill Osborne Media.

Cavaness, C. (2002). *Programming Jakarta Struts*. O'Reilly and Associates.

Grieser, G., Lange, S., and Memmel, M. (2003). DaMiT: Ein adaptives Tutorsystem für Data Mining. In Jantke, K. P., Wittig, W. S., and Herrmann, J., editors, *Von e-Learning bis e-Payment. Das Internet als sicherer Marktplatz, Leipzig, Germany, September 24-26, Tagungsband LIT03*. Akademische Verlagsgesellschaft Aka.

Husted, T., Dumoulin, C., Franciscus, G., Winterfeldt, D., and McClanahan, C. R. (2002). *Struts in Action: Building Web Applications with the Leading Java Framework*. Manning Publications Company.

Jantke, K. P., Lange, S., Grieser, G., Grigoriev, P., Thalheim, B., and Tschiedel, B. (2004). LEARNING BY DOING AND LEARNING WHEN DOING: Dovetailing E-Learning and Decision Support with a Data Mining Tutor. In *This conference proceedings*.

Schulz-Brünken, B., Herrmann, K., and Grimm, R. (2002). Kundenrollen als Vermarktungskonzept im e-Learning. In Klaus P. Jantke, Wolfgang S. Wittig, J. H., editor, *Von e-Learning bis e-Payment. Das Internet als sicherer Marktplatz, Leipzig, Germany, September 24-26, Tagungsband LIT02*. Akademische Verlagsgesellschaft Aka.

Thalheim, B. and Düsterhöft, A. (2001). SiteLang: Conceptual Modeling of Internet. In *SitesConceptual Modeling - ER 2001, 20th International Conference on Conceptual Modeling, Yokohama, Japan, November 27-30, Proceedings*, volume 2224. Springer.