

An Aspect-Oriented Approach to Design  
and  
Develop Hypermedia Documents

by

Ping Zhang

A thesis  
presented to the University of Waterloo  
in fulfillment of the  
thesis requirement for the degree of  
Master of Mathematics  
in  
Computer Science

Waterloo, Ontario, Canada, 2004

©Ping Zhang 2004

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Ping Zhang

## **Abstract**

Hypermedia applications can be defined as collections of interactive multimedia documents that are organized as a hypertext net. The variety of application domains and the complexity of the relationship among the application components make the design and development of these hypermedia applications a difficult process. Hypermedia design models help designers to complete their conceptualization tasks, provide a conceptual understanding of hypermedia systems and defining the relationships among system components. However, when existing document models are used, features such as logging, error handling, access control and personalized view generation are notoriously difficult to implement in a modular way. The result is that the code related to these features is tangled across a system, which leads to quality, productivity and maintenance problems. In this thesis, we provided an aspect-oriented approach to design and develop hypermedia documents. A general aspect-based document model is defined to support the separation of concerns related to the features previously described. As a result, each feature is modularized as a different aspect and, in this way, the “tangling code” problem is solved. We have also applied the general document model in a concrete case study combining DOM and AspectJ. The resulting implementation provided a new prototype dealing with many features defined as aspects such as access control, logging, view generation, annotation and dynamic event handling.

## **Acknowledgements**

It is a pleasure to thank the many people who made this thesis possible.

I would like to thank my supervisor, Paulo Alencar. Throughout my thesis-writing period, he provided encouragement, sound advice, good teaching, good company, and lots of good ideas. I would have been lost without him. My appreciation goes to my other committee members as well, to my co-supervisor, Don Cowan for the support and guidance, to Daniel Berry, for helping me finalize my thesis with valuable comments.

I would like to thank my parents, for their understanding, encouragement and endless patience when it was most required.

Finally, without the support of my wife Tao and my son Eddy, I would never succeed. Although they often had to endure my absence they seldom complained. I hope they will enjoy the effort being completed.

# Contents

<b>1 Introduction.....</b>	<b>1</b>
1.1 Motivation.....	1
1.1.1 Hypermedia applications.....	1
1.1.2 Document model.....	3
1.1.3 Document Evolution.....	4
1.1.4 Aspect-Oriented Techniques.....	5
1.2 Problem Statement.....	6
1.3 Proposed Solution.....	8
1.4 Contributions.....	9
1.5 Thesis Outline.....	9
<b>2 Related Work.....</b>	<b>11</b>
2.1 Hypertext and hypermedia applications.....	11
2.2 Document Models.....	14
2.2.1 Conceptual level modeling.....	14
2.2.2 Document model.....	15
2.2.3 Document Evolution.....	18
2.3 General Aspect-Oriented Programming Concepts.....	20
<b>3 General Aspect Based Document Model.....</b>	<b>23</b>
3.1 Static Components of the Model.....	24
3.1.1 Users.....	26
3.1.2 Contents.....	27
3.1.3 Nodes.....	28
3.1.4 Anchors.....	29
3.1.5 Links.....	31

3.1.6 Attributes .....	32
3.1.7 Location Function and Attribute Value Function.....	33
3.2 The Dynamic Behaviour of the Model.....	33
3.2.1 Operations related to users (U).....	34
3.2.2 Operations related to Nodes (N).....	36
3.2.3 Operations related to Content (C).....	39
3.2.4 Operations related to Anchor (A).....	42
3.2.5 Operations related to Link (L).....	44
3.2.6 Operations related to Attributes (B) .....	46
3.2.7 Attribute List Function (al).....	47
3.2.8 Location function (lo).....	48
3.3 The conceptual level aspects of the model.....	49
3.3.1 Assign users to different group category .....	50
3.3.2 Generate personalized view of the document.....	51
3.3.3 Simultaneous multi-user access.....	52
3.3.4 Support different level of annotation.....	53
3.3.5 Access Control and Member activity tracking .....	54
3.4 Handling operations automatically, anonymously, and consistently .....	55
3.4.1 Create components operation .....	56
3.4.2 Delete components operation .....	57
3.4.3 Change user group operation.....	58
3.4.4 Create components version.....	59
3.4.5 Delete components version.....	59
3.5 Modeling event handling with AOP Techniques .....	60
<b>4 Case Study .....</b>	<b>63</b>
4.1 AspectJ .....	63
4.1.1 Joint points.....	64
4.1.2 Selecting pointcuts.....	66
4.1.3 Signature.....	68
4.1.4 Using wildcards .....	68

4.1.5 Advice.....	69
4.1.6 Static and dynamic crosscutting .....	70
4.2 DOM Application.....	70
4.2.1 dom4j .....	70
4.2.2 Set up user groups and group categories .....	71
4.3 Example 1: Member Activity Tracking .....	73
4.3.1 Pointcut:.....	73
4.3.2 Cut through many classes and operations: .....	74
4.3.3 Advice:.....	75
4.4 Example 2: Several aspects in one application .....	76
4.4.1 Access control aspect.....	76
4.4.2 Member Activity Tracking aspect .....	78
4.4.3 View generation aspect.....	79
4.4.4 Annotation to the document and XPath.....	81
4.4.5 Capture the dynamic operation as aspect .....	82
4.4.6 More dynamic operations captured as aspect.....	85
<b>5 Conclusions.....</b>	<b>87</b>
5.1 Summary .....	87
5.2 Lessons learned .....	87
5.3 Future Work .....	89
<b>Appendix A Sample Code .....</b>	<b>91</b>

# List of Figures

Figure 4-1: ActivityTracking aspect affects classes in DOM package .....	74
Figure 4-2: Views for select the user group of Notation.....	80

# List of Tables

Table 2-1: general AOP concept .....	21
Table 3-1 : The User component.....	27
Table 3-2: The Content component.....	28
Table 3-3: The Node component.....	29
Table 3-4: The Anchor component.....	30
Table 3-5: The Link component.....	31
Table 3-6: The Attributes Component.....	32
Table 3-7: Mandatory Attributes.....	33
Table 3-8 : Dynamic operations to User.....	35
Table 3-9 : Dynamic operations to Node .....	37
Table 3-10 : Dynamic operations to Content .....	39
Table 3-11: Dynamic operations to Anchor .....	43
Table 3-12: Dynamic operations to Link.....	45
Table 3-13: Dynamic operations to Attribute.....	47
Table 3-14: Dynamic operations of attribute list function .....	47
Table 3-15: Dynamic operations of location function.....	48
Table 4-1: Join point category .....	65
Table 4-2: Primitive pointcut designator.....	66
Table 4-3: logical and Boolean operations to pointcut.....	67



# Chapter 1

## Introduction

### 1.1 Motivation

Hypermedia applications can be defined as collections of interactive multimedia documents that are organized as a hypertext net. Unlike the typical printed linear text, which is read sequentially from beginning to end, hypertext is inherently nonlinear: it is comprised of many interlinked chunks of text. Readers are not bound to a particular sequence to navigate through the text, but can browse through information intuitively by association, following their interests by following a highlighted keyword or phrase in the text to bring up another, associated piece of text.

Hypermedia is the generalization of hypertext to include other kinds of media: images, audio clips and video clips are typically supported in addition to text. Individual chunks of information are usually referred to as documents or nodes, and the connections between them as links or hyperlinks.

Hypertext is a term invented by Theodore Nelson to indicate a non-linear writing mode in which users follow multiple links and associative paths through a constantly expanding online library of textual documents to which users themselves may contribute [Mau96].

#### 1.1.1 Hypermedia applications

A wide variety of application domains can be implemented as hypermedia applications. There are four golden rules for determining whether an application is suitable for hypermedia [Niel95]:

- A large body of information is organized into numerous fragments.
- The fragments relate to each other.
- The user only needs a small fraction at any time.
- The application is computer-based.

Given these requirements, hypermedia applications can be applied to domains such as:

- Web applications: The World Wide Web has become the most common hypermedia platform. Web application ranges from simple personal home sites that are composed of a few pages to corporate web sites that are composed of millions of pages.
- Help and documentation: The MS-Windows provided Help system is a classic application of hypertext. Typically, the various command and function descriptions making up online documentation contain many cross-references, links to more detailed information, links to examples, and so forth. Large software system need detailed documentation to store the design decisions made in different design stages of the system, the internal relations among different part of the system, the changes made to the system on different versions, etc.
- Reference works: Electronic dictionaries and encyclopedias are composed of fragmented chunks of material with a high degree of cross-referencing and are hence ideally suited to hypermedia. (hyper card, wiki, wiki hyper card)
- Software engineering: The software engineering development cycle generates a wealth of interrelated documents. Modern CASE tools are also increasingly offering hypermedia features within program code editors, for example to bring up more detailed information about a procedure or variable.
- Interactive fiction: Writers and novelists are now exploring the use of hypertext for interactive, nonlinear fiction. Readers are allowed to explore an underlying fictional construction interactively.

The variety of application domains and the complexity of the relationships among the application components make the design and development of these hypermedia applications a difficult process, especially for large applications.

### 1.1.2 Document model

Hypermedia applications can be specified using hypermedia document models. These models help designers to complete their conceptualization tasks, therefore providing a conceptual understanding of hypermedia systems and defining the relationships among system components.

Design methodologies have been introduced to make this task easier. The creation of those existing design methodologies adopted different techniques from the software engineering and database management areas, such as methodology with Object-Oriented (OO) technique (Object-Oriented Hypermedia Design Model – OOHDM [SR95] and Enhanced Object-Relationship Model – EORM [Lan96]), methodology with Entity-Relationship approach (Hypertext Design Model – HDM [GMP95] and Relationship Management design Methodology – RMM [ISB95]), and methodology that combined OO technique and XML mark-up language (Document Object Model – DOM [W3C04]). These methodologies have similarities, such as using a data model for specifying hyper bases and a mapping of hypermedia links to conceptual and structural relationships in the application domain.

The document models mentioned above are typically composed of four different types of information, (we refer to as the Four Axes Model) [DG00]:

- Content. The main goal of the application is to present this data; hence it is the core of the application.
- Structure. Depending on the objective of the application, the content must be organized in a manner that makes sense. This structure is usually hierarchical in nature.

- **Navigation.** The structure of the application must be broken into hyper-pages, and these pages are cross-linked. The navigation determines how the structure of the application is broken into individual hyper-pages.
- **Presentation.** Once an application is broken into individual hyper-pages, each of these is composed of one or more data attributes. The presentation determines, first, which attributes are presented and which are hidden from the reader; and second, how these attribute is organized into the page.

These Four Axes deal with different concerns of the application. Many document models deal with two or more of these Four Axes separately. The separation of these axes in design and implementation has several advantages, such as the content can be presented in multiple applications; the same content can be traversed in different navigation mechanisms; and the presentation of the contents can be changed easily. The most important advantage is this separation of concerns can make document evolution and document maintenance easier.

### **1.1.3 Document Evolution**

Whenever a hypermedia application with substantial amounts of information is built, a number of important issues arise that are critical for document evolution. These issues include how to ensure that the information remains up-to-date and consistent, how to import new information into the application, how to extend the application with new functionality without changing the existing application structure. Furthermore, in some hypermedia applications, it is ideal if the application involves more than read-only browsing. The application can possess integral facilities for communication and collaboration such as annotations, structured discussion, user feedback, message passing, and collaborative authoring.

Many document models provide excellent support for the creation of the hypermedia application, but lack appropriate content management strategies for document combination based on different concerns, document use and subsequent maintenance and their interactions. The problem of composing document parts related to different document

concerns to create or maintain large hypermedia documents is still a challenge [Hac97]. In addition, research studies indicate that strategies to support active document users and maintainers are critical to the production and delivery of increasing complex hypermedia documents [ZFCC01] [CNML92]. It is necessary to extend the existing document model with the ability to deal with document evolution.

#### **1.1.4 Aspect-Oriented Techniques**

In recent years, many aspect-oriented languages and methods have been proposed in the literature to manage separation and evolution of concerns that can be spread throughout a software system [EFB01] [Kic96] [Lop03]. These languages and methods provide new abstraction and composition mechanisms to cope with crosscutting concerns, special concerns that overlap the boundaries of other concerns. These crosscutting concerns include adaptation, synchronization, persistence, context-sensitive behavior, system wide error handling, feature variations and logging, and deal with issues such as requirements involving global constraints, system properties and protocols.

Some aspects of system implementation, such as logging, error handling, standards enforcement and feature variations are notoriously difficult to implement in a modular way. The result is that code is tangled across a system and leads to quality, productivity and maintenance problems. Aspect Oriented Software Development enables the clean modularization of these crosscutting concerns. When implemented in a non-aspect-oriented fashion, the code for these concerns typically becomes spread out across entire programs. Aspect-Oriented Programming (AOP) techniques control such code-tangling and make the underlying concerns more apparent, making programs easier to develop and maintain.

There have been several approaches of AOP techniques developed for a variety of languages from a range of paradigms (OO, functional, procedural). They all share the common goal of providing an improved separation of crosscutting concerns. One of the leading AOP approaches is AspectJ [XP04] first developed at Xerox PARC, and then

integrated as one project of IBM eclipse technology. AspectJ is a seamless aspect-oriented language extension to Java.

## 1.2 Problem Statement

While the existing document models successfully support the creation of the document, there are some problems that are not well supported by traditional design techniques, such as security concerns and document evolution. There are also some problems that are trivial for small systems, but will generate a lot of complexity for large systems, such as dealing with the dynamic behaviors of the document components. In OO design methodology, event handling techniques are used to model these dynamic behaviors.

For the existing document models, the design and implementation of several concerns are not well supported. First of all, some concerns are intermixed in different objects, each object has to implement these concerns separately and repeatedly, which leads to messy code. To make things worse, when change happens, each of these implementations has to be found and updated separately. Because the implementation of these concerns is dispersed throughout the system, there is no good method to guarantee that all the places that require change are found and updated except by manually browsing through all the code. It is always a big effort to keep the application consistent. In the mean while, knowledge is lost when the application decays, it is a good idea to keep this knowledge in the system by making the implicit “evolution” operations. This means, the system automatically proceeds to find and update the relevant components when change happens, such as when a delete content operation is performed, the system captures the operation, finds and updates the links and anchors related to this content.

Hypermedia applications are becoming more and more sophisticated. As a consequence, it is more likely that these applications are not developed by a single individual. It is crucial to maintain communication between the different parts of the team. For a large organization, in

order to serve its working groups, the collaborative features that enable people to work together during the design, development and maintenance phases have to be supported.

For example, a large organization intends to develop a WWW application to describe the hierarchy of the company, its several divisions and departments, its employees and their activities. The application should also be used for facilitating project monitoring and management. Complex relationships among information items are required. Moreover, most of the content needs to be updated frequently and the application should support different views of its contents for different kinds of users (e.g. external users, employees, managers, directors).

To construct a document model considering these concerns, the following design questions should be considered:

- How to satisfy the ever-increasing needs of the designers to reuse design experience?
- Is that possible that the authors and end users can easily modify the content of the document and all have a personalized view of the document?
- When the document is modified, is it possible that the system can automatically check the related components and perform the operations necessary to keep the system consistent?
- How to generate views dynamically according to the attributes of the contents and links.
- How to achieve multiple user access and generate different personal documents according to the user's accessibility rights to the documents.

In general, these questions can be classified in two categories: first, how to separate the different concerns of the document model and deal with them; second, how to design the document model to make it easier when dealing with the document evolution.

### 1.3 Proposed Solution

A possible solution would be to extend hypermedia document models with abstractions that would allow different concerns such as document evolution and security (e.g., access control) to be modeled as different aspects of the system. Many researchers are doing research on how to separate these system concerns with aspect-oriented techniques. [AEB03] [BS03] [FGR03] [NCE03] [PFF02]. In this way, because these aspects can be dealt with separately, the system will be simplified and additional modeling elements would allow these aspects to be explicitly modeled.

The core concept of AOD (Aspect Oriented Design) is to separate the system concerns and deal with them separately. This idea complies with the divide and conquer design concept. Two major characteristics of the AOD technique are automation and anonymity. Automation means that the system will perform some operation under certain conditions without human intervention. Anonymity means that some trivial or labor-intensive operation can be performed in a way that is transparent for the designer. Aspect-oriented techniques allow us to construct higher level of abstractions and deal with different concerns separately. In the case of the security concern, for example, whenever a critical operation is performed, access control checking is performed automatically and anonymously. The designer does not have to know when the checking occurs or how it is accomplished.

In above section, several system level concerns are being pointed out, such as access control, member activity logging, personalized view generation, design knowledge reuse, and methods to make document evolution easier. In the domain of hypermedia application, these concerns can be dealt with by redefining the static components of the document model, and applying Aspect-Oriented techniques to the dynamic behaviors of the document model. Attributes are assigned to contents, links and anchors to make the operations of separating concerns feasible. In programming, the aspect-oriented extension to java – AspectJ, is used to capture dynamic operations of the model through pointcuts and reactions are defined through static constructs such as before, after and around advices.

## 1.4 Contributions

The goal of this research is to construct document model with AOD in mind. This model separates several system concerns and deals with these concerns separately. By doing this, the model is simpler and clearer. It also provides more flexibility when dealing with complex system with many features required. The designer can concentrate on the core functionality without being concerned about tedious common tasks. The model is suitable for both group editing of a document and a flexible personal view. With document evolution in mind, the model is more modularized, easy to understand and easy to use.

The work involved in attaining this goal has produced the following contributions:

- Identified and separated several system level concerns of the document model. This kind of concern is difficult to be implemented by OO technique or cross cuts many classes and causes the code to be messy and hard to maintain.
- Extended static document model based on separation of concerns.
- Defining dynamic behaviors of the document model based on Aspect-Oriented Design techniques, established relationship among the operations of different components.
- Extended document model to deal with access control and member activity logging as system concerns.
- Development of case study applying the extended document model.

## 1.5 Thesis Outline

In this thesis we construct an aspect-oriented document model which extends the existing document model by applying aspect-oriented design techniques. This model achieves separation of concerns and improves the methods when dealing with document evolution. This chapter introduces the concepts.

Chapter 2 gives the background about hypermedia application and introduces the basic components of document model, introduces a general document model we are going to extend. Then we introduce general AOP concepts. In chapter 3, will use this general features to discuss how to apply AOP technique to the document model.

Chapter 3 specifies the static components of our model and the methods to deal with system level concerns. In chapter 3, we also describe how to deal with the dynamic behavior of a document model with AOP.

Chapter 4 first introduces the major AspectJ features that we will used in our case study. Then we give a brief introduction of the Document Object Model implementation – dom4j, we will extend dom4j with aspect construct to show how to implement aspect features for dom4j with several case studies.

Finally, in chapter 5 we summarize our work and discuss future research directions.

# Chapter 2

## Related Work

In this chapter, we give brief background information about hypertext, hypermedia applications, document models and general AOP concepts.

### 2.1 Hypertext and hypermedia applications

More and more people and organization are choosing to store and deliver their documents electronically; many of them are in hypertext format, from the simple indexed view of document contents to the complex corporation web application which is composed of millions of interlinked pages. Hypertext is becoming a tool in many areas of life, and change the way we work, read, write, and learn. We give a brief history of hypertext and hypermedia in the following paragraphs, to provide a better idea of the concepts of hypertext and hypermedia, and how hypertext and hypermedia applications are envisioned by the experts in this area.

The history of hypertext as a computing phenomenon begins in 1945, with an article by Vannevar Bush in the Atlantic Monthly. Bush proposed a system called Memex, short for "memory extender." Bush envisioned the Memex as a microfilm-retrieval system, since computers were still huge devices, and were used exclusively for numeric computations, not text processing. Many of the features Bush envisioned for the Memex are very similar to our modern hypertext systems. It would have a scanner that allowed users to input their own documents. It would include the capacity for users to attach their own notes and annotations to documents and, most importantly, Bush envisioned the provision of 'trails', sets of links relating information relevant to a specific purpose, it would keep track of the trails through the materials, so that it could create webs of relationships between documents based on topic.

In 1962, Doug Engelbart's NLS (oN-Line System), which, although it was not designed as a hypertext system, had some hypertext-like features. NLS was envisioned as something closer to a word-processor than a hypertext system, but it did support a kind of developer's journal in which people working on the project stored their notes, papers, and memos and could cross-index them.

The term hypertext was coined by Ted Nelson in 1965. His (proposed) Xanadu system was the first computer system built explicitly to support hypertext. Nelson's idea for the project was to incorporate a docuverse, a universal repository for all the world's information and literature ever published. Nothing would ever be deleted, new versions of texts existing beside older versions, and to use relationships established between documents to allow users to navigate the texts. Nelson has struggled for 30 years to implement Xanadu, so far without success.

The first real working hypertext system was the Hypertext Editing System built at Brown University in 1967 under the leadership of Andries van Dam. Douglas Engelbart's NLS system (part of the Augment project), which had a number of hypertext features such as interlinked cross-references and a graphical mouse-based interface, was successfully demonstrated in 1968.

The first working hypermedia system was the Aspen Movie Map developed at the Massachusetts Institute of Technology by Andrew Lippman and his colleagues in 1978. The town of Aspen, Colorado, which has a very regular, rectangular street layout, was filmed by driving a truck through every street and taking front, rear, left and right view photographs every three meters. These were transferred to videodisc and linked to follow the street grid. Users sat in front of a vertical monitor showing the street view and a flat monitor showing an overview map and could navigate forwards, backwards and (at a junction) left and right with a joystick. Short video clips of many of the buildings in Aspen were also linked in, so users could stop and explore them.

Intermedia, developed at Brown University from 1985 to 1990, introduced the concept of link anchors and the use of a separate database for links (rather than storing them within

documents), allowing links to be bidirectional and link webs to be maintained as distinct entities. Intermedia was a multi-user system based on the client-server architecture and combined hypermedia features with information retrieval facilities such as full text search and dictionary lookup. Intermedia only ran on the Apple Macintosh under UNIX, and disappeared because of lack of funding to upgrade to the new operating system version.

The first widely available commercial hypertext product was OWL's Guide, developed out of the University of Kent's original UNIX version and released in 1986 for the Macintosh and later for the IBM PC. This system marked the transition of hypertext from a plaything of the research community to a real-world technique for actual applications.

The real breakthrough in popularity for hypermedia came in 1987, when Apple decided to bundle HyperCard free with every Macintosh. Within months, everyone seemed to be producing HyperCard 'stacks', as HyperCard documents are called. HyperCard is a frame-based, standalone hypermedia system incorporating a simple, but powerful scripting language called HyperTalk. Links do not have to be hardwired, but can be programmed in HyperTalk and computed dynamically.

The next major milestone was the World Wide Web (WWW or W3). The idea of the Internet came from Paul Baran. Baran outlined an idea for a totally distributed computer network, in which terminals were connected to one another through a network of redundant lines, so that, if part of the network was damaged, traffic could be routed around the blockage. The result of Baran's idea and much other research and development was something called ARPANET in 1969. Several computers at physically distant locations could now exchange data over the already existing telephone network. Back then, the Internet had little to do with hypertext. Documents could not be linked to one another because a convenient way of moving between documents on disparate systems had not been developed. The Internet itself was far more difficult to navigate than today, and required knowledge of Unix commands and FTP programs. The World Wide Web was invented by Tim Berners-Lee in 1989, with the first working system deployed in 1990. The real breakthrough came in 1993 when the National Center for Supercomputing Applications (NCSA) released Mosaic, a

point-and-click graphical browser for the WWW. HTML, along with the development of Web browsers designed to read it, made it possible for people to place documents online and establish relationships (links) between documents regardless of their physical location, and for those documents to be relatively easily viewed by anyone.

## 2.2 Document Models

In most areas of computer science, practice is helped by the use of models that characterize the entities involved, their interactions, and their behavior. A model provides a simplified representation, which aids the understanding and analysis of some aspect of the entities, allowing predictions about their behavior under varying conditions. From late 1980's, as the hypermedia application became more complex, document models were proposed to help the designers finalize their conceptual understanding of the application and direct the implementation of hypertext application.

### 2.2.1 Conceptual level modeling

At the conceptual level, applications are described by means of high-level primitives which specify the structural, navigational, and presentational views in a way that abstracts from any architectural issue.

Structural modeling primitives describe the types of objects that constitute the information base and their semantic relationships, without commitment to any specific mechanism for storing, retrieving, and maintaining the actual instances of such object types. Examples of notation that can be used to express structural features include the best-known conceptual data models, like the entity-relationship model [Chen76] and various object models [RBPE91].

Navigation modeling primitives express the access paths to objects of the information base and the available inter- and intra-object navigation facilities, again without committing to any specific technique for implementing access and navigation. Many techniques can be

employed for navigation modeling: data models extended with built-in navigation semantics [GPS93] [ISB95], or explicitly annotated with behavioral specifications [Kess95] [SR95], Petri nets [SF89], finite state machines [ZP92].

Presentation modeling aims at representing the visual elements of the application interfaces in a way that abstracts from the particular language and device used in the delivery. Many different techniques can be used, ranging from simple storyboard specification [MA93] to the use of software tools [My95] and formal methods. The independent specification of presentation, separate from structure and navigation, is particularly relevant in the Web context, because the final rendering of interface depends on the browser and display device, thus it may be necessary to map the same abstract presentation scheme to different designs and implementations.

### **2.2.2 Document model**

The common ancestors of many subsequent proposals are the entity relationship model [Chen76], in the database field, and the Dexter model [HS94] in the hypermedia area. The Dexter model originates from the effort to provide a uniform terminology for representing the different hypertext structuring primitives offered by hypertext construction systems; the core of the model is the ordered representation of a hypertext application at three levels: storage, within-component, and runtime levels. The storage level describes the network of nodes and links of the hypertext. The detail on the inner structure and node content is the focus of the within-component layer. The runtime level deals with the dynamics and presentation of the hypertext.

The modeling concepts available at the storage level are very basic: components describe pieces of information that constitute the hypertext, and can be either atomic or composite. Links are a special kind of component used to represent navigable paths. The Dexter model advocates many concepts, such as the distinctions among the structure, navigation, and presentation of a hypertext, whose influence has been long-lasting.

Several subsequent models are based on the Dexter model, and added more complex forms of hypertext organization and more powerful navigation primitives [GPS93]; time and multimedia synchronization features [HBV94]; and formal semantics of navigation and a structured design process [ISB95]. Among these evolutions, HDM [GPS93] and RMM [ISB95] have been particularly influential in the design of hypermedia applications. HDM integrates features of the entity relationship model and the Dexter model, to obtain a notation for expressing the main abstractions of a hypermedia application, their internal structure and navigation, and application-wide navigation requirements. Web structure is expressed by means of entities, substructured into a tree of components. Navigation can be internal to entities (along part-of links), cross-entity (along generalized links), or noncontextual (using access indexes, called collections [GPS94]). RMM (Relationship Management Methodology) evolves HDM by embedding its hypermedia design concepts into a structured methodology, splitting the development process into seven distinct steps and giving guidelines for the tasks. RMM's data model structures domain entities into slices, and organizes navigation within and across entities using associative relationships and structural links.

In hypermedia, the evolution from the creative definition of content to the content-independent organization of the structure of a hypermedia application is attributed to the work on HDM, which stresses the difference between authoring in the large, i.e., designing general structure and navigation, and authoring in the small, i.e., deciding the layout and synchronization aspects of specific component types. HDM, however, did not prescribe a formal development lifecycle, which was first advocated by RMM, where the following seven activities, i.e., entity relationship design, slice design, navigation design, conversion protocol design, interface design, behavior design, and implementation and testing are proposed. The first three activities provide a conceptualization of the hypermedia application domain in terms of entities, substructured into slices, and navigable relationships. Conversion protocol design is a technical activity that defines the transformations to be used for mapping the conceptual schema into implementation structures. In addition to defining the development lifecycle, RMM also gives guidelines for slice and navigation design, the two tasks most particular to hypermedia design.

OOHDM [SR95] takes inspiration from object-oriented modeling and simplifies the RMM lifecycle to only four steps: domain analysis, navigation design, abstract interface design, and implementation. In domain design, classical object-oriented techniques are used, instead of the entity relationship model. Navigation design adds specific classes (e.g., node, link, index) to represent different forms of navigation. The same is done for presentation, which is described by means of classes (e.g., button, text field) added during interface design. Implementation then fleshes out the classes identified during design with code in the implementation language of choice.

Labyrinth[DAP97] [DAP01] is a hypermedia-oriented model proposed to solve some problems which are hard to solve by the models mentioned above, for instance, Dexter [HS94] and Amsterdam [HBV94] do not include elements to formalize security policies to protect information from unauthorized or improper access, and OOHDM and RMM do not solve the synchronization problem. Labyrinth is platform independent, and does not deal with the storage layer of the application. The model is divided into two related parts: a static part, that includes elements to specify hypermedia applications and a dynamic part, which is made up of a number of operations that can be performed over the elements of the static part. In this paper, we will extend the Labyrinth model with AOP technique.

Document Object Model (DOM) [W3C04] takes advantage of object-oriented modeling techniques and XML, the new markup language being proposed for the World Wide Web. DOM uses the object-oriented technique to model the data structure and construct a tree substructure for the document, uses event-handling mechanism to handle the user interface and dynamic behavior of the web application. DOM uses ancillary facility such as XPointer, XLink, and XPath to specify the location within a text or XML document and the link between different documents, by way of expressing the steps necessary to get from document elements to the required destination in terms of tree-traversal steps, character counting, and other methods, XPath uses name space and Uniform Resource Identifiers (URI) to specify the distinct destination of the links. XPointer, XLink, and XPath enable many of the important feature of hypertext systems not seen previously, such as one-to-many links, "out-

of-line" or externalized links, and calculations of link positions. DOM has not been fully defined and has yet to be widely accepted.

The LivePage software combines relationship DBMS with web publishing. It is designed to control large, complex content bases (web sites and sets of documents), and publish the content to the web. LivePage Enterprise [LP04], is an integrated, multi-user, content-base management system. LivePage enables the storage, management and publication of electronic information seamlessly across intranets, the World-Wide Web and on CD-ROM. LivePage provides a solution to deliver 'live' content from a relational database. For a more formal definition of related models see [DG00].

Open Hypermedia System (OHS) enables client applications to create, edit and activate links which were managed in separate link databases; it contrasted with the traditional approach to hypertext systems in which the functionality of both data and link management was provided within a single indivisible application. Links are stored outside of the documents to which they refer, in one or more databases of links that are queried by the browser just before showing the document. This architectural choice is extremely flexible and powerful, making a lot of currently unavailable functionality possible. For instance, this enables different link sets over the same content for different audiences and tasks [CHD99]. Third-party individuals can provide links and create guided tours over documents owned by others. Individuals and groups could maintain their own personal link bases. Although technically possible, this solution is a sophisticated task that is hard to generalize. OHS are a very powerful class of systems that provide hypertext. The key characteristic of most OHS is that they enable hypertext links to be used between different applications.

### **2.2.3 Document Evolution**

Having looked at the origins and evolution of hypertext and the evolution of document model, we can see the hypertext model's evolution when modeling the content of the document. Content is mostly related to the storage layer of the system. The storage layer evolved from simple file systems, to the utilization of relational DBMS, and finally, to the

use of an OODBMS. OODBMS have enabled the evolution of the role of DBMS in hypermedia, ranging from simple persistent storage systems to complete environments for designing applications.

The dominant hypertext system at the moment is unquestionably the World Wide Web, but the Web is not the best one in terms of its actual hypertext functions. Much current research, which draws on a strong base of work over the last twenty years, is driving towards the improvement of current hypermedia systems. The XML family of recommendations (DOM) will bring changes, most importantly to hypertext within the Web. And the OHS's trend is to enable hypertext links to be used between different applications.

For the hypertext application in the future, there are several necessary functional extension to the current document model. The first task is to provide full hypertext functionalities to hypertext application, such as a flexible personal view of the document and an end-user editing ability to the original document (annotation, remarks, etc.). The second task is to preserve link and referential integrity, there are several proposals to solve this problem, including keeping multiple versions of documents or links to virtual documents which create documents as they are required. The third task is that, having created a hypertext, to maintain and update hypertexts, both links and data, wherever necessary. When changes happen, a working mechanism which automatically checks link and content consistency is needed.

Document evolution is driven by expected or unexpected changes that occur in the hypertext application. These changes can either be an extension to the functionality of the hypertext application or be the maintenance and update operations on the content of the application. From the history of hypertext, we can see this happens continuously and will happen again and again. This requires us to provide a method to consider the document evolution on the document model level. With the aspect-oriented extension to the existing document model, we can provide clean modularization of crosscutting concerns such as: error checking and handling, synchronization, context-sensitive behavior, performance optimizations, monitoring and logging and debugging support.

## 2.3 General Aspect-Oriented Programming Concepts

Separation of concerns is one of the most important concepts of software engineering. It refers to the ability to identify, encapsulate, and manipulate those parts of software that are relevant to a particular concern. Different approaches are adopted to achieve this goal. For example, the class in object-oriented programming encapsulates data concerns. Feature concerns, like printing and persistence are also common. Appropriate separation of concerns can reduce software complexity and improve comprehensibility; promote traceability; facilitate reuse, customization, and evolution; and simplify component integration.

Aspect Oriented Programming (AOP) is a new paradigm in software engineering, which is mainly based on the concept of separation of concerns. Normally, object-oriented programming (OOP) provides a good degree of separation of concerns, but there are concerns, which are properties of a system for which the implementation cannot be cleanly encapsulated as a single functional unit of the overall system [KHH01]. These concerns cross the encapsulation boundaries imposed by classes and interfaces of an OOP implementation. Such concerns are called crosscutting concerns. AOP models these crosscutting concerns as aspects. AOP is not a standalone programming paradigm, but an add-on to Object Oriented Programming.

Currently there are many different implementations of aspect-oriented languages and frameworks. Some examples for Java are AspectJ [XP04], HyperJ [IH04], and AspectWerkz [BSA04]. Most of them are in their early development stages. There are also aspect oriented programming projects for C/C++: AspectC++ [AC04] and Microsoft's .NET framework [AC03] [LN04] [WN03] among others. Most of these AOP languages are based on AspectJ, which is the first complete and powerful language extension for AOP has been created. Each implementation has its own syntax and structure which leads to a compatibility issue with different implementations of AOP.

In our work to apply AOP technique in the designing of document model, we try to define a language independent definition of the model and we do not want to constrain our model to specific language or system implementation. So, we will use the common concept which is

shared by all of the implementations to describe the aspects in our document model. In describing the aspect, the common concepts shared by all of the implementations are: aspect, join points, pointcuts and related advices.

**Table 2-1: general AOP concept**

AOP concept	description
concern	An abstract representation of a need or problem to be addressed in a system or application.
joint point	Any well defined point in the execution of a program which can be captured and special operations related to the aspect should be performed.
pointcut	A programming element that picks up the join point. Capture the call and / or the execution of the methods of the object.
before advice	Advice that excutes before the join point.
after advice	Advice that excutes after the join point.
around advice	Advice that excutes instead of the join point.

Aspects can be defined as modular units that implement a crosscutting concern. A concern is an abstract representation of a need or problem to be addressed in a system or application. Aspects are defined using join points and associated additional implementation information called advice. A Join point is any well defined point in the execution of a program. It can be a call for a method or constructor, a value being read from or written to a variable, among other things. Pointcut is a structure that allows an advice to connect to a given join point. Simply put, it is a “hook” to the object being affected by the aspect. An advice is used to describe what should be done and in what order some extra code should be added when that execution point is reached. There are three kind of advices: before advice, after advice, and

around advice. The before (after) advice executes before (after) the join point. The around advice executes in place of the join point. Table 2-1 summarizes the general AOP concept.

Different programming language implementations have different syntax to describe the aspect. In this paper, we use the following schema:

Aspect A

Pointcut P : description of the method call/execution captured

Before P: description of the action performed

After P: description of the action performed

Around P: description of the action performed

Where A is the name of the aspect and P is the name of the pointcut.

## Chapter 3

# General Aspect Based Document Model

In this chapter, we specify the static components of our extended model and specify the behaviors of the static components in hypermedia applications using our model. The dynamic behavior is made up of a number of operations that can be performed over the elements of the static components. First, we will try to deal with some system level concerns which will affect the whole structure of the document design, such as security concern. Then, we will try to capture the dynamic behavior and use AOP technique to make these operations simpler and consistent. Meanwhile, we will try to test the possibilities of replacing the event handling mechanisms, which are employed by many object oriented document models, using AOP techniques. The event handling mechanism is an OO design technique that enables the object to react with a predefined behavior when certain event happens or certain conditions are met.

As described in chapter 1, there are four apparently orthogonal axes to compose a document model. We try to separate our model's components according to these axes to achieve separation of concerns. We first divide the model components into several groups according to functionality:

- Content – store the information that the document provided.
- Navigation – define the relationship among contents and nodes through links and anchors.
- Structure and Presentation – define how the contents are put together and how the contents are presented to the viewer through node and view.

### 3.1 Static Components of the Model

This model represents a hypermedia application by means of a HyperDocument (HD) where a number of elements are specified to define the structure and behavior of the application. In addition, each user and group of users can have a personalized HD that constitute private spaces, only accessible by their owners, where users can create new components according to their own requirements by associating their user information with the components of the HD. In this way, the general HD view is not changed, but special users can have additional personalized extensions to the document when they generate their personalized view.

In this model, a HyperDocument (HD) is defined as:

$$HD = (U, N, C, A, L, B, lo, al)$$

Where “U”, “N”, “C”, “A”, “L” and “B” are respectively sets of Users, Nodes, Contents, Anchors, Links and attributes, “lo” and “al” are functions. “lo” determines the location of a content into a node, “al” get the list of attributes of a component.

The set of Users represent users and groups of users, the users are assigned to different user groups and the user groups are categorized to different level of accessibility to the document, some users can possess their own personalized components only visible to their owners.

We use two different elements to represent structure and information items: node and content. A node is an abstract information holder that can contain any type and number of information items. The node represents the smallest information retrieval unit of HD from the user's perspective (e.g., a page of an electronic book). Content can not be retrieved without being presented in a node.

This separation between structure and content provides a number of benefits, including cleaner specifications and the ability to share content by reference. But, it does not provide means to represent the inherent structure of most hypermedia applications. To model complex structures, composition mechanisms are required. In particular, two abstraction mechanisms are quite common in hypermedia applications: aggregation and generalization.

Aggregation allows different elements to be referred to by means of a single composite element. For instance, a table of contents of an electronic book aggregates all of its chapters. Aggregated elements remain independent and do not share properties. Generalization defines a composite element whose components will inherit all its properties. For instance, each page of an electronic book can be generalized by means of a generic page where common presentation features are held. Elements sharing a number of characteristics and/or behaviors can be grouped by means of this abstraction mechanism. Our model supports both abstractions not only to define composite nodes but also composite contents, so that design solutions to complex problems can be specified.

Links are used to make the HD connected, they can be embedded into different types of information and their activation can produce a variety of results. For these reasons, the link definition has been complemented with attributes. In our model, there are several types of links (aggregation, generalization, referential, and version). Links can have associated attributes. Mandatory attributes include those of the nodes and two additional ones: LinkType and Direction. When a new link is created, the LinkType attribute is assigned to it, this value is used to discriminate among three different types of links:

- referential links, that define navigation paths;
- aggregation and generalization links, that establish a hierarchical relation between composites and their components;
- version links that sequentially connect different versions of node or content.

Likewise, the “Direction” attribute is associated with each new link and it is used to establish whether the link is unidirectional or bidirectional. Selecting a link can give rise to a number of reactions, including navigating to a different node or opening an external application or program.

The concept of anchor was first introduced by the Dexter model to represent the source and target of a link. A link is defined between two sets of anchors, sources, and targets,

respectively. In our model, links and anchors are associated with users; the result of activating a link can be different depending on the user who selected it.

This section focuses on the description of the static components of the model, in the next section we discuss how the model provides operations related to the use and management of the HD. Sets of Users, Nodes, Contents, Anchors, Links and Attributes can be accessed and updated to satisfy the users' requests. For example, users can define versions of nodes and contents. A new version includes all the elements associated with the node or content (i.e. contents, anchors, links attributes and events), it is automatically connected to the previous and next version by means of a special type of link (called version link) and it can be modified. Operations concerning the model functions ("lo" and "al") allow the assignment and modification values of the components and to get information about function values. Each operation includes a security checking process to determine whether the user involved can or not perform the operation. For instance, operations related to the ability to modify the information should only be permitted to a special user responsible for editing the HD and access controls are dealt with separately in the access control aspect.

The components of the HD are specified and explained bellow.

### 3.1.1 Users

USERS:  $U_i = (\text{UserId}_i, \text{UserGroup}_i, \text{GroupCategory}_i)$

Users are identified by a distinct alphanumeric string "UserId". Every user belongs to a user group. The system can hold many user groups, and these user groups are categorized. Each user group is assigned to a category which determined what the user in this group can do to the document and its components. The user group attribute also determines what kind of general view the document will display to the user in this group. When a "user" with the appropriate privileges makes some personalized modification to some components of the document, such components of the document will be identified by the "UserId" of the "user". See the following description of Content, Node, Link and Anchor.

**Table 3-1: The User component**

	$U = \{U_i \mid i = 0, \dots, n, n \in \mathbb{N}, \text{ where } U_i = (\text{UserId}_i, \text{UserGroup}_i, \text{GroupCategory}_i)\}$ Each user belongs to certain user group, and user groups are categorized.
	UserId <sub>i</sub> : User's unique identifier $\text{UserId}_i \in \{\text{alphanumeric strings}\}$
	UserGroup <sub>i</sub> : set of users who belong to the same access category $\text{UserGroup}_i = \{\text{UserId}_i \mid i = 0, \dots, m, m \in \mathbb{N}\}$
Users	GroupCategory <sub>i</sub> : determines the permitted operations the group of users can perform on the components of the document. $\text{GroupCategory}_i \in \{\text{browsing, notation, editing}\}$ The browsing user group can only browse the document The notation user group can add notation to the document but can not make other change to the document The editing user group can make modification to the document and make any necessary update to the document.

### 3.1.2 Contents

CONTENTS:  $C_i = (\text{ContentId}_i, \text{UserList}_i, \text{UserGroup}_i, \text{ContentType}_i)$

A content is a piece of information identified by a distinct alphanumeric string "ContentId". The "UserList" hold the set of users who possess the personalized view of this content and this content can be edited by the users in this list. If the "UserList" is empty, which means no user holds personalized view to this content, the "UserGroup" determines operational category of this content. To describe which kind of information the "content" is made up of, "Type" and "Units" of the "ContentType" parameter are used. Every kind of information is

allowed (text, graphic, video, program, sound, etc.) but the dimension units (pixels, characters, frames, etc.) must be specified. “Content” can be placed in any position within a node by means of the “location function” (see description of “lo” in section 3.1.7).

**Table 3-2: The Content component**

Content	$C = \{C_i \mid i = 0, \dots, n, n \in \mathbb{N},$ where $C_i = (\text{ContentId}_i, \text{UserList}_i, \text{UserGroup}_i, \text{ContentType}_i)\}$
	ContentId <sub>i</sub> : content unique identifier $\text{ContentId}_i \in \{\text{alphanumeric strings}\}$
	UserList <sub>i</sub> : List of users who make personal editing to this content and users in this list can generate personalized view of this content $\text{UserList}_i = \{\text{UserId}_i \mid i = 0, \dots, m, m \in \mathbb{N} \}$
	UserGroup <sub>i</sub> : Determines which group of users can access this content
	ContentType <sub>i</sub> : determines the type and dimensions of the content $\text{ContentType}_i = \{\text{Type}_i, \text{Units}_i\}$ $\text{Type}_i \in \{\text{text, video, program, picture, ...}\}$ $\text{Units}_i = \{x_{ij}\}, x_{ij} \in \{\text{pixels, characters, sentences, ...}\}, j = 0, 1, \dots, m, m \in \mathbb{N}$

### 3.1.3 Nodes

NODES:  $N_i = (\text{NodeId}_i, \text{UserList}_i, \text{UserGroup}_i)$

The “node” is a container of information defined by a distinct alphanumeric string identifier “NodeId”. The “UserList” hold the set of users who make the personalized modification to this node and this node can be viewed and edited only by the users in this list. If the “UserList” is empty, which means no user holds a personalized view of this node, the “UserGroup” determines the operational category of this node. The content of the “node” is

defined in an independent manner (see description of “content”) and no restrictions are imposed on the nature of the latter (e.g. text, graphic, sound, video ...). Composite “nodes” are logical structures made up of “nodes” (and other composite nodes) that embody the abstraction mechanisms: aggregation and generalization which are defined by [Garg88]. Aggregation is the mechanism by which a collection of objects can be referenced by an identifier. A generalization is the abstraction by which a collection of objects is referenced by a generic object which captures the essential similarity between the objects. Composite “nodes” are also defined as “nodes” and the abstraction mechanisms are defined by means of the links (see description of “link”).

**Table 3-3: The Node component**

Node	$N = \{N_i \mid i = 0, \dots, n, n \in \mathbb{N}, \text{ where } N_i = (\text{NodeId}_i, \text{UserList}_i, \text{UserGroup}_i)\}$ <p>Each node is an abstract object to place a content or anchor at a precise moment and / or location</p>
	<p>NodeId<sub>i</sub>: node unique identifier</p> <p style="text-align: center;"><math>\text{NodeId}_i \in \{\text{alphanumeric strings}\}</math></p>
	<p>UserList<sub>i</sub>: List of users who make personal editing to this node and users in this list can generate personalized view of this node.</p> <p style="text-align: center;"><math>\text{UserList}_i = \{\text{UserId}_i \mid i = 0, \dots, m, m \in \mathbb{N}\}</math></p>
	<p>UserGroup<sub>i</sub>: Determines which group of users can access this node</p>

**3.1.4 Anchors**

ANCHORS:  $A_i = (\text{AnchorId}_i, \text{NodeId}_i, \text{ContentId}_i, \text{UserList}_i, \text{UserGroup}_i, \text{AnchorPos}_i)$

An “anchor” is defined by a distinct alphanumeric string identifier “AnchorId” and it determines a reference locus into a node and/or content. These latter are identified by using the “NodeId” and the “ContentId” parameters respectively. The “UserList” hold the set of

users who make the personalized modification to this anchor and this anchor can provide the start point for personalized links led to personalized nodes or contents when the matched personalized links have match “UserId” in its “UserList”. The personalized anchor can be viewed and edited only by the users in this list. If the “UserList” is empty, which means no user holds a personalized view to this anchor, the “UserGroup” determines operational category of this anchor. “Position” and “Extension” in “AnchorPos”, specifies the location and extension of the anchor.

**Table 3-4: The Anchor component**

Anchor	$A = \{A_i \mid i = 0, \dots, n, n \in \mathbb{N}, \text{ where } A_i = (\text{AnchorId}_i, \text{NodeId}_i, \text{ContentId}_i, \text{UserList}_i, \text{UserGroup}_i, \text{AnchorPos}_i)\}$
	<p>AnchorId<sub>i</sub>: anchor unique identifier</p> <p style="text-align: center;"><math>\text{AnchorId}_i \in \{\text{alphanumeric strings}\}</math></p>
	<p>NodeId<sub>i</sub>: node in which the anchor is embedded</p> <p style="text-align: center;"><math>\text{NodeId}_i = -1 \cup \{\text{NodeId}_j \mid \exists N_j \in \mathbb{N}, \text{ NodeId}_j \in N_j, j = 0, 1, \dots, p, p \in \mathbb{N}\}</math></p> <p style="text-align: center;">NodeId<sub>i</sub> = -1 anchor is tied to a content</p>
	<p>ContentId<sub>i</sub>: content in which the anchor is embedded</p> <p style="text-align: center;"><math>\text{ContentId}_i = -1 \cup \{\text{ContentId}_j \mid \exists C_j \in \mathbb{C}, \text{ ContentId}_j \in C_j, j = 0, 1, \dots, q, q \in \mathbb{N}\}</math></p> <p style="text-align: center;">ContentId<sub>i</sub> = -1 anchor is tied to the whole node</p> <p style="text-align: center;">If both NodeId<sub>i</sub> and ContentId<sub>i</sub> are -1, the anchor is referred to the content only when the content is associated to that node.</p>
	<p>UserList<sub>i</sub>: List of users who make personal editing to this anchor and users in this list can make personalized access to this anchor.</p> <p style="text-align: center;"><math>\text{UserList}_i = \{\text{UserId}_i \mid i = 0, \dots, m, m \in \mathbb{N}\}</math></p>
	<p>UserGroup<sub>i</sub>: Determines which group of users can access this anchor</p>

	<p>AnchorPos<sub>i</sub>: location and extension of the anchor in the node or the content</p> $\text{AnchorPos}_i = \{ \text{Position}_i, \text{Extension}_i \}$ <p>Position<sub>i</sub>: Location of the initial point of the anchor in the node or content</p> <p>Extension<sub>i</sub>: Extension of the anchor into the node or content</p> <p>Position<sub>i</sub>, Extension<sub>i</sub> take values in units of the node and content dimensions</p>
--	--

**3.1.5 Links**

Navigation using links is an essential function of hypermedia systems. Links are usually identified with hot words and icons that users can click to move onto a different node.

LINKS:  $L_i = (\text{LinkId}_i, \text{LinkStart}_i, \text{LinkTarget}_i, \text{UserList}_i, \text{UserGroup}_i)$

**Table 3-5: The Link component**

Link	<p><math>L = \{L_i \mid i = 0, \dots, n, n \in \mathbb{N}, \text{ where } L_i = (\text{LinkId}_i, \text{LinkStart}_i, \text{LinkTarget}_i, \text{UserList}_i, \text{UserGroup}_i)\}</math></p>
	<p>LinkId<sub>i</sub>: Link unique identifier</p> $\text{LinkId}_i \in \{\text{alphanumeric strings}\}$
	<p>LinkStart<sub>i</sub>: set of anchors defining the starting points of the link</p> $\text{LinkStart}_i = \{\text{AnchorId}_{ij} \mid \exists A_j \in A, \text{AnchorId}_{ij} \in A_j, j = 0, 1, \dots, p, p \in \mathbb{N}\}$
	<p>LinkTarget<sub>i</sub>: set of anchors defining the targets of the link</p> $\text{LinkTarget}_i = \{\text{AnchorId}_{ij} \mid \exists A_j \in A, \text{AnchorId}_{ij} \in A_j, j = 0, 1, \dots, q, q \in \mathbb{N}\}$
	<p>UserList<sub>i</sub>: List of users who make personal editing to this link</p> $\text{UserList}_i = \{\text{UserId}_i \mid i = 0, \dots, m, m \in \mathbb{N}\}$
	<p>UserGroup<sub>i</sub>: Determines which group of users can access this link</p>

A “link” is a uni- or bi-directional labeled connection between two sets of points, sources and targets. In a hypermedia application, “LinkStart” and LinkTarget” parameters are anchors which are considered as sources and targets. Four types of links are included in this model: Referential, Aggregation, Generalization and Version (see description of “Attributes”). The Referential type is used to represent arbitrary connections between elements (nodes or contents). The Aggregation and Generalization types are used to create the abstractions to form composite nodes. The “Version” type has been included to link together successive versions of the document.

### 3.1.6 Attributes

ATTRIBUTES:  $B_i = (\text{AttributeName}_i, \text{Value}_i)$

**Table 3-6: The Attributes Component**

Attributes	$B = \{B_i \mid i = 0, \dots, n, n \in \mathbb{N}, \text{ where } B_i = (\text{AttributeName}_i, \text{Value}_i)\}$
	<p>AttributeName<sub>i</sub>: Attribute unique identifier</p> <p>AttributeName<sub>i</sub> ∈ {alphanumeric strings}</p>
	<p>Value<sub>i</sub>: Values assigned to the attribute</p> <p>Value<sub>i</sub> ∈ {alphanumeric strings}</p>

“Attributes” are properties that can be associated with users, nodes, contents and links through the “attribute list function” (see description of “al”). An “attribute” is represented by a “Value” assigned by default during the definition phase and modifiable by means of the “al” function. There are no restrictions in the number of “attributes” for each element of the model although each user, node, content and link must have at least those attribute summarized in Table 3.7. “Label” is a name or a description containing semantic information of the element to which it is tied. “author” indicates the user who created the element.

“LinkType” is a tag used to discriminate between the Referential, Version, Generalization or Aggregation links. “Direction” differentiates between uni- and bi-directional links.

**Table 3-7: Mandatory Attributes**

	Label	Author	LinkType	Direction
User	X			
Node	X	X		
Content	X	X		
Link	X	X	X	X

### 3.1.7 Location Function and Attribute Value Function

LOCATION FUNCTION:  $lo(C_i, N_i) = \{Position_i, Time_i\}$

The location function “lo” places content into a node. Within the node, the content is located at “Positon” at time “Start Time” and it remains in that place for “Duration” time units.

ATTRIBUTE VALUE FUNCTION:  $al(x) = \{AttributeName_i, Value_i\}$

The function “al” assigns a value to an attribute, tying it to a user, node, content or link.

## 3.2 The Dynamic Behaviour of the Model

A HD is a fully connected hypermedia application made up of nodes connected through links, in such a way that each node is connected to a “hub” node. The “hub” node plays an important role in the HD definition since it represents the HD itself and contains information concerning the entire application. In our model, we not only consider the HD as a whole, but also put emphasis on the components of the model, as described in section 3.1, as we add user information to content, node, link and anchor components. This way, we can make the document more personalized by generating different personalized views and defining different navigation paths according to the viewer group category, such as external users,

internal users and internal editors. We can specify these browsing group categories, and associate links with different categories, then when the group view is generated, only the links with the matched group category are shown, and are activated, the users belonging to different groups can navigate the document through different paths and view different content, if the content is also associated with the group category.

The dynamic part of the model includes operations that allow the components of the HD to be created, removed and modified. Each dynamic operation of the model is checked by the access control aspect to decide whether the operation can be performed. If this security checking does not succeed, the operation is not performed. Each operation is treated as a transaction that is only executed if all its actions are considered safe. In the mean time, the member activity tracking aspect will record any modifications to the document made by the editor; this tracking can be extended to other users such as author easily by capturing relevant operations performed by the authors. In this section, we define the dynamic behavior of these components.

The dynamic behavior makes all modifications to the document. We call this the mutation operation. The users in the browsing group can not make modification to the document. So the following mutation operation can only be performed by the users of the notation and editing group. Generally, the users in the notation group can only make modifications to their own components, in other words, these users can only modify the components if their userid is in the UserList of these components. The editor from the editing user group can change the components to different user group, and make the modifications to all the public and personalized components.

### 3.2.1 Operations related to users (U)

USERS:  $U_i = (\text{UserId}_i, \text{UserGroup}_i, \text{GroupCategory}_i)$

The dynamic operation concerning users includes operations to create and delete users; create and delete UserGroups; modify user group categories. Because different users have different levels of accessibility to the document, operations that may compromise the general security

policy of the application (e.g., creation of users and user group or modification of their group category) should be available only to a special user responsible for the security of the system. This security manager can be implemented as special sub-group of the editing user group category, which is a very restricted group of users that should not include all editors.

**Table 3-8: Dynamic operations related to User**

Operation name	Description of the operation
createUser	Assign a unique id to a user and assign the user to a UserGroup. The user group should be already defined.
deleteUser	Delete a user. Check user group category, if the user group is notation, find all the personalized components belongs to this user, when the components belong to a group of user, remove the userid from the components' UserList, delete the components which belong only to this user.
createUserGroup	It creates a user group and assigns it a group category.
deleteUserGroup	Delete a UserGroup, all the users belong to this group will be deleted too.
ModifyUsergroup	Change user group, delete user from one group and add user to another user group.

We can see from the table above, special attention should be paid to the delete user operation, we can catch the operation of delete user of the notation group and do the required additional operation, and when the user of other group is deleted, no special operation is performed.

*Aspect deleteUser*

*pointcut: deleteUser && (user belongs to notation group)*

*before: find components (content, node, link and anchor) with matched userid in their UserList*

*after: remove userid from matched components' UserList*

*If matched components' UserList is empty after remove userid, delete it.*

### 3.2.2 Operations related to Nodes (N)

NODES:  $N_i = (\text{NodeId}_i, \text{UserList}_i, \text{UserGroup}_i)$

As described in section 3.1, a node ( $N_i$ ) has a unique identifier (NodeId), which, in the system implementation, can be resolved to an internal value automatically assigned by a centralized system or to a URI (Uniform Resource Identifier) referring to external resources of a distributed environment. Each node belongs to a user group, and it may also belong to a special user or several users of this group, which makes it a personalized node, the special user's userid is shown in the UserList. Only users belong to the right user group can access and/or modify the node. This establishes the permission type of operation that the user can perform on the node and its components (contents, links, attributes). Users belonging to different UserGroup are permitted visiting, adding personalization, and updating node operations, respectively.

Contents and anchors can be placed into nodes at a precise moment and/or location. Moreover, any number of attributes that enrich the node definition can be assigned to a node by means of the  $al$  function. In particular, nodes have two mandatory attributes automatically assigned whenever a node is created: "Label" that adds meaningful information to the node description and, "Author" that contains the name of the user who created the node. In addition to information holders, nodes can also act as composites that, combined with typed links, are used to represent two structural relationships: aggregation and generalization. When a composite generalizes/aggregates a number of nodes, a Generalization/Aggregation link is established from the composite to its components. The set of Nodes is managed by means of several operations related to the composition and management of nodes as well as with the definition of aggregations and generalizations. Versions can be defined by a special

typed version link. These three kind of structural links also can be personalized by associating them with userid and user group.

**Table 3-9: Dynamic operations related to Node**

Operation name	Description of the operation
createNode	It adds a new node. Some mandatory attributes have to be tied to the node by including new values into al. Normally, the node is first created as a personalized node by one author in the notation group, and then the editor in the editing group assigns the node to different group of users through the “changeNodeUserGroup” operation.
deleteNode	It removes a node. lo and al are updated. Anchors and links embedded into the node must also be updated, so A and L are modified. The user in the notation group can only delete the node when his userid is in the node’s UserList.
duplicateNode	It duplicates a node along with its elements (location of contents, anchors, attributes), except from the incoming links. The same components as in “deleteNode” are modified.
generalizeNode	It creates a relation of generalization between nodes $N_i$ and $N_j$ in the hyperdocument, making the elements associated with $N_i$ (location of contents, anchors and attributes) be inherited by $N_j$ . A special type of structural link (generalization link) is set between them and A, L and al are updated.
aggregateNode	It creates a relation of aggregation between nodes $N_i$ and $N_j$ in the hyperdocument, making $N_j$ be referred to by means of the identifier of $N_i$ . A special type of structural link (aggregation link) is set between them and A, L and al are updated.

decomposeNode	The result of a generalization or aggregation is removed from the node. That is, the corresponding generalization or aggregation link is deleted.
createNodeVersion	It creates a new version of a node. That is, a duplicated node is created and connected to the original node through a version link. The same components as in “duplicateNode” are updated.
deleteNodeVersion	One version of a node is removed. The node is deleted and the link connecting it to the previous and later version, that are now connected by a new link version. The same components as in “createNodeVersion” are updated.
changeNodeUserGroup	Modify the user group of the node; make the node to be viewed by different user groups. When a node in the notation group category changes group, the UserList should be cleaned. The attached links and anchors will not be changed.
shareNode	Modify the UserList of the personalized node; so it can be viewed by another author. This operation adds one userid to the UserList of the node, then this node can be viewed and modified by the new author with this userid. This node should belong to the notation category.
unshareNode	Modify the UserList of the personalized node; so it can not be viewed and modified by the specified author. This operation removes the userid form the UserList of the node. This node should belong to the notation category. If the last user is removed from the UserList and the node still belongs to the notation category, this node should be deleted, since no one owns this node anymore.
getGeneralizednodes	It returns the list of the nodes that generalize a given node. N,

	A, L are accessed but not modified.
getAggregatedNodes	It returns the list of the nodes that aggregate a given node. N, A, L are accessed but not modified.
getNodeComponents	It returns the list of the nodes that are generalized or aggregated by a given node. N, A, L are accessed but not modified.

### 3.2.3 Operations related to Content (C)

CONTENTS:  $C_i = (\text{ContentId}_i, \text{UserList}_i, \text{UserGroup}_i, \text{ContentType}_i)$

Content is an information item (e.g., a text, a graphic, or an animation) that can be placed into different nodes. According to its static definition in section 3.1, content ( $C_i$ ) consists of an identifier (ContentId), the user information (UserList and UserGroup) to specify which group user the content belongs to and the publicity of the content. The type (ContentType) defines the kind of information represented (Type<sub>i</sub>) and the different units that make up the content's representation space (Units<sub>i</sub>). The content identifier can be implemented in the same way as the node identifier to refer to both external and local resources. The same as a node, content can also be a composite and has associated attributes. Mandatory attributes of contents are the same as those of nodes. Operations concerning this set allow contents to be added, removed and duplicated from one hyperdocument to another.

**Table 3-10: Dynamic operations related to Content**

Operation name	Description of the operation
createContent	It adds new content. Some mandatory attributes have to be tied to the node by including new values into al. Normally; the content is first created as a personalized content by one author in the notation group, and then the editor in the editing group assigns the content to different groups of users through the “changeContentUserGroup” operation.

deleteContent	It removes content. lo and al are updated. Anchors and links embedded into the content must also be updated, so A and L are also modified. The user in the notation group can only delete the content when his userid is in the content's UserList.
duplicateContent	It duplicates content along with its elements (anchors, attributes), except from the incoming links. The same components as in the "deleteContent" are updated.
generalizeContent	A relation of generalization between contents $C_i$ and $C_j$ is established. The elements tied to $C_i$ will be inherited by $C_j$ (anchors, and attributes). A special type structural link (generalization link) is set between them and A, L and al are updated.
aggregateContent	A relation of aggregation between contents $C_i$ and $C_j$ is established, making $C_j$ referred to by means of the identifier of $C_i$ . A special type of structural link (aggregation link) is set between them and A, L and al are updated.
decomposeContent	The result of a generalization or aggregation is removed for content. That is, the corresponding generalization or aggregation link is deleted. A, L, and al are updated.
createContentVersion	It creates a new version of content. A copy is created and connected to the original content through a version link. The same components as in "duplicateContent" are updated.
deleteContentVersion	One version of content is removed. The content is removed as well as the version link associating it to the previous and later version that is now connected by means of a new link version. The same components as in "createContentVersion" are updated.

changeContentUserGroup	Modify the user group of the content; make the content to be viewed by different user groups. When content in the notation group category changes group, the UserList should be cleaned. The attached links and anchors will not be changed.
shareContent	Modify the UserList of the personalized content; another author can view the content by adding his userid to the list. This operation adds one userid to the UserList of the content, then this content can be viewed and modified by the new author with this userid. This content should belong to the notation category.
unshareContent	Modify the UserList of the personalized content; make the content so it cannot be viewed and modified by the specified author. This operation removes the userid from the UserList of the content. This content should belong to the notation category. If the last user is removed from the UserList and the content still belongs to the notation category, this content should be deleted, since no one owns this content anymore.
addUnits	It adds new units to the representation space of content. Adding units can affect the anchors placed in the content, therefore, A might be modified.
deleteUnits	It removes a specific unit from the representation space of content. Deleting units affects the anchors placed in the content. A might be modified. If the last unit of content is deleted, the content should be deleted too.
getGeneralizedContent	It returns the list of contents that generalize a given content. C, A, L are accessed but not modified.
getAggregatedContent	It returns the list of contents that aggregate a given content. C, A, L are accessed but not modified.

getContentComponents	It returns the list of contents that are generalized by or aggregated in a given content. C, A, L are accessed but not modified.
----------------------	--

### 3.2.4 Operations related to Anchor (A)

ANCHORS:  $A_i = (\text{AnchorId}_i, \text{NodeId}_i, \text{ContentId}_i, \text{UserList}_i, \text{UserGroup}_i, \text{AnchorPos}_i)$

An anchor is an interval of coordinates of a representation space that is used as a starting or ending point of links (e.g., a hot word in a text, and a hotspot in an image). Thus, anchors can be embedded into nodes and contents and, if the anchor interval is not specified, it refers to the whole component. As the static definition in section 3.1, an anchor ( $A_i$ ) has an identifier ( $\text{AnchorId}_i$ ), references to the elements where it is embedded ( $\text{NodeId}$  and  $\text{ContentId}$ ), the user information ( $\text{UserList}$  and  $\text{UserGroup}$ ) specify which group user the anchor belongs to and the publicity of the anchor, and an  $\text{AnchorPos}_i$ , which defines its initial location ( $\text{Position}_i$ ) and extension ( $\text{Extension}_i$ ). Anchors can be embedded into nodes and contents in three different ways:

- a. An anchor can be tied to a content and only be active when the content is presented in a particular node by assigning a value different from -1 to both  $\text{ContentId}_i$  and  $\text{NodeId}_i$ .
- b. An anchor can be tied to a content and be active in every node where the content is placed by assigning -1 to  $\text{NodeId}_i$ .
- c. An anchor can be tied to a node by setting  $\text{ContentId}_i$  to -1.

In cases a and b,  $\text{AnchorPos}_i$  is expressed using the different units that make up the representation space of the involved content. Thus, an anchor can be embedded into any place and any type of content. If the anchor is tied to a node, case c,  $\text{AnchorPos}_i$  refers to a temporal and/or spatial locus or interval (e.g., the right area of a node can be used to navigate to the next node whereas the left one can link to the previous node).

**Table 3-11: Dynamic operations related to Anchor**

Operation name	Description of the operation
createAnchor	It adds a new anchor. Normally; the anchor is first created as a personalized anchor by one author in the notation group, and then the editor in the editing group assigns the anchor to different group of users through the “changeAnchorUserGroup” operation.
deleteAnchor	It removes an anchor. lo and al are updated. If it is the unique source or target of a link, then the link is also removed. The user in the notation group can only delete the anchor when his userid is in the anchor’s UserList.
changeAnchorUserGroup	Modify the user group of the anchor; make the anchor to be viewed by different user groups. When an anchor in the notation group category is changed group, the UserList should be cleaned. The attached links will not be changed.
shareAnchor	Modify the UserList of the personalized anchor; make the anchor able to be viewed by another author. This operation add one userid to the UserList of the anchor, then this anchor can be viewed and modified by the new author with this userid. This anchor should belong to the notation category.
unshareAnchor	Modify the UserList of the personalized anchor; make the anchor not able to be viewed and modified by the specified author. This operation removes the userid form the UserList of the anchor. This anchor should belong to the notation category. If the last user is removed from the UserList and the anchor still belongs to the notation category, this anchor should be deleted, since no one owns this anchor anymore.

modifyAnchor	It modifies the values of NodeId <sub>i</sub> and/or ContentId <sub>i</sub> of an anchor definition in a document.
modifyAnchorPosition	It modifies the position and/or extension of an anchor in the document.
getAnchorLink	It returns the list of the links associated to an anchor. L is accessed but not modified. Return different result to different user, for user in the notation group, only return the personalized links he can access; for users in the editing group, returns all the links to the anchor associated.

### 3.2.5 Operations related to Link (L)

LINKS:  $L_i = (\text{LinkId}_i, \text{LinkStart}_i, \text{LinkTarget}_i, \text{UserList}_i, \text{UserGroup}_i)$

A link defines a relationship between two sets of anchors: the starting and ending points of a connection that can have navigational or structural purposes. All the links of the application are included in this set. As defined in chapter 3, each link ( $L_i$ ) consists of an identifier ( $\text{LinkId}_i$ ), two lists that contain the start ( $\text{LinkStart}_i$ ) and end anchors ( $\text{LinkTarget}_i$ ), respectively, the user information ( $\text{UserList}$  and  $\text{UserGroup}$ ) specify which user group the link belongs to and the access level of the link. Consequently, n-ary links can be defined. Moreover, since both the source and the target can be procedurally defined, virtual links (including dangling, warm, and hot links) can also be modeled.

The conceptual and physical separation between the link and the content/node where it is embedded makes the hyperdocument easier to maintain. Since the identifiers of nodes and contents refer to external resources, links can point at documents which do not belong to the local hyperdocument.

The dynamic management of links includes operations as creation, deletion, duplication, activation, as well as operations to obtain information about the structure of the hyperdocument (for instance, to retrieve the nodes connected by a link).

**Table 3-12: Dynamic operations related to Link**

Operation name	Description of the operation
createLink	It adds a new link. To set the value of the mandatory attributes, al is modified. Normally; the link is first created as a personalized link by one author in the notation group, and then the editor in the editing group assign the link to different group of users through the “changeLinkUserGroup” operation.
deleteLink	It removes a link. lo and al are updated. If the anchors associated with this link have no other associated links, these anchors are also deleted. The user in the notation group can only delete the link when his userid is in the link’s UserList.
activateLink	It activates a link. A is accessed but not modified.
changeLinkUserGroup	Modify the user group of the link; make the link to be viewed by different user groups. When a link in the notation group category is changed to another group, the UserList should be cleaned. The attached anchors will not be changed.
shareLink	Modify the UserList of the personalized link; make the link able to be viewed by another author. This operation adds one userid to the UserList of the link, then this link can be viewed and modified by the new author with this userid. This link should belong to the notation category.
unshareLink	Modify the UserList of the personalized link; make the link not able to be viewed and modified by the specified author. This operation removes the userid form the UserList of the link. This link should belong to the notation category. If the last user is removed from the UserList and the link still belongs to the notation category, this link should be deleted, since no one owns

	this link anymore.
getLinkTargets	It returns the list of targets of a given link. A is accessed but not modified.
getLinksSources	It returns the list of sources of a given link. A is accessed but not modified.
getLinksTo	It returns the list of links whose target is a specific node or content. A and L are accessed but not modified. If the element is a generalized node or content, then a recursive process retrieves the inherited links.
getLinksFrom	It returns the list of links whose source is a specific node or content. A and L are accessed but not modified. If the element is a generalized node or content, then a recursive process retrieves the inherited links.

### 3.2.6 Operations related to Attributes (B)

ATTRIBUTES:  $B_i = (\text{AttributeName}_i, \text{Value}_i)$

An attribute is a characteristic that will be used to add semantic information to an element of the application. The attributes are associated with the elements to which it belongs, so attributes' accessibility is determined by its associated element. As described in section 3.1, an attribute ( $B_i$ ) is defined by means of a label ( $\text{AttributeName}_i$ ) and a value ( $\text{Value}_i$ ). There are some mandatory attributes, such as "Label," "Author," "LinkType," or "Direction," that always exist in a hyperdocument. The dynamic part of the model includes operations to create, delete, and modify attributes. Moreover, information about the attributes can be retrieved. With respect to the security rules, only users granted an editing category will be able to modify all the attributes. To modify an attribute, the user has to get access to the associated element first.

**Table 3-13: Dynamic operations related to Attribute**

Operation name	Description of the operation
createAttribute	A new attribute is added to a component.
deleteAttribute	An existing attribute is removed from a component. The attribute list of the components associated with this attribute is modified.
modifyAttributeValue	The existing value of an existing attribute is modified.

### 3.2.7 Attribute List Function (al)

Attributes are tied to users, nodes, contents, and links by means of the al function, which acts as a repository of properties. The default value specified in the attribute definition can be changed giving a new value ( $Value_i$ ). Since each element has at least one mandatory attribute “Label,” all users, nodes, contents, links and anchors defined in the hyperdocument take part in this function. Indeed, whenever a new user, node, content, or link is created, the list of its mandatory attributes is added to this function and if no value is specified for them, the default value given in the attribute definition is assumed.

Attributes constitute a powerful source of information about users, nodes, contents, and links that can be exploited in different ways. The dynamic part of the model contains operations that permit assigning attributes to nodes, contents, links, and users, as well as modifying their value. Designers are also provided with several access operations, which can be used to seek information about the different tuples of this function.

**Table 3-14: Dynamic operations of attribute list function**

Operation name	Description of the operation
assignAttribute	It assigns an attribute to a user, node, content, link or anchor. The attribute can be bound to a specific value.
deassignAttribute	It deletes the assignment between an attribute and a user, node,

	content, link or anchor.
modifyValue	It modifies the value of an attribute.
getAttributeList	It returns the list of the attributes of the components the associated attribute.
getAttributeValue	It returns the value of an attribute.

### 3.2.8 Location function (lo)

Contents can be placed into nodes by giving a value to the lo function. A logical value (Position<sub>i</sub>) specifies a spatial locus in the node representation space, while another one (Time<sub>i</sub>) defines the time when it is presented (StartTime<sub>i</sub>) and the duration of this presentation (Duration<sub>i</sub>).

**Table 3-15: Dynamic operations of location function**

Operation name	Description of the operation
placeContentNode	A content is placed into a particular node, a user of the notation group can put a personalized content to a personalized node, then the editor of the editing group changes the group category through the “changeContentUserGroup” and “changeNodeUserGroup” operation
removeContentNode	Content is removed from a node, if an anchor is tied to the node and to the content, the anchor is also removed.
alignContents	The location of two groups of content inside one node is compelled to fulfill a particular spatial relation.
getSpatialRelation	It returns the spatial relation that exists between two groups of content. The lo function is accessed but not modified.
synchroniseContents	The location function of two groups of content inside one node is

	compelled to fulfill a particular temporal relation.
getTemporalRelation	It returns the temporal relation that exists between two groups of content.
changeLocation	Content is moved in the representation space of a node.
getLocation	It returns the position of content in a specific node. The lo function is accessed but not modified.
getContentsNode	It returns the list of the contents of a node. The lo function is accessed but not modified.
getNodesContent	It returns the list of the nodes to which content is placed. The lo function is accessed but not modified.

### 3.3 The conceptual level aspects of the model

After we define the static components of the document model, we can apply the aspect-oriented technique to the documents on the conceptual level to solve some problems such as access control, generate different personalized view according to the userid and group category of the contents, nodes, anchors and links, and deal with the member activity tracking.

By adding user information to content, node, link and anchor, we can identify the following concerns and deal with them separately with the help of AOP techniques:

- Simultaneous multi-user access, including robust concurrency control mechanisms. Allow multiple users to access concurrently the development environment while preserving the consistency and integrity of data.
- Personalized view generation. How to generate personalized views from the document according to the person's userid using the personalized setting in the components of the model.

- Support different level of annotations (private, workgroup or public). Creation of annotations (comments) is a basic right for hypermedia readers as well as a basic tool for collaboration and exchange of ideas. The environment should support private, workgroup or public annotations, providing them to any person who has the appropriate access permissions.
- Access control while allowing multi-user access and even allowing end users have limited rights to edit the document (such as to make personal notes). Access permissions may prove most useful for allowing individuals to maintain a personal set of nodes, links and annotations, while collaborating workgroups maintain shared sets of objects.
- Member activity tracking. Tracking actions and / or modifications made by each member of a team.

### **3.3.1 Assign users to different group category**

After a document was designed according to this model, the users are classified as belonging to several group, the group category defines the users' access level to the document.

The user in the browsing category can only browse the document, and only can browse the components (content, node, link and anchor) that are assigned to this group category. Generally, the user in this group can not have a personalized view of the document, so the UserList should be empty. The users of this group category should be the reader of the document.

The users in the notation category can browse the document and make personal extension to the document. Then the users in this category can add their personal userid to the UserList of the components (content, node, link and anchor), the user can view the general contents assigned to this group and the personal contents match their userid, the general content in this group may be different from the content viewed by the browsing group member. The users in this group should be the people who are the author of the document.

The user in the editing group category can perform arbitrary editing activity on the document and view all the contents of the document. The users in this group should be assigned carefully to the people in the administration level; the user in this group can change the personalized components to general components (by remove personal UserId from the UserList of the components) and change the contents to different group category (e.g. change components from notation group category to browsing group category, this will let the reader read more content). Normally, the user in this group will not need the personalized view of the document, and can view all the personalized components of the notation group category by selecting different personalized links (e.g. if a personalized anchor associated with multiple links is clicked, a way is provided such as a pop-up window, to let the user selects the appropriate link to go). The user in this group is responsible for the final editing of the document.

#### **3.3.2 Generate personalized view of the document**

When a document is selected to view by a user, the user's group category is checked first, the components of the document are scanned; only the matched category is marked as viewable and moved to a temporary space. This can generate document views dynamically from the base document.

For the matched content, it also can be classified to three categories:

- For the reader group, the users only browsing it. The view can be generated and stored in a place and when the user in this group chooses to view the document, this saved copy can be activated and displayed to the user directly. When editing to the document happened, this view can be regenerated and saved accordingly.
- For the notation group, there are common components which are shared by all the authors, these are the components with the UserList empty; but some components belong to one author or a small group of authors, their userid is included in the UserList, these components are personalized components. The personalized components are extensions the author made to the base document, and they also

belong to this notation group. When users in this group choose to view the document, the views should be generated dynamically according to the userid. This can be achieved through AOP technique. Select all the components in the notation group, but before the components are displayed we check if the userid matches the userid in the UserList (to see if the components belong to the user), then decide if this user can view this component.

*aspect Displaying*

*pointcut Displayed():*

*display components with group category are notation;*

*before(): Displayed()*

*Check if the userid does not match the userid in UserList, select the components that is belongs to this userid with proper xpath construct. Display the selected contents.*

- For the editing group, the display is also simple; the editor of the document can view everything, so everything is marked as viewable. The different personalized links can help the navigation.

### 3.3.3 Simultaneous multi-user access

This model can easily support simultaneous multi-user access, with robust concurrency control mechanisms. Multiple users can be allowed to access the development environment concurrently while preserving the consistency and integrity of data.

According to the user group category, only two groups of users can modify the document – the notation group and editing group. The browsing group can not make modifications to the document.

For the notation group, the author can not make modifications to the base document. All the personal extension to the document is classified as personal notation. If deleting the content of the base document is needed, the author can describe what should be deleted and what should be added instead through the personal notation, when the suggestion is approved, only the editor can make the adjustment to the base document accordingly. All the

users in this group can make notations to the document, but the notations are only related to the user making the modification, and can not affect other users in the same group. In this way, all the authors can access the document and make notations to it concurrently, the data consistency and integrity is preserved.

The editing group handles changes to the base document; operations such as delete, update, create, move and modify the components of the base document are handled by users in this group. This kind of operations can be separated into two steps: first, users in the notation group make modifications to their personalized document; second, the user in the editing group verifies the modification, and change the personalized view to group view. Then the update to the components can be achieved by one operation. For example, change some user's personal components to group level, just find all the UserList with the userid in it and remove the userid from that UserList, in the mean time, the group category can also be changed.

#### **3.3.4 Support different level of annotation**

After the user group categories are classified and the personalized view is generated dynamically, we can support different level of annotations (private, workgroup or public) easily. Only the users belong to the notation group can make annotation to the document, to achieve different level of annotation; we can further divide the notation group category into several smaller groups. Then the user from different sub-group can make different personalized annotation to the document, these annotations can be viewed according to the userid.

If the user wants to share a personalized notation in a small group, the user can add the userid of this group into the UserList of that personalized notation, the notation then can be viewed by the members of this group, and this notation is shared by this group. The user with the appropriate access permissions can make a different level of annotations to the document easily.

### 3.3.5 Access Control and Member activity tracking

Access Control is very important to the management of the document. A document model should provide access control mechanisms while allowing multi-user access to the document. In this model, different user group categories have different level of access permissions to the document. The browsing user group can only browse the components classified to this group. The notation user group can browse the components that classified to this group and make personalized notations to the document. This group of users has limited rights to modify the document. This group of users maintains a personal set of components, contents, nodes, links and anchors in the form of notations, while collaborating workgroups maintain shared sets of components that belong to the group. The users in the editing group have the highest level of access to the document. They can access all the group components and all the personalized components. They can make critical modification to the document. The user in this group should be restricted to the person who is in charge of the document modification, and not every author of the document can modify the document. The access control can be implemented with AOP technique in the following method:

```

aspect AccessControl

    pointcut Modification():
        Capture the modification operation like delete *, modify *,
        update *, create *. (The * include: content, link, anchor, node,
        attribute, etc).

    before(): Modification()
        Call the access control to confirm the user has the rights to
        do the indicated operation.

```

Member activity tracking is also very important to the management of the document. The activity of the browsing group does not need to be tracked, they only browse the document. The activity of the notation group also does not need to be tracked; the users in this group only modify their own components. The activity that they perform does not affect other users of the document. Only the activity of the user in the editing group has to be tracked, the user in this group makes all the modification to the document, and this activity affects all the users

of the document. The member activity tracking can be implemented with AOP technique using the following method.

```
aspect ActivityTracking

    pointcut Modification():

        Capture the Modification operation like delete *, modify *,
        update *, create *. (The * include: content, link, anchor, node,
        attribute, etc).

    before(): Modification()

        Log the userid, component, time and other information.

    after(): Modification()

        Log the userid, component, time and other information.
```

Since AccessControl and ActivityTracking are triggered by user's activity of modification to the document, it is necessary to give the precedence and sequence to these activities. This is implemented as follow:

```
declare precedence: AccessControl, ActivityTracking, * ;

    At each join point, advice from AccessControl has precedence over
    advice from ActivityTracking, which has precedence over other advice.
```

### 3.4 Handling operations automatically, anonymously, and consistently

When an operation is performed on an object, all the other objects that may be affected by this operation have to be checked to maintain the consistency of the document. The model consistency also can be treated as an aspect of the model. The way to deal with the consistency problem can be combined with the operations of dealing with the interactive behavior of the document by the AOD techniques. For example, when content is deleted, the associated anchors and links to this content should be found and deleted together, the link target content from this content also have to be checked, the target anchor of this link should

be removed. This series of operations can be achieved easily by AOD techniques as the following:

```
pointcut deleteContent(): capture call of deleteContent

before: deleteContent()

    Find all the anchors and links which source is this content and
    also find all the contents which are link target from this
    content

after: deleteContent()

    Delete the anchors and links; remove the link from the target
    content and the link target anchor from the target content.
```

This way, when an operation is captured, the auxiliary operations are performed automatically and anonymously, the designer need only consider the necessary operations that have to be accomplished by the delete content operation, the consistency problems are dealt with by the AOD techniques automatically, and in the *after* part of this aspect, the delete anchor and delete link operations will be automatically captured and necessary consistency maintenance will performed according to the definition of the aspect of delete anchor pointcut and delete link pointcut.

In this model, we will deal with the interactive operation (especially the mutation operation to the document) as aspect, and the consistency maintenance of the model will be performed together. We decided to do this is based on the thinking that only the modification to the document can affect the consistency of the document. In the following section, we will define the interactive operations of the model and how to apply AOD techniques to model these operations as aspect.

For node, content, link and anchor, some of the dynamic operations for these components are similar, so they can be captured and dealt with together.

### **3.4.1 Create components operation**

The operations that create these components can be captured, and the common operation can be dealt with together in one aspect.

```

aspect CreateComponent

    pointcut createComponents(): execution(createNode(..)) ||
        execution(createContent(..)) || execution(createLink(..)) ||
        execution(createAnchor(..))

    after(): createComponents() {

        Find the mandatory attributes that tied to the components, call
        proper attribute setting function to include new attribute values to the
        attribute list (al).}
    }

```

### 3.4.2 Delete components operation

The operations that delete these components can be captured, and the common operations can be dealt with together in one aspect. It is a little bit different for the delete operation, the nodes and contents share the similar operations such as delete embedded links and anchors; the deletion of links and anchors does not affect the node or content in which they are embedded. So, these operations should be dealt with separately.

```

aspect DeleteComponent

    pointcut deleteNodeorContent():
        execution(deleteNode(..)) || execution(deleteContent(..))

    before(): deleteNodeorContent() {

        Find all the links and anchors that embedded in this node or
        content.}
    }

    after():deleteNodeorContent() {

        Update the al and lo function of the related component.

        Remove the anchors embedded in this component.

        Remove the link from information in the link attribute. If this is
        the only component the link from, delete the link}
    }

    pointcut deleteLinkorAnchor():
        execution(deleteLink(..)) || execution(deleteAnchor(..))

    before():deleteLinkorAnchor() {

```

```

        Find the related links and anchors that are associated with
        this links or anchor.}

    after():deleteLinkorAnchor() {

        Update the al and lo function of the related component.

        Update attribute information of the anchor when delete a link, if
        this is the only link the anchor starts or ends, delete the anchor too.

        Update attribute information of the link when delete an anchor, if this is
        the only anchor the link starts or ends, delete the link too.}

```

### 3.4.3 Change user group operation

The node, content, link and anchor all have operations to change user group, we can capture it as one aspect and deal with it together.

```

aspect ChangeUserGroup

    pointcut changeUserGroup():

        call(changeNodeUserGroup(..)) ||
        call(changeContentUserGroup(..)) ||
        call(changeLinkUserGroup(..)) ||
        call(changeAnchorUserGroup(..))

    before(): changeUserGroup() {

        Find all the components that associated with the element to be
        changed group and still remained in old user group}

    after(): changeUserGroup() {

        Prompt the editor to decided if he wants to change the
        associated components that found in the before() operation.

        If the editor responses a yes to this prompt, another call to
        this component will happen with the same arguments.

        This operation will cause a chain operation to keep the
        consistency of the related components.}

```

### 3.4.4 Create components version

The operations that create the version of the components can be captured, and the common operation can be dealt with together in one aspect.

```
aspect CreateComponentVersion

    pointcut createNodeVersion(): execution(createNodeVersion(..))

    before():createNodeVersion() {

        Find the node that to be duplicated

    }

    after():createNodeVersion(){

        Create the version link between the base node and the new node
        version.

    }

    pointcut createContentVersion(): execution(createContentVersion(..))

    before():createContentVersion(){

        Find the content that to be duplicated

    }

    after():createContentVersion() {

        Create the version link between the base content and the new
        content version.

    }
```

### 3.4.5 Delete components version

The operations that delete a version of the components can be captured, and the common operation can be dealt with together in one aspect.

```
aspect DeleteComponentVersion

    pointcut deleteNodeVersion(): execution(deleteNodeVersion(..))

    before():deleteNodeVersion {

        Find the nodes that are the previous and next version of this
        node. Get the version links to and from this node.

    }

    after():deleteNodeVersion(){

        Delete the version link to and from this node and adjust the
        related nodes' anchor. Create new version link between the previous
        and next version node.

    }

    pointcut deleteContentVersion(): execution(deleteContentVersion(..))
```

```
before():deleteContentVersion(){  
  
    Find the contents that are the previous and next version of  
    this content. Get the version links to and from this content.}  
  
after():createContentVersion(){  
  
    Delete the version link to and from this content and adjust the  
    related contents' version link anchor. Create new version link  
    between the previous and next version content.}
```

Many other mutation operations also can be captured and dealt with by the help of the AOD technique. By capturing the critical operations and dealing with the common operation together in the before and after operation, the system's consistency is maintained automatically. The designer can focus on the major operations and leave this consistency maintenance to the system.

### **3.5 Modeling event handling with AOP Techniques**

An event is the representation of some asynchronous occurrence (such as a mouse click on the presentation of the components, or the removal of a node or content from the document) that gets associated with an event target; the event target is the object related to the event. The core concept of event handling is that the system captures the user's action on the components of the system, and responds to that action with certain predefined operations. The process executed when the event is triggered is expressed using the set of operations making up the dynamic part of the model. Thus, events are a useful way to specify interactive behaviors.

Many document models that are implemented with object-oriented techniques employ event handling mechanism to deal with different events directed at different purposes. Models like OOHDM use events only for presentation purposes. The Labyrinth model enlarged this concept by making use of events to model interactive behaviors. The DOM model extends the user of events to an even larger range of purposes. DOM uses event handling to deal with presentation (by means of capturing the GUI event and the keyboard

event), navigation (by capturing the mouse event) and the document structure and content modification (by capturing the mutation event). DOM also specifies two special event to deal with the load and save aspect of the document model (The DOM Load and Save specification).

The mechanism of event handling has two parts. An object registers an event listener for the kind of events related to this object. The object body contains the definition of the operations that must be performed when the matched events occur. The concept of event handling is to capture the event by the event listener, and react to the event with predefined operations performed by the object that is responsible for this event. With AOD techniques, we can remove the event listener from the object and capture the event directly through the function calls and capture the function executions. This way, we can also remove the event dispatch mechanism from event handling, the event dispatch mechanism is to define how a composite component react to an event after it receive it, when this event is targeted to one small piece of this composite component. Using the AOD technique, the operation to the small piece can be captured directly through defining different target object signature of the operation. In our model, we try to use AOD technique to define the interactive behaviors. The interactive behavior will be captured as pointcut, and dealt with as an aspect.

One example of dealing with event handling using AOP technique is to model the mouse event. The object-oriented design technique captures the static components of the system and model it as object. The aspect-oriented design technique can capture the common operations of the object and deal with these operations separately as aspect. The different event of the system can be dealt with as an aspect and the object's operation is simplified. Different event can be captured as different aspects and each aspect can deal with the dynamic behaviors of different object. For example, the mouse click event for different component can be treated as one aspect.

```
aspect MouseClicked  
  
    pointcut mouseClickComponent(): execution(mouseClick(component))  
  
    before():mouseClickComponent() {
```

```
    Find the type of the component clicked, (which means find what
    type of component (node, content, link or anchor) is clicked)
after():mouseClickComponent(){

    Define the behaviors of the components clicked:

    Case node:

    Case content:

    Case link:

    Case anchor:

    Case other components:
```

Other mouse events can be defined as different aspects in a similar way. The system is enhanced with an organized aspect module, and each object is simplified by moving the event handling operations to these aspects.

# Chapter 4

## Case Study

In this chapter, we make several case studies to show the concepts described in the former chapters. To construct the cases, we select AspectJ as the programming language; AspectJ is the AOP extension to java. We also select DOM (Document Object Model) as the target document model. DOM can be viewed as a subset of the general document model described in the former chapters. DOM emphasizes the Content and Node and structure aspect of the general document model, and is less focused on the link and anchor aspect. To work with AspectJ, the java based AOP language; we select a java implementation of DOM – dom4j as the base document model implementation. We will give a detailed introduction of AspectJ and dom4j in section 4.1 and 4.2 respectively.

In our case study, we want to show the benefit of applying AOP techniques to document models by implementing specific cases using aspects. These problems include the ones that are hard to be solved by classic OO techniques, such as access control and member activity tracking, and problems that deal with the dynamic operations of the document and its components, specifically, capturing the modification operation and deal with it separately or introduce additional operations through aspect constructs. We also use AOP techniques to solve the problem of generating various views for different user groups and make personalized views and annotations available to users.

### 4.1 AspectJ

In this section, we introduce the major concept of AspectJ and the major constructs of AspectJ which we will use in our case study.

AspectJ[XP04] is an aspect-oriented language that supports the separation between the software components developed in languages such as Java and the aspects, which are properties of a system for which the implementation cannot be cleanly encapsulated as a single functional unit of the overall system [XP04] [KHH01]. The aspects can crosscut components in a system's implementation and in general denote non-functional units.

“An aspect is a crosscutting type” [XP04]. Crosscutting holds the connotation of crossing boundaries of encapsulation. The term crosscutting is often related to the word concern to describe a property or quality that has to be added to system functionality. The difficulty with crosscutting concerns is that they cannot be directly modeled with OO constructs. The crosscutting concerns do not map nicely to classic OO types. To allow direct modeling of crosscutting concerns, AspectJ supports crosscutting types via the aspect construct. In an aspect type, operations are defined in terms of which join points they apply, where a join point is “a well-defined point in the program flow” [XP04], and what qualifications the operations make on the behavior of those join points. The crosscutting related syntax appears independently or as prefaces to non-crosscutting syntax, which simplifies the separation of the two.

Aspects are defined using join points and associated additional implementation information called “advice.” In a general way, join points are used to characterize well defined points in the execution of an aspect program and advice is used to describe what should be done and in what order some extra code should be added when that execution point is reached.

#### **4.1.1 Joint points**

Join points are well-defined points in the execution of a program, and include method and constructor calls, and field accesses such as assignment and reference. Join points are selected sets using pointcut specification. The category of join point is summarized in the following table.

**Table 4-1: Join point category**

Join point category	Description
<i>Method call</i>	When and where a method is called, not including <i>super class</i> calls.
<i>Method execution</i>	When the body of code for an actual method executes.
<i>Constructor call</i>	When an object is built and a constructor is called, not including <i>this</i> or <i>super class</i> constructor calls.
<i>Initializer execution</i>	When the non-static initializers of a class run.
<i>Constructor execution</i>	When the body of code for an actual constructor executes, after its <i>this</i> or <i>super</i> constructor call.
<i>Static initializer execution</i>	When the static initializer for a class executes.  A static initializer is a block of code defined as a class member and which starts with the static keyword. I.e. it is not part of a constructor or method.
<i>Object initialization</i>	When the object initialization code for a particular class runs.  This encompasses the time between the return of its parent's constructor and the return of its first called constructor. It includes all the dynamic initializers and constructors used to create the object.
<i>Field reference</i>	When a non-final field is referenced.
<i>Field assignment</i>	When an assignment is made to a field.
<i>Handler execution</i>	When an exception handler executes.

### 4.1.2 Selecting pointcuts

A pointcut is a program element that picks out join points, as well as data from the execution context of the join points. Pointcuts are specified using combinations of primitive pointcut designators. A primitive pointcut designator determines the temporal point in program execution where the pointcut occurs. A primitive pointcut designator is qualified by an argument that is usually a signature or type pattern. This argument defines the position, relative to program source code, where the pointcut occurs. Certain wildcards are available for specifying signatures and type patterns. The following table presents the list of primitive pointcut designators.

**Table 4-2: Primitive pointcut designator**

Primitive pointcut designator	Description
<i>call(Signature)</i>	Selects method and constructor call join points. These join points occur before control is passed from the caller to the callee.
<i>execution(Signature)</i>	Selects method and constructor execution join points. These join points occur after control is passed from the caller to the callee.
<i>Initialization (Signature)</i>	Selects object initialization join points. If multiple constructors are executed during object creation and more than one is selected by the signature, only first one executed is selected by this pointcut primitive. These join points occur after control is passed from the caller to the callee.
<i>get(Signature)</i>	Identifies accesses to an object attribute.
<i>set(Signature)</i>	Identifies assignments to an object attribute.

<i>staticinitialization</i> ( <i>TypePattern</i> )	Identifies static initializers of types identified in the type pattern parameter.
<i>within</i> ( <i>TypePattern</i> )	Identifies all join points defined by those types selected in the type pattern argument.
<i>withincode</i> ( <i>Signature</i> )	Identifies all join points that occur within the method and constructors that match the signature parameter.

In many cases, several primitive pointcut descriptors are required to broaden or narrow the number of join points being selected. To accomplish this task, logical operators are available to combine several pointcut specifications. AspectJ supplies a reflective object that dynamically provides the execution context of program execution. A primitive pointcut designator is available that dynamically derives join points by carrying out Boolean comparisons on elements of the reflective object. As the following table shows:

**Table 4-3: logical and Boolean operations to pointcut**

Primitive pointcut designator	Description
<i>! Pointcut</i>	Picks out all join points that are not picked out by the pointcut.
<i>Pointcut0 &amp;&amp; Pointcut1</i>	Picks out all join points that are picked out by both of the pointcuts.
<i>Pointcut0    Pointcut1</i>	Picks out all join points that are picked out by either of the pointcuts.
<i>( Pointcut )</i>	Picks out all join points that are picked out by the parenthesized pointcut.
<i>if</i> ( <i>BooleanExpression</i> )	Picks out all join points where the Boolean expression

	evaluates to true.
--	--------------------

Naming a pointcut takes place in two parts. First, the pointcut is given a name that allows it to be referred to independently of its definition. Furthermore, interesting variables in the context of the pointcut are given a typed name so that they too can be referenced. The second portion involves identifying the join points in the program to which the pointcut corresponds and mapping the typed names to variables in the context of the join point.

### 4.1.3 Signature

Signatures are used to cite members of types. In AspectJ, these members are either attributes, i.e. member data, or methods, including constructors. The two categories of signatures correspond to field and method references. Constructor signatures are a special case of method signatures.

Method signatures are used as parameters for call, execution, initialization and withincode primitive pointcut designators. Field signatures are used as parameters for set and get primitive pointcut designators.

### 4.1.4 Using wildcards

There are two wildcards: ".." and "\*".

".." corresponds to a variable size array of parameters. In the case of AspectJ, ".." is used to indicate zero or more arguments, so `execution(void m(..))` picks out execution join points for void methods named `m`, of any number of arguments, while `execution(void m(.., int))` picks out execution join points for void methods named `m` whose last parameter is of type `int`.

"\*" corresponds to any valid identifier or modifier, or portion thereof. Indeed, most elements can be replaced by wildcards. So `call(public final void C.foo())` picks out call join points to that method, and the pointcut `call(public final void *.*())` picks out all call

join points to methods, regardless of their name or which class they are defined on, so long as they take no arguments, return no value, are both public and final.

#### 4.1.5 Advice

Advice is used to specify how to execute behavior relative to the join points. In some cases, however, the behavior is executed rather than the join point. There are three kinds of advice: before advice, after advice, and around advice. The before (after) advice executes before (after) the join point. The around advice executes in place of the join point, but it can still activate the join point when required.

The parameters of advice statements are passed by copy. AspectJ allows the behavior of an advice statement to make use of reflective objects that providing varying levels of detail on the context in which the advice is executing. These objects allow advice behavior to be written in a more general fashion.

There are three approaches to defining after advice. The type of after advice used determines how an exception call is dealt with by the advice. When the keyword **returning** is used, the after advice does not execute if an exception is thrown by the join point. The keyword **throwing** is used if the advice executes only if the join point throws an exception. Finally, if neither keyword is used, the advice executes in both cases. They are demonstrated in the following example:

```
aspect A {  
    pointcut publicCall(): call(public Object *(..));  
    after() returning (Object o): publicCall() { some action }  
    after() throwing (Exception e): publicCall() { some action }  
    after(): publicCall(){ some action }  
}
```

Another capability of after advice is to expose the return value of the join point. This is done by specifying a formal parameter following the keyword after, to correspond to the

return type. Should the return type be a primitive type, the variable is boxed and an object reference returned.

The around advice replaces the execution of a join point, the `proceed` keyword offers the opportunity to perform processing before and after the execution of the join point. Around advice can selectively invoke the join point it replaces using the `proceed` keyword, which passes control to whatever was replaced by the around advice.

#### 4.1.6 Static and dynamic crosscutting

Aspects can crosscut components in a system's implementation in two ways: static and dynamic. Static crosscutting modifies structure and corresponds to the addition of type members. Dynamic crosscutting modifies behavior and corresponds to the use of advice statements that are used to modify the behavior of join points. Crosscutting between aspects and components is a key feature of aspect-oriented programming. The benefit of crosscutting is the replacement of scattered, tangle-prone and ad-hoc composition by a disciplined combination of separated aspect behavior with component at well-defined join points. In this paper, we only consider the static crosscutting.

## 4.2 DOM Application

### 4.2.1 dom4j

dom4j is an Open Source XML framework for Java. dom4j allows one to read, write, navigate, create and modify XML documents. It integrates with DOM and SAX and is seamlessly integrated with full XPath support. While DOM is a quite large language independent API, dom4j is a simpler, lightweight API which is optimised for the Java language making extensive use of the Java 2 platform such as the Java 2 collections. Though dom4j fully supports the DOM standard allowing both APIs to be used easily together. Dom4j attempts to make it easier to use XML on the Java platform.

dom4j is an object model representing an XML Tree in memory. It offers a easy-to-use API that provides a powerful set of features to process, manipulate or navigate XML and work with XPath as well as integrate with SAX and DOM. dom4j is designed to be interface-based in order to provide highly configurable implementation strategies. The XML tree implementations can be created by simply providing a DocumentFactory implementation. This makes it very simple to reuse much of the dom4j code while extending it to provide other implementation features.

In this chapter, we use two simple applications as examples to show that the ideas described in chapter 3 are practical. In these two applications, we illustrate how to apply AOP techniques to a DOM implementation. We use the programming language AspectJ to implement and compile the aspect construct and java implementation of DOM – dom4j to provide the APIs for DOM functionality. We choose these cases to show how to extend DOM with different aspect implementations. These are not all the cases that can be applied to a complicated application. Many other behaviors can be dealt with similarly, such as dealing with the exception handling as an aspect, or capturing the operation to handle more dynamic behaviors by implementing them as an aspect instead of as an event.

#### **4.2.2 Set up user groups and group categories**

As described in chapter 3, the user group category generally can be divided into three large groups:

- Browse: the users in this group can only browse the document and cannot make any modification to the document.
- Notate: the users in this group can browse the document and make some personalized annotation on the document, but cannot perform critical modification such as delete, change and update to the information in the document.
- Edit: the users in this group can browse the document and make all the modifications necessary to the document.

The view of the document is generated according to the user group information; each user group has its own specific view of the document that is designed to this group of user.

The browse category can be assigned to the user group – Browser. The Browser can only view the portion of the document content assigned to their user group. If necessary, the browser can be further divided into several smaller user groups according to the user's browsing interest.

The notate category can be assigned to the user group – Notation. The users in this user group make annotations on the document. So the users in this group not only can view the general contents that are classified to the Browser, they also can view the personalized content belonging to themselves. The users in this group only have limited rights to the modification of the document, such as make annotations to the nodes of the document. Many critical mutation operations are not allowed to the users in this user group.

The edit category can be assigned to the user group - editor. The members of this group are responsible for making modifications to the document and creating new users for different user groups. The modifications include:

- change the components' user group, this operation modifies the components user group.
- update values of components, this operation modifies the content of the document.
- change the structure of the document.

After the user groups have been set up, the users can be created and added to different groups; and the document and its components can be associated with the different user groups and developed and maintained accordingly. Those users granted an “editor” category will be able to create nodes, content, and other elements and, therefore, they will contribute to the document development. Those users having a “notation” category will be able to create personalized document notations that will not be seen by other users. Finally, users assigned a “browser” category will only be able to read and interact with the document.

### 4.3 Example 1: Member Activity Tracking

After the user group information is associated with the document and its components, it is useful to record the user activities and store this information for future reference, this kind of activity includes userid, user group, time of activity, modification made to what components and what modification made. To simplify this example, we do not go to the detail of the userid level; we only record the users' group information.

This functionality involves several classes and many operations to the components of the document. If implemented using the classic OO technique, all the classes and methods should include codes regarding this functionality. This approach will cause messy and hard to maintain source code. We capture the users' activity in an aspect and log it.

This example gets the state of the current JVM system properties and generates an XML file through dom4j. In this example, create document and many create and add operations are performed on different components; we use AspectJ to capture these operations and log them through the activity tracking aspect. We set the default user group as "Editor".

#### 4.3.1 Pointcut:

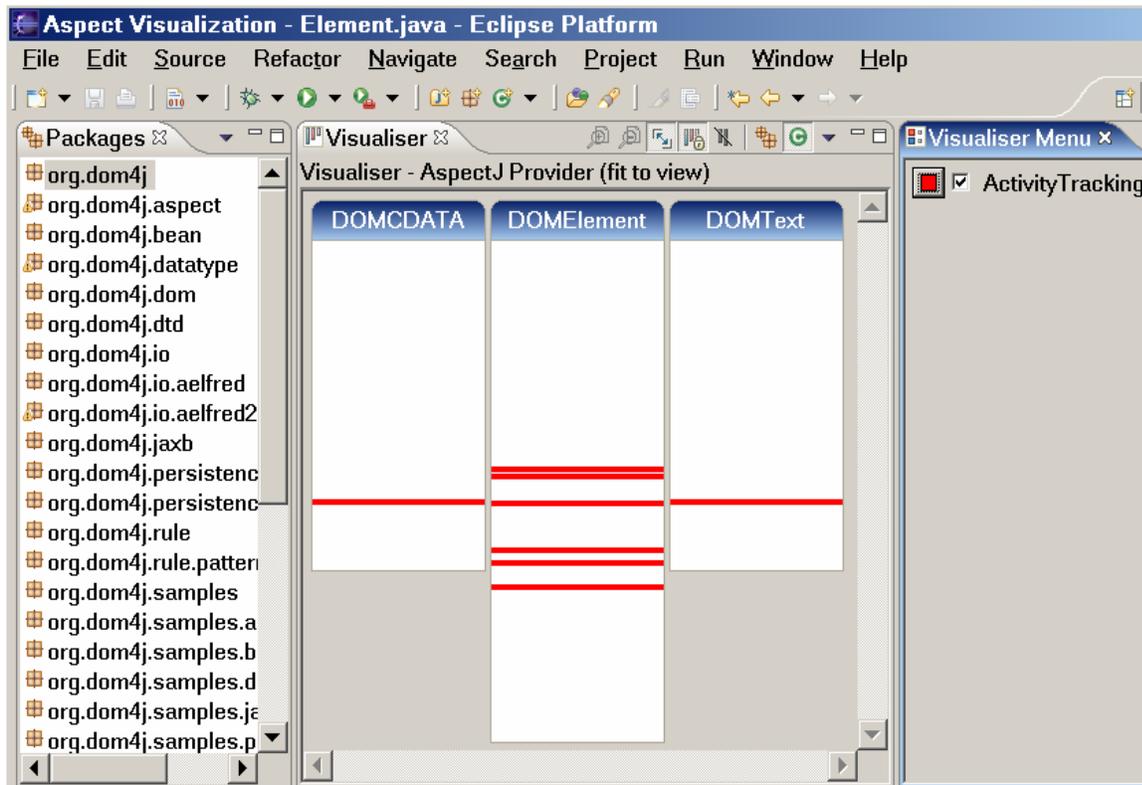
The first thing to implement the member activity tracking aspect is to capture the operations through AspectJ's pointcut construct. For this aspect, the modification operations performed to the document and its components should be captured and recorded. This aspect is only interested in recording the result of the modification, so the failed attempts (such as users from the "Browser" group category attempting to modify the document and refused by the access control aspect.) should not be recorded. Only the execution of the modification is captured and the after advice is used to handle this aspect.

```
pointcut modifiedDoc(): execution(static * DocumentHelper.create*(..))
    || execution(void Element.add(..))
    || execution(Element Element.add*(..))
    || execution(void Element.appendAttributes(..))
    || execution(Element Element.createCopy(..))
```

```
|| execution(boolean Element.remove(..));
```

### 4.3.2 Cut through many classes and operations:

This pointcut cut through the DocumentHelper class that implements the helper method for the document class, and the Element interface that defines the structure modification to the elements of the document, this class and interface are in the dom4j package, This pointcut also captures all the classes which implement the Element interface; these classes include the AbstractElement and DefaultElement classes in the dom4j tree package. The wildcard “\*” and “..” in this pointcut helps to capture many of the operations on the components of the document.



**Figure 4-1: ActivityTracking aspect affects classes in DOM package**

Figure 4.1 is the screen shot of the visualization of the member activity tracking aspect in the package DOM. This screen shot obtained from the Aspect Visualization perspective of

the Eclipse project [Ecl04]. Each line in the figure is one method call captured by the aspect. This aspect also crosscuts many other packages in the dom4j project. This seemingly simple aspect can dramatically simplify a complicated application.

### 4.3.3 Advice:

The purpose of this aspect is to record related information after the action is performed, so after advice is used.

```
after() returning: modifiedDoc() {  
    logfile.info("Returning from: " + thisJoinPoint);  
}
```

This advice is only interested in the operation performed and when returned from that operation, related logging information is written to the log file. To achieve flexibility of adjusting the format and pattern of the log information, another open source project log4j [Log04] is used.

The following shows several lines generated from a test run:

```
2004-12-19 00:41:42,510 dom4j.aspect.ActivityTrack recorded: User in UserGroup: Editor  
    is making the following modification: Returning from: call(Document  
org.dom4j.DocumentHelper.createDocument())  
2004-12-19 00:41:42,620 dom4j.aspect.ActivityTrack recorded: User in UserGroup: Editor  
    is making the following modification: Returning from: call(Element  
org.dom4j.Element.addElement(String))  
2004-12-19 00:41:42,620 dom4j.aspect.ActivityTrack recorded: User in UserGroup: Editor  
    is making the following modification: Returning from: call(void  
org.dom4j.tree.AbstractElement.addAttribute())  
2004-12-19 00:41:42,620 dom4j.aspect.ActivityTrack recorded: User in UserGroup: Editor  
    is making the following modification: Returning from: call(Element  
org.dom4j.Element.addAttribute(String, String))
```

2004-12-19 00:41:42,680 dom4j.aspect.ActivityTrack recorded: User in UserGroup: Editor  
is making the following modification: Returning from: call(Element  
org.dom4j.Element.addText(String))

## 4.4 Example 2: Several aspects in one application

After illustrating that it is possible to use AOP techniques to deal with member activity tracking as aspect, we want to construct a more general example. In this example, we will read an XML file – a `periodic_table` to create a `dom4j` document, display it with a `JTable` GUI. The definition of the table is given using an XML descriptor document, this descriptor document give instructions to indicate which element in the `periodic_table` should be displayed. We assign each “ATOM” in the `periodic_table` with user group category randomly. In this example, access control, member activity tracking, XPath, annotation, personalized view generation, and dynamic behavior capture are dealt with using AOP techniques.

### 4.4.1 Access control aspect

After the user group category is associated with the document and its components, appropriate implementation of the access control is important to the application.

In `dom4j`, the document and its components are implemented as classes, trying to implement access control inside the classes is messy and hard to maintain. In our case, we use AspectJ’s pointcut feature to capture the critical operations when the method call happens and before the execution of the method, and handle the access control by advices.

#### 4.4.1.1 Pointcut:

The first thing to implement the access control functionality is to capture the operations through AspectJ’s pointcut. For the access control aspect, all the modification operations on

the document should be captured, and then handled according to the users' group category and accessibility to the document and its components.

```
pointcut modifyDoc(): call(static * DocumentHelper.create*(..))
    || call(void Element.add(..))
    || call(Element Element.add*(..))
    || call(void Element.appendAttributes(..))
    || call(Element Element.createCopy(..))
    || call(boolean Element.remove(..));
```

#### 4.4.1.2 Cut through many classes and operations:

As in the member activity tracking aspect in example 1, this pointcut also captures DocumentHelper class and all the classes which implement the Element interface. Any attempts that trying to modify the document is captured through the use of wildcard "\*" and "..", this simple pointcut captures many operations to the components of the document. The "create\*" captures all the create operations in the components of the document, which include create: Attribute, CDATA, Comment, Document, Element, Entity, Entity, Namespace, Pattern, ProcessingInstruction, QName, Text and XPath. The wildcard ".." represents all the methods with the same name and different number and type of parameters. The wildcard "\*" in *static \** represents different return type of the methods captured.

#### 4.4.1.3 Advice:

To provide different reactions to different groups of users, after capturing the modification operation through the above pointcut, different behaviors are defined through the around advice. The variable currentUserGroup in this aspect helps to decide which user group is trying to execute this operation.

For the users from the "Browser" category, the operation is not allowed, so the operation is not performed but prints the following message instead.

```
void around(): modifyDoc() && (currentUserGroup == "Browser"){
```

```

        System.out.println("Users from Browser category are not allowed to
        make modification to the document.");
    }

```

This advice make it possible that the users from the “Browser” category can only view the document but can not make any modification to the document. If they try to call any of the methods defined in the modifyDoc() pointcut, the message is displayed and the operation is not performed.

For the users from the “Notation” category, only the modification to the comment is allowed. Through modification to the comment part of the document, users can make annotations on the document. Other modifications on the document are not allowed.

```

void around(): modifyDoc() && (currentUserGroup == "Notation") &&
    (! call(static Comment DocumentHelper.createComment(..))
    || ! call(Element Element.addComment(..)) {
    System.out.println("Users from Notation category can only make
    Comment to the document.");
}

```

This advice makes it possible that the users from the “Notation” category can view the document and make comments on the document. If they try to call any of the methods other than the methods related to “Comment” defined in the modifyDoc() pointcut, the message is displayed and the operation is not performed.

For the users from the “Editor” category, all the modification operations can be performed, there is no advice needed.

#### 4.4.2 Member Activity Tracking aspect

The member activity tracking aspect for this example is the same as in example 1. This is another benefit provided by AOP techniques, different applications can share the knowledge when they are dealing with the same base classes and expect the same kind of effect.

### 4.4.3 View generation aspect

Now that the components of the document are associated with different user groups, we can display different content in the table based on the user that accesses the document and the associated user group.

#### 4.4.3.1 Pointcut:

The first thing to implement this functionality as an aspect is to decide what operations to capture through AspectJ's pointcut construct. For this aspect, we choose to capture the constructor of the model to be displayed. The purpose of this pointcut is to capture the argument of this constructor, and then perform the select operation on the argument and only the contents satisfying the user group access privilege left to be displayed. We use *around* advice to handle this pointcut.

```
pointcut selectAttr(Document doc, Object obj):call ( XMLTableModel.new(
    Document, Object)) && args(doc) && args (obj);
```

#### 4.4.3.2 Advice:

After obtain the “document” argument, we use `currentUserGroup` information to construct an Xpath expression, then we use this Xpath expression to select the appropriate components from the document and save the nodes to an object list, then proceed to the operations related to the new object list.

```
void around(Document doc, Object obj) : selectAttr(doc, obj) {
    String text = "";
    if (currentUserGroup == "Browser")
        text = "//ATOM[USERGROUP='" + currentUserGroup + "']";
    else if (currentUserGroup == "Notation")
        text = "//ATOM[USERGROUP !='Editor']";
    else
        text = "";
}
```

```

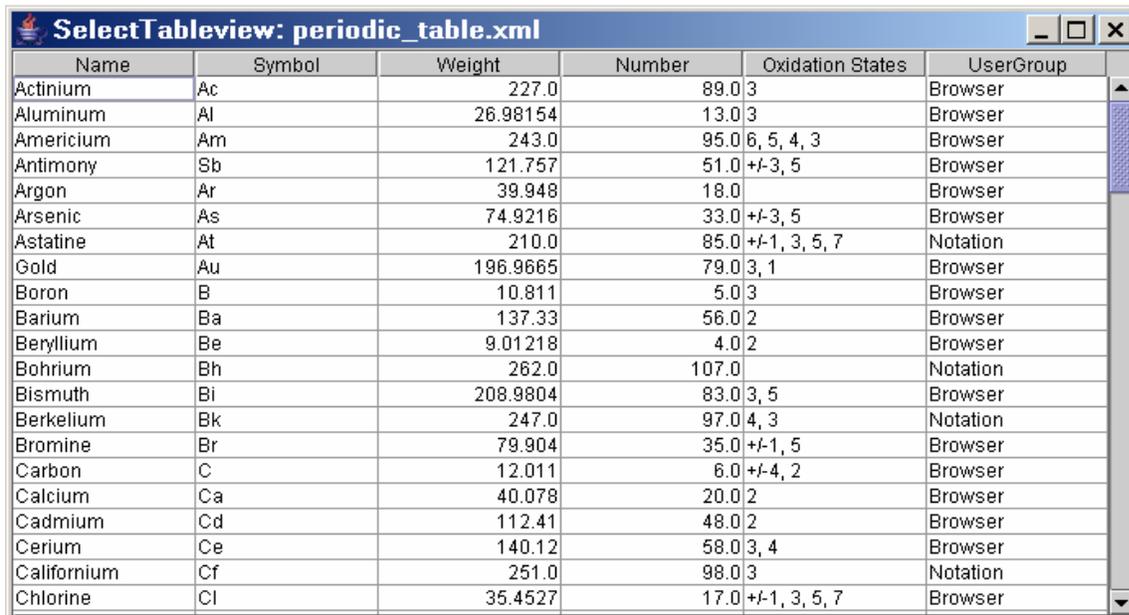
XPath xpath=DocumentHelper.createXPath(text);

List list = xpath.selectNode(document);

try {
    proceed(doc, list);
} catch (Exception e){
}
}

```

For the users in the “Browser” group category, only the component with the UserGroup attribute is “Browser” can be displayed. For the users in the “Notation” group category, the components with UserGroup attribute are “Browser” or “Notation” can be displayed, in other words, only the components with UserGroup attribute is “Editor” can not be displayed. For the users in the “Editor” group category, everything should be displayed, so the select xpath text is “\*”.



Name	Symbol	Weight	Number	Oxidation States	UserGroup
Actinium	Ac	227.0	89.0	3	Browser
Aluminum	Al	26.98154	13.0	3	Browser
Americium	Am	243.0	95.0	6, 5, 4, 3	Browser
Antimony	Sb	121.757	51.0	+/-3, 5	Browser
Argon	Ar	39.948	18.0		Browser
Arsenic	As	74.9216	33.0	+/-3, 5	Browser
Astatine	At	210.0	85.0	+/-1, 3, 5, 7	Notation
Gold	Au	196.9665	79.0	3, 1	Browser
Boron	B	10.811	5.0	3	Browser
Barium	Ba	137.33	56.0	2	Browser
Beryllium	Be	9.01218	4.0	2	Browser
Bohrium	Bh	262.0	107.0		Notation
Bismuth	Bi	208.9804	83.0	3, 5	Browser
Berkelium	Bk	247.0	97.0	4, 3	Notation
Bromine	Br	79.904	35.0	+/-1, 5	Browser
Carbon	C	12.011	6.0	+/-4, 2	Browser
Calcium	Ca	40.078	20.0	2	Browser
Cadmium	Cd	112.41	48.0	2	Browser
Cerium	Ce	140.12	58.0	3, 4	Browser
Californium	Cf	251.0	98.0	3	Notation
Chlorine	Cl	35.4527	17.0	+/-1, 3, 5, 7	Browser

**Figure 4-2: Views for select the user group of Notation**

With this simple aspect construct, we separate the details on selecting the appropriate content to display to an aspect, this also makes later modification easier. This pointcut

capture the argument to generate the table, other kind of filter criteria can be added here easily and do not affect the main program.

Figure 4-2 shows a screen shot of running the case when the `currentUserGroup` is Notation for the XML file `periodic_table` and associates the ATOM with user group information.

#### 4.4.4 Annotation to the document and XPath

The annotation is a very important feature for a document process application. DOM “Comment” construct can be used to hold the user’s annotation to the document and its components. Because the document and its components are associated with user groups, different user group have different authorized levels to make annotations on the document. In case 1, when we handle the access control aspect, we also define who can make annotations on the document; the users from the “Browser” group category can not make “Comment” to the document and its components. The users from the “Notation” group category can make “Comment” to the document and its components but cannot make any other modifications to the document. The users form the “Editor” group category can make any modification to the document and its components. In case 3, different views are generated dynamically according to the user group information, this makes the personalized view of the document possible. The user from the “Notation” group can make some annotation, this annotation is associated with the components belonging to this user group, and will not be viewed by the users from the “Browser” group.

XPath is a language for addressing parts of an XML document. We can load XML into a DOM, and then easily query the structure with XPath to extract the data. In case 3, we use `xpath` to select nodes from the document with the node satisfying the specified criteria. The `xpath` also can be dynamically constructed (take input from the user at run time), and be used to select contents from the XML document. The contents selected by the `xpath` expression are stored in an object list; this list can be further processed.

In our case, we can generate the personalized view with help from `xpath`. If the document and its components are associated personal user id, we can use the `userid` to construct `xpath`

expression, then select relevant contents from the document and display it. We can change the selectView aspect in case 3, add the userid information to the selection criteria.

```
else if (currentUserGroup == "Notation")
    text = "//ATOM[UserGroup == 'Browser'] |
//ATOM[UserId=='currentuserid']";
```

This will select the contents with the group category “Browser” and the contents with the group category “Notation” and userid is the same as the current user.

#### 4.4.5 Capture the dynamic operation as aspect

The AOP technique is designed to capture the method call or method execution, in another word, it captures the dynamic operation of the application. Object-oriented programming uses the event handling technique to handle this kind of dynamic operation. In the following section, we will give an example of an event, and shows how to handle it with OO technique and AOP technique.

##### 4.4.5.1 Example:

In dom4j, when a very large XML document is processed, the XML file is parsed and some parts of the document are pruned after processing to avoid having to keep the entire document in memory. For example, to process a very large XML file that is generated externally by some database process and looks something like the following (where N is a very large number).

```
<ROWSET>
  <ROW id="1">
    ...
  </ROW>
  <ROW id="2">
    ...
  </ROW>
  ...
  <ROW id="N">
    ...
```

```
</ROW>
</ROWSET>
```

Each <ROW> can be processed at a time; and we do not need to keep all of them in memory.

#### 4.4.5.2 OO implementation in dom4j:

dom4j provides an Event Based Mode for this purpose. First register an event handler for one or more path expressions. These handlers will then be called on the start and end of each path registered against a particular handler. When the start tag of a path is found, the onStart method of the handler registered to the path is called. When the end tag of a path is found, the onEnd method of the handler registered to that path is called. The onStart and onEnd methods are passed an instance of an ElementPath, which can be used to retrieve the current Element for the given path. If the handler wishes to "prune" the tree being built in order to save memory use, it can simply call the detach() method of the current Element being processed in the handlers onEnd() method.

So to process each <ROW> individually we can do the following.

```
// enable pruning mode to call back as each ROW is complete
SAXReader reader = new SAXReader();

reader.addHandler( "/ROWSET/ROW", new ElementHandler() {

    public void onStart(ElementPath path) {

        // do nothing here...

    }

    public void onEnd(ElementPath path) {

        // process a ROW element

        Element row = path.getCurrent();

        Element rowSet = row.getParent();

        Document document = row.getDocument();

        ...

        // prune the tree

        row.detach();
    }
}
```

```

    }
}
);

Document document = reader.read(url);

// The document will now be complete but all the ROW elements
// will have been pruned.

```

#### 4.4.5.3 AOP solutions:

From the above OO solution, we can see there are three steps to deal with this problem. First use the `addHandler` operation to capture the event, (the event happens when the specified path encountered); second, call `onStart` method to define what to do before the start of dealing the content (do nothing in this example); third, call `onEnd` method to define what to do after dealing with the content (detach the element from the tree). The same process also can be designed as a perfect AOP example, by three steps. First, use `pointcut` to capture the when the parser processes a certain path; second, use `before()` advice to define what to do to the contents before the start of dealing the content, if do nothing, we can skip the definition of this before advice; third, use `after()` advice to define what to do after dealing with the content. The AOP technique also provides the option of use the `around()` advice to provide other options without making lots of changes to the source code.

So we can define the aspect as the following.

```

Aspect SelectContent{

    pointcut selectPath(ElementPath path): call (* reader.read(..)

        && args(path);

    after(ElementPath path):selectPath(path) {

        // process a ROW element

        Element row = path.getCurrent();

        Element rowSet = row.getParent();

        Document document = row.getDocument();

        ...
    }
}

```

```
        // prune the tree
        row.detach();
    }
};
```

This is only one possible solution to this kind of problems. AOP techniques can be used to capture the operations dynamically and handle them separately; AOP can be a good substitute for the OO design's event handling mechanism.

#### 4.4.6 More dynamic operations captured as aspect

In this chapter, we use two examples to illustrate the possibility of applying AOP techniques to dom4j – a java implementation API for DOM. We select several cases to show different aspects of the application, these are not all the cases that can be applied to a complicated application. There are many other aspects that can be dealt with similarly, such as dealing with the exception handling as an aspect, or extending the capture of the dynamic operation to handle more dynamic behaviors by implementing them as an aspect instead of as an event.

For example, we can further extend the application of example 2, add interactive mouse operations to the GUI, either by OO event handling or by capture mouse operation and define the behaviors through aspect. Then we can put the aspects in example 2 together as the following scenario:

- When the user tried to load the table, user group information is checked and the select view aspect is called when the table is constructed. Only appropriate information is displayed to the user.
- Mouse single-click selects the attributes of the ATOM selected and modification to the value of the attribute can be performed. Through this modification call, user group category is checked by access control aspect, unauthorized modification is not preceded and warning information is given. Authorized modification is performed and member activity is logged by the member activity tracking aspect.

- Mouse double-click selects the ATOM node, modifications are also monitored by the aspects as in Mouse single-click. Users in the Notation group can make personal annotations to the ATOM too.

From the case studies, we conclude that with the help of AspectJ, we can apply AOP techniques to dom4j. These successful running cases prove that our attempts to apply AOP technique to document models are practical.

One problem for the examples is that the version of AspectJ integrated in Eclipse, which we used to compile our examples, is not stable. Some times it compiles successfully, some times not. No compiling error is given, and for different installation in different machines, the running result for the compiled example is different.

# Chapter 5

## Conclusions

### 5.1 Summary

In this thesis, we have analyzed the hypermedia document model and extended the model. A general hypermedia document model is specified and AOP techniques are applied to it. We first define the static components of the Aspect based document model, and then we specify the dynamic operations of each components of the model. These dynamic operations are captured by AOP constructs and dealt with by AOP advices.

After the general model is specified, we give descriptions on where and how to weave AOP techniques into the model to solve different concerns. Then we apply this AOP technique to a subset of the general model – DOM, with emphasis on dealing with the contents and structure of the document.

In our case study, we choose dom4j as the base implementation of the document model – DOM. We use AspectJ as the AOP programming language to implement the aspect. The result of the case study shows that the methodology described in the general model is practical.

### 5.2 Lessons learned

When we define the general model and apply AOP techniques to the dynamic operations of the model components, we find that there are two kinds of concerns in this general document model.

One kind of concern is the system level concerns that are difficult to solve by classic OO techniques, such as the access control, member activity tracking, exception handling etc. This kind of concern is dispersed throughout the whole system, every component has to provide implementation for this kind of concern, but the behavior of the components is almost always the same. This property makes this kind of concern a perfect candidate for AOP implementation. The related operations are captured and dealt with in one aspect. There are many benefits of dealing with this kind of concerns this way.

- The system is better modeled, each component does not implement this concern inside its class, but the entire concern is implemented in one aspect. This way, the concern is stored in one place. When modification to this concern is needed, changes are in one place – the aspect, people do not have to look through the whole system.
- Each component class is simpler, the implementations to this kind of concerns are removed, and the class can concentrate on the major functionality of the components. This makes the code easier to understand. It is also easier to maintain system consistency.

The other kind of concern is use AOP technique to capture and deal with dynamic operations of the components. Some operations need preparation before the action is performed, and some operations can affect other components. This kind of operations can be handled by AOP technique. For example, when we delete a node, the links associated with this node should be found first, this can be dealt with using AOP *before* advice. After the node is deleted, the links associated to this node should also be updated, the link target and source information related with this node have to be updated to keep the system consistent, this can be done by the AOP *after* advice.

In OO methodology, the interactive dynamic operations of the components are dealt with by event handling mechanism. The AOP techniques also provide a way to deal with this kind of interactive dynamic operations. More studies are required to determine the benefit of using

AOP techniques to handle the interactive dynamic operation instead of the OO event handling mechanism.

### 5.3 Future Work

While AOP techniques have many advantages in dealing with the crosscutting concerns, the AOP technique also violates the well encapsulation discipline imposed by the OO techniques. In one aspect construct, many classes and operations may be involved; this might cause other unanticipated problems while solving the crosscutting concerns. Some features, such as introducing a new field and method to the class inside the aspect, that we do not discuss in our paper are very powerful. It is useful but we have to use these features with care. If we introduce too many fields and methods inside the aspect, the system structure will break because this will violate the class's integrity. The AOP techniques are still under development, rules are needed to define the following topics:

- What kind of questions is best solved by AOP techniques?
- When is it appropriate to apply AOP technique?
- What good practices can be developed to specify the kind of operations to capture when defining the aspect?

There are several directions to proceed for future work.

- Identify and separate more aspects in the model.
- More studies considering the conditions to switch from OO event handling mechanism to AOP techniques for dealing with the interactive dynamic operations.
- The case study shows several standalone prototypes of how to implement aspect in AspectJ to DOM implementation – dom4j. Other programming languages other than AspectJ should be tested when they are available.

- Other features of the general model that are not focused by DOM, such as links and anchors, also should be tested.

# Appendix A

## Sample Code

### ActivityTracking.java

```
package org.dom4j.aspect;

import org.apache.log4j.*;
import org.dom4j.DocumentHelper;
import org.dom4j.Element;

public aspect ActivityTracking {

    static Logger logger = Logger.getLogger(ActivityTrack.class.getName());

    pointcut modifiedDoc(): call(static * DocumentHelper.create*(..))
        || call(* Element.add*(..))
        || call(void Element.appendAttributes(..))
        || call(Element Element.createCopy(..))
        || call(boolean Element.remove(..));

    after() returning: modifiedDoc () {
        logger.info("Returning from: " + thisJoinPoint);
    }
}
```

**ActivityTrack.java**

```
package org.dom4j.aspect;

import java.io.FileWriter;
import java.util.Enumeration;
import java.util.Properties;
import org.dom4j.samples.*;
import org.apache.log4j.*;

import org.dom4j.Document;
import org.dom4j.DocumentHelper;
import org.dom4j.Element;
import org.dom4j.io.OutputFormat;
import org.dom4j.io.XMLWriter;

public class ActivityTrack extends AbstractDemo {
    static Logger logger = Logger.getLogger(ActivityTrack.class.getName());

    public static String currentUserGroup = "Editor";

    public static void main(String[] args) {
        setupLogger();
        run( new ActivityTrack(), args);
    }

    public ActivityTrack() {
    }

    public static void setupLogger() {
        String pattern = "%d{ISO8601} %c{3} recorded: User in UserGroup : " +
```

```
        currentUserGroup + "%n";
    pattern += "    is making the following modification: %m %n %n";

    PatternLayout layout = new PatternLayout(pattern);
    FileAppender ap = null;
    try {
        ap = new FileAppender(layout, "AspectActivityTrack.log", false);
    } catch (Exception e) {}

    logger.addAppender(ap);
    logger.setLevel((Level) Level.DEBUG);
}

public void run(String[] args) throws Exception {
    Document document = createDocument();
    OutputFormat format = new OutputFormat(" ", true);

    if ( args.length < 1 ) {
        XMLWriter writer = new XMLWriter( System.out, format );
        writer.write( document );
    }
    else {
        String fileName = args[0];
        println( "Writing file: " + fileName );
        FileWriter out = new FileWriter( args[0] );
        XMLWriter writer = new XMLWriter( out, format );
        writer.write( document );
        out.close();
    }
}
```

```
protected Document createDocument() throws Exception {
    Document document = DocumentHelper.createDocument();
    Element root = document.addElement( "system" );
    int i=0;

    Properties properties = System.getProperties();
    for (Enumeration enum=properties.propertyNames(); enum.hasMoreElements();){
        String name = (String) enum.nextElement();
        String value = properties.getProperty( name );
        Element element = root.addElement( "property" );
        element.addAttribute( "name", name );
        element.addText( value );
        document.addComment("Comment"+ i);
        i++;
    }
    return document;
}
}
```

### **AccessControl.java**

```
package org.dom4j.aspect;

import org.apache.log4j.*;
import org.dom4j.DocumentHelper;
import org.dom4j.Element;

public aspect AccessControl {

    pointcut modifyDoc (): call(static * DocumentHelper.create*(..))
        || call(* Element.add*(..))
```

```
        || call(void Element.appendAttributes(..))
        || call(Element Element.createCopy(..))
        || call(boolean Element.remove(..));

void around(): modifyDoc() && (currentUserGroup == "Browser"){
    System.out.println("Users from Browser category are not allowed to
    make modification to the document.");
}
void around(): modifyDoc() && (currentUserGroup == "Notation")
    && (! call(static Comment DocumentHelper.createComment(..))
    || ! call(Element Element.addComment(..))) {
    System.out.println("Users from Notation category can only make
    Comment to the document.");
}
}
```

### SelectAspectView.java

```
package org.dom4j.aspect;

import java.util.List;
import org.dom4j.XPath;
import org.dom4j.Document;
import org.dom4j.DocumentHelper;
import org.dom4j.io.XMLWriter;
import org.dom4j.swing.XMLTableModel;

public aspect SelectAspectView {

    protected String currentUserGroup = "Browser";
```

```
pointcut selectAttr(Document doc, Object obj):call ( XMLTableModel.new(
    Document, Object)) && args(doc) && args (obj);

void around(Document doc, Object obj) : selectAttr(doc, obj) {
    String text = "*";
    if (currentUserGroup == "Browser")
        text = "//ATOM[USERGROUP='" + currentUserGroup + "']";
    else if (currentUserGroup == "Notation")
        text = "//ATOM[USERGROUP !='Editor']";
    else
        text = "*";
    XPath xpath = DocumentHelper.createXPath(text);
    List list = xpath.selectNodes( obj );
    try {
        proceed(doc, list);
    } catch (Exception e) {}
}
}
```

### SelectTableView.java

```
package org.dom4j.aspect;

import java.util.List;

import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import org.dom4j.XPath;
import org.dom4j.Document;
```

```
import org.dom4j.DocumentHelper;
import org.dom4j.io.SAXReader;
import org.dom4j.swing.XMLTableModel;

public class SelectTableView {
    public static void main(String[] args) throws Exception {
        SelectTableView sample = new SelectTableView();
        sample.run(args);
    }

    public void run(String[] args) throws Exception {
        if ( args.length <= 1 ) {
            System.out.println( "Usage: <tableXMLDescription> <xmlFileName>" );
            System.out.println();
            System.out.println( "This example will display the periodic table in a JTable" );
            System.out.println( "  java aspect.SelectTableView tableForAtoms.xml
                                periodic_table.xml" );

            return;
        }
        // parse document
        SAXReader reader = new SAXReader();
        Document definition = reader.read( args[0] );
        Document document = reader.read( args[1] );

        XMLTableModel model = new XMLTableModel( definition, (Object) document );
        // make the widgets
        JTable table = new JTable( model );

        JFrame frame = new JFrame( "SelectTableview: " + document.getName() );
        frame.setSize(300, 300);
    }
}
```

```
frame.setLocation(100, 100);
frame.getContentPane().add( new JScrollPane( table ) );
frame.validate();
frame.setVisible(true);
}
}
```

## Bibliography

- [AC03] AspectC#, [http://www.dsg.cs.tcd.ie/index.php?category\\_id=168](http://www.dsg.cs.tcd.ie/index.php?category_id=168) 2003
- [AC04] AspectC++, <http://www.aspectc.org/> 2004
- [AEB03] O. Aldawud, T. Elrad, and A. Bader, UML profile for aspect-oriented software development. *Proceedings of Third International Workshop on Aspect-Oriented Modeling*, March 2003
- [BS03] M. Basch, and A. Sanchez, Incorporating aspects into UML, *Proceedings of Third International Workshop on Aspect-Oriented Modeling*, March 2003
- [BSA04] BEA Systems AspectWerkz, <http://aspectwerkz.codehaus.org/> 2004
- [CHD99] L. Carr, W. Hall, and D. De Roure, The evolution of hypertext link services. *ACM Computing Surveys*, 31(4), Dec. 1999.
- [Chen76] P.P. Chen, The entity-relationship model: Toward a unified view of data. *ACM Trans. Database System*. 1(1)9-36, 1976.
- [CNML92] T.T. Carey, R.B. Nennecke, J. Mitterer, D. Lungu, Prospects for Active Help in Online Documentation, *ACM SIGDOC Conference*, October 1992.
- [DAP97] P. Diaz, I. Aedo, and F. Panetsos, Labyrinth, An Abstract Model for Hypermedia Applications. Description of Its Static Components, *Information Systems*, 22(8)447-464, 1997.
- [DAP01] P. Diaz, I. Aedo, and F. Panetsos, Modeling the Dynamic Behavior of Hypermedia Applications, *IEEE Transactions on Software Engineering*, 27(6)550-572, June 2001.
- [DG00] D. M. German. Hadez: a Framework for the specification and verification of hypermedia applications. PHD thesis, 2000.
- [Ecl04] IBM Eclipse project. <http://www.eclipse.org/> 2004
- [EFB01] T. Elrad, R. Filman, A. Bader, Aspect-Oriented Programming, *Communications of the ACM*, 44(10): 29-32, October 2001.

- [FGR03] R. France, G. Georg, and I. Ray, Supporting multidimensional separation of design concerns, *Proceedings of Third International Workshop on Aspect-Oriented Modeling*, March 2003
- [GMP95] F. Garzotto, L. Mainetti, and P. Paolini. *Designing User Interfaces for Hypermedia*, chapter Hypermedia Application Design: a structured design. *Springer Verlag*, 1995.
- [GPS93] F. Garzotto, P. Paolini, and D. Schwabe, HDM-a model-based approach to hypertext application design. *ACM Transactions of Information System*. 11(1)1-26, Jan. 1993
- [GPS94] F. Garzotto, P. Paolini, and D. Schwabe, Adding multimedia collections to the Dexter model. *In Proceedings of the 1994 ACM Conference on Hypermedia Technology*. 1994.
- [Garg88] P. K. Garg, Abstraction mechanisms in hypertext. *Communications of the ACM*, 31(7):862-870, 1988
- [Hac97] J.T. Hackos, Online Documentation: The Next Generation, *Proceedings of the 15<sup>th</sup> Annual Conference on Computer Documentation*, pp. 99-104, October 1997.
- [HBV94] L. Hardman, D. C. A. Bulterman, and G. Van Rossum, The Amsterdam hypermedia model: Adding time and context to the Dexter model. *Communications of ACM*, 37(2)50-62, Feb. 1994.
- [HS94] F. Halasz and M. Schwartz, The Dexter hypertext reference model. *Communications of ACM*, 37(2)30-39, Feb. 1994.
- [IH04] IBM HyperJ <http://www.alphaworks.ibm.com/tech/hyperj>, 2004
- [ISB95] T. Isakowitz, E. Stohr, and P. Balasubramanian. RMM: A Methodology for structuring hypermedia design. *Communications of The ACM*, 38(8)34-44, August 1995.
- [Kess95] M. Kessler, A schema-based approach to HTML authoring. *In Proceedings of the Fourth International Conference on The World Wide Web* Boston, 1995.
- [KHH01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W.G. Griswold, An Overview of AspectJ, *15th European Conference on Object-Oriented Programming*, pp. 327-353, Springer-Verlag, 2001.

- [Kic96] G. Kiczales, Aspect-Oriented Programming, *ACM Computing Surveys*, 28, p. 154, 1996.
- [Lan96] D. Lange. An Object-Oriented Design Approach for Developing Hypermedia Information Systems. In *Journal of Organizational Computing*, 1996.
- [LN04] LOOM.Net <http://www.ajlopez.net/ItemVe.php?Id=996> 2004
- [Log04] Log4j. <http://logging.apache.org/log4j/docs> 2004
- [Lop03] C.V. Lopes, *AOP: A Historical Perspective*, Addison-Wesley, 2003.
- [LP04] LivePage Enterprise. <http://www.livepage.com> 2004
- [MA93] K. H. Madsen and P. H. Aiken, Experiences using cooperative interactive storyboard prototyping. *Communications of ACM*, 36(6)57-64 June, 1993.
- [Mau96] H Maurer *HyperWave - The Next Generation Web Solution*, Addison-Wesley, 1996.
- [My95] B. A. Myers, User interface software tools. *ACM Trans. Comput. Hum. Interact.* 2(1)64-103, Mar. 1995.
- [Niel95] Jakob Nielsen *Multimedia and Hypertext: The Internet and Beyond*, Morgan Kaufmann, 1995.
- [NCE03] P. Netinant, C.A. Constantinides, T. Elrad, M.E. Fayad, and A. Bader, Supporting the design of adaptable operating system using aspect-oriented frameworks, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 2000, Vol.1 (271-277).
- [PPF02] M. Pinto, L. Fuentes, M. Fayad, and J. Troya, Separation of coordination in a dynamic aspect oriented framework, *Proceeding of the First International Conference on Aspect-Oriented Software Development*, April 2002.
- [RBPE91] J.Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, B. Lorensen, AND W. Lorensen, Object Oriented Modeling and Design. *Prentice-Hall, Englewood Cliffs, NJ*. 1991.
- [SF89] P. D. Stotts and R. Furuta, Petri-netbased hypertext: Document structure with browsing semantics. *ACM Trans. Inf. Syst.* 7(1)3-29, Jan. 1989.

- [SR95] D. Schwabe, G. Rossi The Object-Oriented Hypermedia Design Model. *Communications of the ACM*, 38(8) 45- 46 Aug. 1995.
- [SRB95a] D. Schwabe, G. Rossi and S. Barbosa. Abstraction, Composition and Lay-out Definition Mechanisms in OOHDM. In *Proceedings of ACM Workshop on Effective Abstractions in Multimedia*, 1995.
- [SRB95b] D. Schwabe, G. Rossi and S.D.J. Barbosa. Systematic Hypermedia Application Design with OOHDM, in *Proc. Hypertext 96*, ACM Press, 1996.
- [W3C04] W3C Document Object Model (DOM). <http://www.w3.org/DOM/> 2004
- [WN03] Weave.Net [http://www.dsg.cs.tcd.ie/index.php?category\\_id=193](http://www.dsg.cs.tcd.ie/index.php?category_id=193) 2003
- [XP04] Xerox PARC AspectJ. <http://eclipse.org/aspectj/> 2004
- [ZFCC01] M. Zachry, B.D. Faber, K.C. Cook, D. Clark, The Changing Face of Technical Communication: New Directions for the Field in a New Millennium, *Proceeding of the ACM SIGDOC'01 Conference, Annual ACM Conference on Systems Documentation*, pp. 248-260, 2001
- [ZP92] Y. Zheng and M. C. Pong, Using statecharts to model hypertext. In *Proceedings of the ACM Conference on Hypertext*, 1992.