Patterns, XML and MDV platform, a case study

Marko Niinimäki[1], Miika Tuisku[2], Matti Heikkurinen[2]
{marko.niinimaki,miika.tuisku,matti.heikkurinen}@cern.ch
[1] Department of Computer Science, University of Tampere, Finland
[2] Helsinki Institute of Physics at CERN, Geneva, Switzerland

Abstract

In this paper, we aim to study design patterns in the context of a software package called MDV (MetaData Visualisation). The development of this software was relatively fast-paced and performed by a team split into two physical locations for large part of the most active development period. The original design documents did not contain any overt use of design pattern methodology and all of the developers were not familiar with design patterns. We believe that analysing the design of such a software, starting from the original forces that drove the requirement specification, can provide a valuable insight into the general nature of patterns. We will also discuss ways to measure if the quality and productivity of further development of this kind of project could be improved by using pattern-based analysis of the existing software.

1 Introduction

From the user's perspective, MDV is a system for managing data in the web environment. Since not all of the data formats are easily previewed (e.g. large text documents, audio files or video-clips), the system aids the user by providing a mechanism to store descriptive data (metadata) about the data files. In order to help retrieval and organization of the data, the system also allows each document[1] to belong to one or more categories. Relevant data can be searched either based on the metadata fields or by selecting categories or an intersection of several categories. MDV platform features (i) a data access to various types of storage systems

[1] An entity containing one or more data files and the metadata related to them.

(ii) modular user interface (based on the Cocoon framework), and (iii) extensive use of eXtensible Markup Language (XML).

In its basic configuration, MDV works as a simple document management system with user and group authorisation and authentication, configurable document types, document operations (adding, editing, adding files, deleting), and grouping documents into different categories. The original forces for the development of the concept were the following:

- Aim to integrate several distinct data sources in a distributed environment with no centralised control;
- Have standardised interfaces between the major software components for maintenability and flexibility reasons;
- Support for output formats other than HTML in order to allow easy integration to other systems.

It was deemed that the best match for these requirements and the available implementation techniques would be provided by some kind of XML framework. XML was seen as a solution solving some of the problems associated with integrating data from different sources and at the same time providing a standard interface between the program components. Furthermore, due to the strong developer interest in the XML, it was assumed that this solution would not limit the generality and applicability of the software in its intended environment.

Many web applications share similar types of components, such as access rights of individual users and groups, session management, distribution and integration with legacy systems, to name just a few. Multiple solutions, ranging from programming language modules (like in J2EE, see e.g. [MH00]) to software platforms or frameworks (like Expresso, see [Jco02]) have been proposed to provide readily applicable solutions for application developers. The main benefit of XML as a data representation language is that it makes the data easier to migrate from one computer system to another [Gol01]. A wide range of XML tools is available as well. XML has been used in multiple applications and increasingly as a formalism to transfer data between applications, which can reside across organizational boundaries (e.g. in case of the RosettaNet).

While the design of these software components and their interfaces has undoubtedly been influenced by the design patterns, they usually do not form coherent pattern languages or systematically use patterns in their documentation. Software design patterns offer a more conceptual toolkit to software design. According to [BMR+96], "a pattern is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts." The goal of this paper is to try to find further support for this recurrent and universal nature of these abstract,

portable recipes (patterns) to portable data by analysing a software with only random contact with the conscious pattern application.

We approach the goal by using the MDV XML presentation framework as an example. MDV consists of clearly separable layers of user interface, business logic (data flow and state transformations), domain model and storage interfaces. As explained in more detail below, in a basic MDV installation, the main components are as follows. (i) DataAccess java class communicates with (ii) the storage interface, a relational data base by default. (iii) Abstract java classes of Documents (and their constituents), Categories (collections of documents) and Principals (users and groups) are used as a domain model (see Fig. 1). The Cocoon XML publishing framework ([Maz02]) is used to control the data flow and state transformations, and finally eXtensible Stylesheet Language Transformations (XSLTs) are used to create user interfaces.

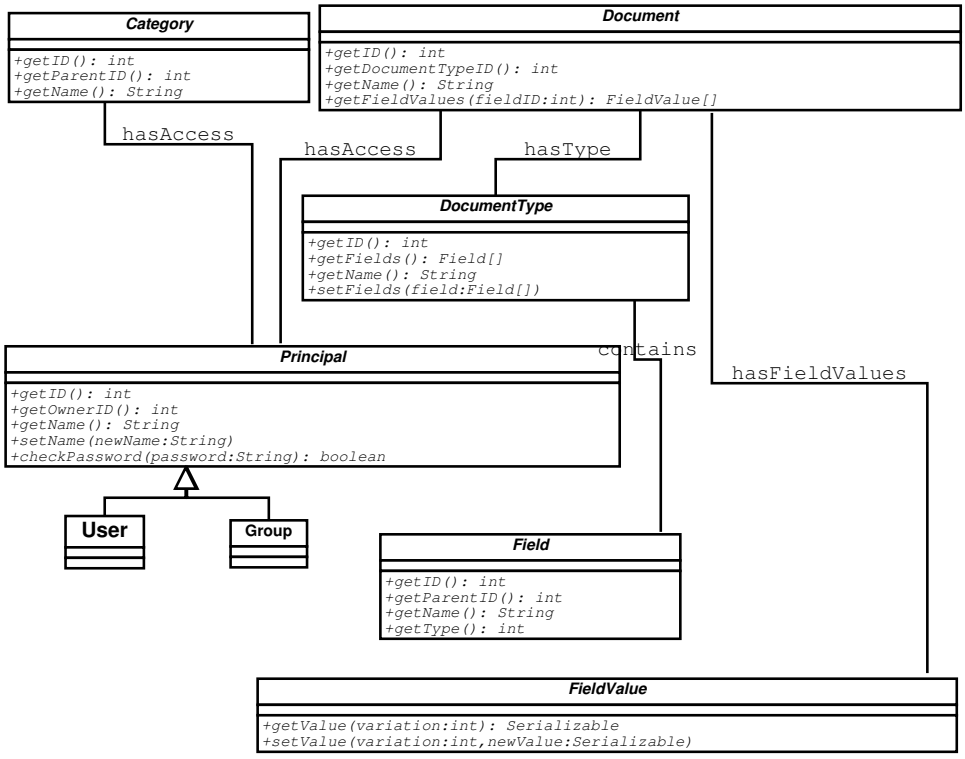


Figure 1: Domain objects (domain model).

This paper is organised as follows: in Section 2, we explain the basic terminology used in the context of patterns and "modern" software development tools and methods. In Section 3 we study how they are related to MDV software's architecture and how they help MDV obtain the required features (distinct data sources,

standardised interfaces and flexible output). In Section 4, we give an account of how the architecture can be understood through well-known patterns and what role these patterns have in the architecture. Finally, in Section 5, we see how the patterns help with redesigning the software and extending its capabilities.

2 Concepts, building blocks and terminology

A design pattern in software development is supposed to have four essential elements [GHJV94]: the pattern name (a handle we can use to describe a design problem, its solutions, and consequences); the problem (when to apply the pattern); the solution (the elements that make up the design, their relationships, responsibilities and collaborations) and the consequences (results and trade-offs of applying the pattern).

XML (eXtensible Markup Language) is "the universal format for structured documents and data on the Web" [Con02]. As such, XML is a meta language – a language for describing other languages – which lets one design customised markup languages for different types of documents, using a declaration syntax defined in recommendation [Con00]. A particular XML language (like the one we use to describe documents in MDV) conforms to a grammar, that can be defined either using Document Type Definitions (DTD's) or XMLSchemas (see [Con02]).

According to the terminological conventions, an XML document consists of elements that are either entities or attributes (of entities). An XML processor (a computer program) can parse an XML document for as long as these conventions (and some further constraints) are not violated – even if there is no DTD for this document. The role of a DTD is to define valid element type names and attribute type names and in what order they must occur in the document.

A simple DTD and a document that conforms to it are shown in Fig. 2 (adapted from [HM01]). Here, according to the DTD, in a valid `person` document, there must be one or more names and zero or more professions; a name must have attributes last and first, a profession must have attribute value; and each of the attribute values may consist of arbitrary character strings.

XSLT (eXtensible Stylesheet Language Transformations) is "a language for transforming XML documents into other XML documents" [Con99]. XSLT is based on stylesheets that filter and re-arrange the input (an XML document) and thus generate the output (another XML document). In the context of MDV, the objects that are active in the application (like the current user or the documents the user is currently manipulating) have an XML representation. This representation contains the information the user needs, but in order to make the information visible to him, the XML representation needs to be output in a form that his web

```
<?xml version="1.0"?>
<!DOCTYPE person [
  <!ELEMENT person (name+, profession*)>
  <!ATTLIST name first CDATA
                last CDATA>
  <!ATTLIST profession value CDATA>
 ]>


 <person>
    <name first="Alan" last="Turing"/>
    <profession value="computer engineer"/>
    <profession value="mathematician"/>
    <profession value="cryptographer"/>
 </person>
```

Figure 2: A simple DTD and document.

browser can understand, i.e. HTML.[2] This solution can be similar to the Interpreter pattern [GHJV94] where the DTD defines the grammar and the XSLT file encapsulates the whole "AbstractExpression" tree. The "Interpret" operation is provided by the XML framework and the context contains contains information such as the Uniform Resource Locator (URL) the expression was accessed from. With modern hardware, and especially when taking the application into account (rendering the user interface representation in the web environment) one can argue the that requirements of simple grammar and non-critical efficiency, requested by [BMR+96], are fulfilled. Together with the flexibility of the grammar and desired results of its "evaluation", this might explain why these kinds of XML frameworks have been so successful.

Apache Cocoon is a Java based XML publishing framework [Maz02]. The main design principle of Cocoon is the separation of concerns: content, logic and style. Apart from facilitating the web application design it is thought to help project management with clearly separated roles in the software construction process. In a web server environment, Cocoon needs a servlet container to be run, which can be seen as a facade pattern [GHJV94] to the system services in different operating systems. Cocoon communicates with a web server through this servlet facade, and also with data repositories (like files or data bases containing XML documents). In many modern web applications, the content is formed out of persistent data that lies in data repositories. In Cocoon, logic is implemented

[2]For an example of a XSLT file, see Section 3.

using eXtended Server Page (XSP) files, that are, in fact, XML files that contain a specific logic-element - a java program. When it comes to presentation, Cocoon can be programmed to dynamically serve different kinds of clients (web browsers, wap phones) with differing requirements on the views (HTML, WML) of the same data.

The web server used to run Cocoon, in our case, is Apache Tomcat 4 which incorporates a servlet container by default (see [Pro02a]).

3 The architecture of MDV

The MDV architecture is depicted in Fig. 3. Overall, it resembles the layers pattern of [BMR+96]. When analysing the benefits of this approach, one can immediately assume that MDV software requirements (distinct data sources, standardised interfaces and flexible output) and the layers pattern offered a good match in the following way:

- Reuse of layers will allow a large group of people to work largely independently from each other, once the initial version of the layered architecture is in place. Blackbox-type of use of services provided by the layers means that it is easier to avoid conflicting updates in version control systems.
- Support for standardisation. Especially on the level of XSLT use, it is a fairly safe assumption that no changes that would require major changes to all the Cocoon-based systems will happen.
- Modularity ("exchangeability") was one of the basic requirements in the initial MDV concept, layered approach is natural choice for having several data sources.

As far as the liabilities mentioned in the pattern, in the case of applying the MDV software following things should be noted:

- Cascades of changing behaviour. One of the critical junctions in the architecture is the interface between the data access and domain objects. Since the data repositories can be under some other organization's control, mapping the data to the domain objects can be difficult (e.g. in case of a streaming, real-time video feed - the interfaces all the way through the architecture would have to contain operations to open and close the the "channel"). Since at the moment the intended application area dictates that the data files and their metadata contain stored data, this limitation is mainly theoretical.

- Lower efficiency. The approach can cause duplicate work to be performed. In case of using the Spitfire [pro02b] software as a data repository, the software translates data from relational database to XML-format, which is in turn translated back to domain objects, which in turn are translated to XML-data for presentation layer. Frameworks like Joda [Jod02] may in the near future offer help in optimising these translations. Joda framework attempts to extend the JavaBeans interface with dynamic properties of XML data. Joda-beans based domain objects have a predefined XML-identity and even complex Joda-bean hierarchies can be serialised and de-serialised in this XML form. As such it replaces Castor style [Gro02b] Java introspection based marshallers.

- Granularity of layers. One could argue that the "middle layers" can be seen as atomic entities whereas presentation layer and infrastructure layers can contain layered third party architectures themselves.

Considering Fig. 3 in more detail, domain classes are used to model the problem domain of application, reflecting the data stored in data repositories. In the default case, the domain objects are as follows (see also Fig. 1):

- DocumentType – defines a document type that has an identity and 0..n fields.

- Field – defines a field, that is textual, numerical, enumerated, binary or user defined.

- Document – a document that has an identity and corresponds to a document type. Since the document type may have fields, the document has corresponding field values for each of the fields.

- Category – documents are classified (either automatically or by users) into categories. A category can contain zero or more documents and a document can belong to zero or more categories. Categories can be hierarchical.

- Principal – Principal is an entity as an individual user or a group that may have some permissions on the data or may be the owner of a document or category. Principals are permanently stored, so they have a permanent identity.

As a simple example, consider a category that a user whose id is 1 has created to store his holiday photographs. In a relational database (that comes with the installation package), it is stored as follows:

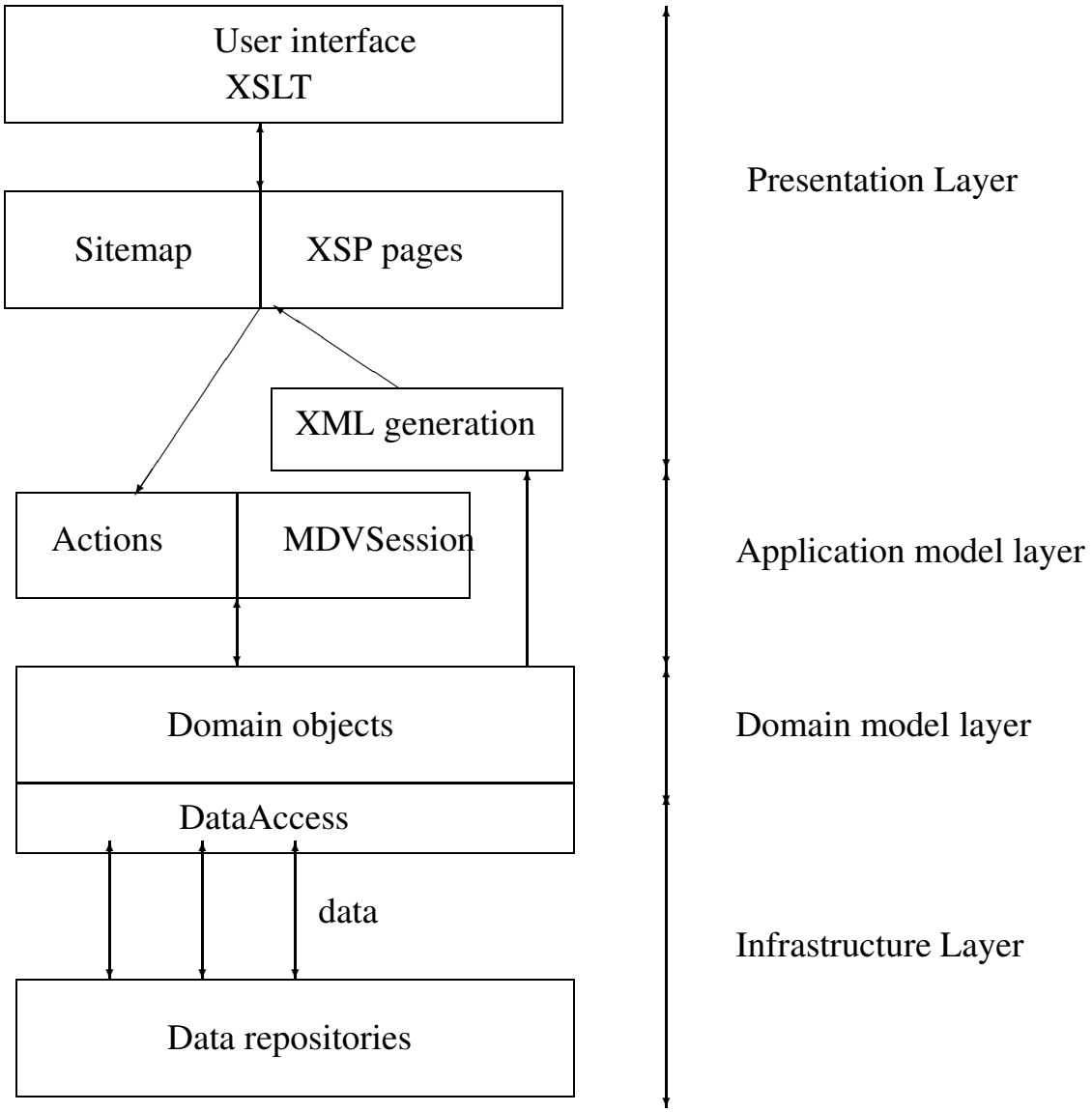
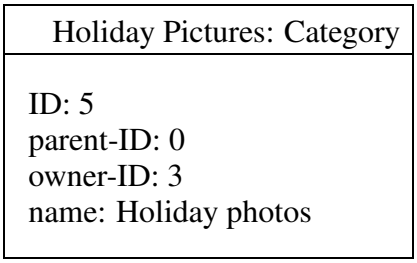


Figure 3: The architecture of MDV.

ID	parentID	ownerID	name
5	0	3	Holiday photos

There, parentID 0 defines that this category does not have a parent category and thus it is on top of the hierarchy. The corresponding fragment of a UML[3] object diagram would be:



DataAccess, a facade pattern [GHJV94], provides the upper layers a uniform Application Programming Interface (API) to heterogeneous data repositories using the domain objects and thus hiding the irrelevant details. The next example illustrates how an upper layer agent creates a DataAccess object and gets the categories using it.

```
DataAccess da = new RDBDataAccess("org.hsqldb.jdbcDriver",
                    "jdbc:hsqldb:testdb", "sa", "");
//create a new relational data base data access
da.login("admin", "sala");  // login as built-in user
Category[] cats = da.readCategories();
```

The user interface is not created out of domain objects directly, but an application model layer is used to mediate between the user interface and object domain model. The application model layer translates the messages from the user (web applications typically use string inputs) into messages understood by the domain layer. The mechanisms are hidden into a MDVSession object, which contains several parts:

- XulTree – an object model used to navigate between categories. XulTree uses the XUL language (a specific XML language), invented in the Mozilla project for XML representations. As such, it could use Mozilla's GUI-widgets as a front-end. In the case of MDV, a simple XSLT transformation is used to create a HTML view of the tree by default.

[3]Unified Modelling Language, see [Gro02a].

- SelectionManager – A tool to map string selections to domain object selections. Access to domain objects is needed to produce their XML-representation.
- Translator – A mediator for constructing Domain objects and synchronising them via DataAccess.
- DataAccess – Sometimes it is useful to a provide direct access to the domain model layer.

Each of the domain objects has an XML representation (created using Castor marshaller, see [Gro02b]). For instance, the XML representation of the example category would be:

```
<default-user-category ID="5" share-type="2" parent-iD="0" owner-iD="3">
  <name>Holiday photos</name>
</default-user-category>
```

It would be impractical, anyway, to create XML representations mechanically out of all the objects in the domain, since we do not know in advance what objects the user wants to manipulate. In order to do that, a user action must be modelled. This is done using two mutual methods: the sitemap and XSP pages.

The sitemap is a Cocoon standard for managing state changes (in practical terms, moving from one web page to another in the context of the World Wide Web). Suppose there is a form by which the user can create a new category. In an ideal situation, the user fills in the name of the category and presses the submit button on the page. The new category is then created and a new web page is shown, where the user can see his new category. However, this may fail if the user (i) is not authenticated, i.e. his session has expired, in which case he is required to log in again or (ii) the user fails to enter a name for his category, in which case his request is not processed but he is taken back to the category creation form.

A fraction of a sitemap implementing this is shown in Fig 3.

The sitemap cooperates with XSP pages that are responsible for extracting the information (from DataAccess, but in XML format) that is needed to carry out the operation the user requests. For instance, if the user wants to share his holiday photo category with another user, the information required for the operation consists of:

- information about the category;
- information about the user (i.e. if he is allowed to perform this operation);
- information (at least names and id's) of users and groups that are available, so that the user can select whom to share the category.

```
<map:match pattern="mdv-add-category">
   <!-- first validate whether user has logged in -->
   <map:act type="mdv-session-validator">
   <!--then check if the group name parameter is ok-->
   <!--(it's not a good practise to create categories without a name)-->
    <map:act type="form-validator">
     <map:parameter name="descriptor"
            value="context://protected/descriptors/params.xml"/>
     <map:parameter name="validate" value="setName"/>
     <map:act type="mdv-add-category">
        <map:redirect-to uri="protected"/>
     </map:act>
    </map:act><!--form-validator-->
    <map:redirect-to uri="CATEGORYNEW"/><!--formval fails-->
   </map:act><!--session-->
   <map:redirect-to uri="login"/><!--session fails-->
 </map:match>
```

Figure 4: A part of the sitemap.

Using the sitemap, the MDV application has been programmed so that as a response to URL "CATEGORYSHARE", XSP page category.xsp is executed, and the XML output of that page is formatted using XSLT stylesheet categorySHARE.xsl. A part of the "raw" XML output of category.xsp is shown in Fig. 3, a fraction of the XSLT stylesheet that transforms the name and id of the category into a part of an HTML form is shown in Fig. 6, and, finally, the output in the user's browser is seen in Fig. 7.

4 Patterns

As [BMR+96] states, patterns can reside in different levels of abstraction, including architectural patterns (skeletons of overall system architecture), design patterns (general issues of organization in a software) and idioms (concrete, language dependent conventions). The focus of this section lies in explaining the features of MDV from the point of view of architectural patterns and design patterns.

- Model-View-Controller pattern: Data - XSLT - sitemap. A Model-View-Controller decouples views and models to increase flexibility and reuse. In Web application development, partly because of the stateless nature of http-protocol it is less used. However, one can see the this pattern's characteristics in XML publishing. Here, the XML data is the model, XSLT is the

```
<result>
 <default-user-category ID="5" share-type="2" parent-iD="0" owner-iD="1">
  <name>Holiday photos</name>
 </default-user-category>
 <principallist>
  <user ID="3" owner-iD="1">
   <password>a2lOjPh/yI05Ltjr+B2ThQ==</password>
   <name>Administrator</name>
   <login-name>admin</login-name>
  </user>
  <group ID="2" owner-iD="1">
   <name>Administrators</name>
  </group>
  <user ID="4" owner-iD="1">
   <password>9WGq9u8L8U1CCLtGpMyzrQ==</password>
   <name>A friend</name>
   <login-name>friend</login-name>
  </user>
 </principallist>
 <currentuser>
  <user ID="3" owner-iD="1">
   <password>a2lOjPh/yI05Ltjr+B2ThQ==</password>
   <name>Administrator</name>
   <login-name>admin</login-name>
  </user>
 </currentuser>
</result>
```

Figure 5: The “raw” XML output of category sharing (generated by category.xsp).

```
Share category
 <xsl:value-of select="//result/default-user-category/name"/>
   <input type="hidden" name="id">
     <xsl:attribute name="value">
       <xsl:value-of select="//result/default-user-category/@ID"/>
      </xsl:attribute>
   </input>
to whom?
```

Figure 6: A fraction of an XSLT stylesheet.

Figure 7: The output in a browser.

view and in the case of Cocoon, the sitemap and its components are the controller. The same model (XML) is the source for different views (HTML, WML etc). Still, the Cocoon framework represents only the presentation layer in the whole architecture.

- Chain of responsibility pattern: sitemap. A chain of responsibility [GHJV94]: "[give] more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it". In order to forward the request along the chain, each object on the chain must know its successor. The notion of successor is not obvious, since the sitemap is practically an XML description. The use of SAX (Simple API to XML) parsing technique involves an event handler style mechanism which is applied to an XML structure in many ways similar to a successor list.

  It seems reasonable to identify the use of sitemap with the chain on responsibility pattern, though one finds there features that refer to the mediator pattern [GHJV94] as well.

  In practice, the extract of the sitemap in Fig. 3 illustrates the idea.

  Here, in an ideal case, the chain is as follows:

  - The request, identified as a group addition request, arrives;
  - The request is passed to mdv-session-validator. This accepts the request (the user is logged in) and passes the request forward;
  - The request is passed to a form-validator. This checks if the parameters of the request are correct and (assuming they are) passes the request forward;
  - The request is passed to mdv-add-group -handler that carries out the request;
  - Finally, the chain of responsibility is taken to "redirect-to" action by which new page is loaded.

It is easy to observe that with the help of a chain of responsibility pattern, one can implement branching as well. For instance, if the form validation fails, URL CONFIGUREGROUPADD is loaded instead of passing the request to mdv-add-group -handler.

- The observer pattern ([GHJV94]): XulTree. XulTree is widely used in Java language in Listener APIs. In MDV a listener is used to "fire" DataAccess events in depending data structures such as XulTree. Every time a document or category is created or destroyed by user, the XulTree is updated accordingly. This is accomplished by implementing the DataAccessListener Interface in MDVSession and subscribing for DataAccess events.

- The facade pattern: DataAccess. A facade "provides a unified interface to a set of interfaces in a subsystem" [GHJV94]. The default domain objects (domain model) of MDV represent various classes (see Fig. 1). Naturally, each of these classes have their own interface. DataAccess combines these for the use of client classes. DataAccess contains methods for session handling (login, logout), principal handling (reading, deleting, saving), document type handling (reading, deleting, saving) and document and category handling (reading, deleting, saving, finding documents in given categories, getting all field values of a given document). A typical use of DataAccess can be found in Fig. 8. This is a fraction of a XSP page that generates all the information needed for document manipulation (given the current user, currently selected categories and documents).

```
Session htsession = request.getSession (false);
..
MDVSession mdvSession = MDVSession.getInstance(htsession);
..
DataAccess mdvDA = mdvSession.getDataAccess();
SelectionManager mdvSM = mdvSession.getSelectionManager();
..
DocumentType[] doctypes = mdvDA.readDocumentTypes();
..
Category[] selcats = mdvSM.getSelectedCategories();
//get the cats, maybe needed when adding the new doc
//into the selected category
result = result + XMLConverter.Convert(selcats);
```

Figure 8: Use of DataAccess in document.xsp.

5 Discussion and conclusions

In general, we see a twofold role of patterns in this case study. On one hand, patterns "were there" because of the tools the developers used, and thus made some design decisions natural. This is especially true in the case of the interpreter pattern that can be seen to reside behind the XML representation paradigm. On the other hand, the developers seemed to adopt patterns from each other and promote them within the developer community.

This elementary analysis of the MDV software and its component seems to indicate strong correlation between successful software components like the Cocoon framework and the common structures presented in the basic design patterns. One could perhaps argue, that especially in the case of open-source software development, this correlation could be explained by causal factors, such as:

- If the overall structure of the software can be understood quickly by an outsider, it is much more likely that they will contribute to it. The consequences of using applicable patterns can be similar to the consequences of the simply understood code [Gab00] pattern, but on the larger scale.
- The supporting community can be much larger, since a clear logical structure makes things like division of responsibilities more intuitive. This helps to preserve the original design paradigm.
- Changes to the code are localised, thus reducing conflicts between simultaneous module changes in a multi-user version control system.

Even though researching the formation of developer communities can be difficult, a study comparing the results of pattern based introduction to the software with one based on standard framework documentation could be useful. Our research group is looking into the possibility of carrying out such a study in the context of the next version of the MDV software, which will be developed by a similarly distributed group of people. Furthermore, since the framework will be used in projects which can push the limits imposed by some of the documented liabilities of the design patterns (e.g. handling of extremely large data video data files and forming domain objects from them) we feel that a more in depth analysis of the architecture and the consequences of the patterns used is of paramount importance.

From the historical perspective it would be interesting to analyse the predecessors of XML frameworks (such as ASN.1 in CORBA) in order to determine if their relative lack of success can be explained by some of the factors presented in the Interpreter pattern. Another item to be studied is if the articulated use of patterns will help newcomers (summer students) to adopt quickly the principles of the software at hand.

References

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley & Sons, 1996.

[Con99] The World Wide Web Consortium. Xsl transformations (xslt), version 1.0, w3c recommendation 16 November, 1999. `http://www.w3.org/TR/xslt`

[Con00] The World Wide Web Consortium. Extensible markup language (xml) 1.0 (second edition), W3C recommendation 6 October, 2000. `http://www.w3.org/TR/2000/REC-xml-20001006`

[Con02] The World Wide Web Consortium. Extensible markup language (xml), 2002. `http://www.xml.com/pub/a/2002/02/13/cocoon2.html`

[Gab00] Richard Gabriel. Simply understood code, 2000. `http://c2.com/cgi/wiki?SimplyUnderstoodCode`

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[Gol01] C. F. Goldfarb. *XML Handbook*. Prentice Hall, 2001.

[Gro02a] Object Management Group. Uml resource page, 2002. `http://www.omg.org/uml/`

[Gro02b] The ExoLab Group. The castor project, 2002. `http://castor.exolab.org`

[HM01] E.R. Harold and W. S. Means. *XML in a Nutshell*. O'Reilly, 2001.

[Jco02] Jcorporate. Expresso framework project, 2002. `http://www.jcorporate.com`

[Jod02] Joda.org. Joda - the java properties framework, 2002. `http://joda.sourceforge.net`

[Maz02] Stefano Mazzocchi. Introducing cocoon 2.0, February 2002. `http://www.xml.com/pub/a/2002/02/13/cocoon2.html`

[MH00] R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, 2000.

[Pro02a] The Apache Project. Apache tomcat, 2002.
http://jakarta.apache.org/tomcat

[pro02b] The DataGrid project. Spitfire, grid enabled middleware for accessing data bases, 2002.
http://cern.ch/hep-proj-spitfire