Addendum

p 146 line 7: “four” is substituted by “six”.

p 168: The following paragraph is inserted at the end of Section 10.3:

> “With regards to the evolutionary mechanism of application systems on the Web, how to trace and capture the document changes on the Web is a challenge topic. Further work can be done in this area to improve the evolutionary mechanism of these systems.”

p 117: The following section is inserted after Section 8.1.4:

8.1.5 Evolution Algorithm

This section develops an algorithm for the Life Design evolutionary process described previously. For the purpose of this algorithm, we only model class genes and life forms as object identities, not their structure. So let *OId* be a set of names (object identities) and let *GId* be a set of gene names (gene identities).

The function

$$genome: OId \rightarrow Set(GId)$$

associates each organism with a set of genes.

If $genome(O) = \{g_1, g_2,..., g_n\}$ then we also write $O.g_1, O.g_2,..., O.g_n$ to indicate that O possesses the genes $g_1, g_2,...,g_n$.

Moreover if $G =$ $genome(O)$ then we call G the *main class genome* of O (see Sections 6.4.1.2 and 6.4.2.2.2).

Let

$$organisms: Set(GId) \rightarrow OId$$

such that:

$$organisms(G)= \{O \in OId \mid G = genome(O)\}$$

The function *addGene(G, g)* adds gene g to G, and the function *removeGene(G, g)* removes g from G, using the self-modification functions described in Section 6.4.2.4.3.

Let T_g ($0 \leq T_g \leq 1$) be an arbitrary evolutionary threshold applicable to the presence of gene g in a set of organisms, and T_{-g} ($1 - T_g \leq T_{-g} \leq 1$) be an arbitrary evolutionary threshold applicable to its absence (see Section 6.4.2.4.4).

At each meta-level, the Evolve algorithm is defined for $O \in OId$ and $g \in GId$, such that:

```
Evolve $(O, g)$ =
        $G = genome(O);$
        $L = organisms(G);$
        $L_g = \{ O \in L \mid O.g \};$
        if $(g \notin G$ and $\frac{|L_g|}{|L|} \geq T_g)$ then
                $addGene(G, g);$
        else if $(g \in G$ and $\frac{|L \setminus L_g|}{|L|} > T_{-g})$ then
                $removeGene(G, g);$
        end
end
```

An Object-Oriented Data Model for Evolvable Web Systems

by

Thuy-Linh Nguyen, BA (Hon)

Dissertation

Submitted by Thuy-Linh Nguyen

for fulfilment of the Requirements

for the Degree of

Doctor of Philosophy

In the School of Computer Science and Software Engineering at

Monash University

Monash University

December 7, 2000

Contents

List of Tables

List of Figures

An Object-Oriented Data Model for Evolvable Web Systems

Thuy-Linh Nguyen, PhD

Monash University, 2000

Supervisor: Heinz Schmidt

Abstract

The exponential growth of the Web in a very short time has resulted in many deployment problems that stem from inadequate system design. Some major ones are: data management and maintenance, resource discovery, referential integrity, customisability, adaptability, and evolvability. Many projects have attempted to correct these problems, but most are adhoc and suffer from the same design inadequacy. Others have tried to tackle these problems directly at the design level, and have the potential to correct some of the problems. However, even in these projects, some issues are only partially addressed, in particular, adaptability and evolvability are still unchecked. The enormous growth of the Web necessitates an infrastructure that can capture its dynamic characteristic and cope with future changes.

The principal contributions of this thesis are *Life Design* and *LifeWeb*, a design and a data model for a manageable, maintainable, customisable, adaptable and evolvable (self-evolving) system. *LifeWeb* also facilitates resource discovery and enhances referential integrity. *LifeWeb* is a data model for the Web document system, implementing *Life Design* and represented in XML (Extensible Markup Language). *Life Design* is an object-oriented system design methodology gleaned from the structure and working of organic life forms at the molecular, genetic and biological levels. Entities in *LifeWeb* possess biologically life-like functionalities, and can evolve themselves in a fashion similar to the Darwinian evolutionary process, driven by user needs or customisation requirements. These entities exist in a recursive architecture of multiple meta-modelling levels. Evolutionary changes are incrementally accumulated, conditionally and gradually propagated from lower to higher levels, forming groups of entities with increasingly divergent "genes". In *LifeWeb*, such a group corresponds to a document type or a "document species". The *LifeWeb* evolutionary model makes it possible to derive new document types (including supporting software and tools) from existing ones (which increases reusability, manageability, maintainability and extensibility), and to maintain their interoperability. A prototype system for *LifeWeb* has been implemented, which allows automatic system evolution, where entities can be dynamically added to or removed from the runtime system. *LifeWebManager*, a *LifeWeb* application and a tool for creating and managing *LifeWeb* documents, is also implemented.

We claim that our technology improves on the Web manageability, maintainability and customisability over existing Web data models, and lays a foundation for the Web adaptability and evolvability. Our claim is substantiated by deep comparison with existing Web data models.

An Object-Oriented Data Model for Evolvable Web Systems

Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

[redacted]

Thuy-Linh Nguyen

20 October 2000

Acknowledgements

Firstly, I would like to thank my supervisor, Professor Heinz Schmidt, who in the last year of my candidature tirelessly helped me to straighten up my thoughts and document my research results with his expert knowledge in many areas and his valuable advice. A special thank to Scott Milton, who allowed me to use his software HotServ, and to the students in the Industrial Experience Group 11, 1999, who implemented the *LifeWebManager* application. A sincere thank to Professor John Crossley, Dr Hai-Tan Tran, and Professor Luz Weigner and for the time they had generously given to proofread and give constructive feedback on my thesis.

My thanks are also extended to colleagues in the School of Computer Science and Software Engineering, who in one way or another had helped me during my candidature. In particular, I would like to thank my two former supervisors Associate Professor Xindong Wu and Associate Professor Sayed Sajeev, Associate Professor Christine Mingins, Harriet Searcy, Daniel Ngu, Dr Phu-Dung Le, Matt Mitchell, Dr Jason Lowder, Dean Thompson, See Hung Ngieng, Tom Pearcy, Michelle Ketchen, Monique Spence, Allison Muller, Solvejg Moyle, and Claire Gee.

A special thank to Rev Anthony Quang, for his loving support during my whole candidature; and to the community of the Daughters of St Paul, for their love, care and patience extended to me. Thank you to Mum and Dad, and all my brothers and sisters, whose love has always given me so much strength to try to achieve my dream.

Above all, thank be to God, who seeded the idea in my mind and is the true author of my whole project.

To my parents

Chapter 1 Introduction

1.1 Background

The World-Wide-Web (hereafter also referred to as Web or WWW) has its data model based on *hypertext*, a concept originated by Vannevar Bush [BUS45], Scientific Advisor to the U.S. President Roosevelt, back in 1945. Motivated by the need to assist people to access the huge amount (even then) of available data, Bush felt that it should be possible to design some mechanism that imitated the associative memory of the human brain. This imaginary system (which he named "Memex") should store items of information with associative links between them, and allow data to be accessed with "exceeding speed and flexibility" [BUS45]. The system was not in fact realised until there were sufficiently powerful computers available. The first hypertext system was implemented by Doug Engelbart [CON87] in the 1960s, as part of his AUGMENT/NLS system. In 1965, Ted Nelson [NEL65a & b] coined the term "hypertext" and began work on his vision of the *Xanadu* project [NEL87, NEL90], a universally accessible computer storage and retrieval system for documentation. Since then hypertext has become increasingly popular. Early milestones were the commercial support from Apple with the release of the *HyperCard* system,[1] and the first major global hypertext conference *Hypertext'87* [NIE88], both in 1987.

In 1989, a proposal was made by Tim Berners-Lee addressing an internal information management issue at CERN, the European Particle Physics Laboratory in Geneva, Switzerland [BER89]. Due to the high personnel turnover at CERN, information of past projects and events was constantly being lost, wasting a lot of effort in recovering the lost information or even "reinventing the wheels" altogether. There was a need to create "a pool of information which could grow and evolve with the organisation and the projects it describes" [BER89]. This idea brought Berners-Lee to discover the hypertext data model, which had been around for some time. With the need at CERN, this model found favour over the traditional database or hierarchical structure due to its simplicity and flexibility. It allows the recording of "random links" between "arbitrary nodes" without any constraint on data structure [BER89]. The Web was designed as a universal space of shared information where new relationships between objects could be added at any time without the need of any major or central control. "The information space will simply reshape to represent the new state of knowledge" [BER94].

After having obtained CERN's approval, Tim Berners-Lee started working on his first hypertext browser and editor, a program that he named "WorldWideWeb". In 1991 the first global hypertext system was released. At this stage only a line-mode browser was available. The Web started to grow exponentially [INT91, NSF95, GVU95, NGU96, ZAK96, ISC00] when *Mosaic*, the first GUI (graphical user interface) Web browser, developed by the National Center for Supercomputing Application (NCSA), became available in 1993. Very soon afterwards the Web became commercialised when Netscape released its browser in 1995 and gained almost three billion dollars on the first day of stock market trading.

This phenomenal success of the Web, however, has revealed many of its shortcomings that result from early engineering decisions. Some major ones (which will be analysed in Chapter 2) involve: data management and maintenance, resource discovery, referential integrity, customisability, adaptability, and evolvability. If these problems are left unchecked, the Web could eventually become unmanageable. Although much work has been done to improve the Web (some which relates to our work will be analysed in Chapter 3), most inherits the weaknesses of the Web infrastructure and addresses its problems in an adhoc or patchy manner. Recently, the advent of the Extensible Markup Language (XML) [XML98] and its associated standards has enabled the correction of some of these

[1] It has been argued that HyperCard is not really a full hypertext system. However it did popularise the concept [HAG94].

problems (for example, data management and maintenance, referential integrity, resource discovery, and extensibility) at the fundamental design level (see Sections 3.2.2, 3.2.3 and 3.3). However, some issues, especially those regarding adaptability and evolvability are still not addressed. The Web is a highly dynamic system and needs an infrastructure that can capture this dynamic characteristic and cope with future changes.

The principal contributions of this thesis are *Life Design* and *LifeWeb*, a design and a data model for a manageable, maintainable, customisable, adaptable and evolvable (self-evolving) system. *LifeWeb* also facilitates resource discovery and enhances referential integrity. *Life Design* is a concept drawing upon natural life at the molecular, genetic and biological levels. It imitates the structure and working of organic life forms at the molecular and genetic levels to construct entities with biologically life-like functionalities. These entities can evolve in a fashion similar to the Darwinian evolutionary process. The evolutionary mechanism is defined specifically in *Life Design* with a multi-layered meta-modelling, where changes are conditionally propagated from lower to higher levels to form groups of individuals with increasingly divergent "genes". This design, on which *LifeWeb*, a data model for the Web document system, is built, makes it possible for *LifeWeb* to address a range of issues that the Web is facing, such as those mentioned in the previous paragraph. A prototype system for *LifeWeb* has been implemented, and has shown very promising results. Both *Life Design* and *LifeWeb* use object-oriented technology. It may be possible, however, to apply *Life Design* to some other design technologies.

1.2 Research Questions

This study attempts to solve or alleviate some major problems with the Web at the design level by developing an object-oriented data model for the Web. In particular it addresses the following research questions:

RQ1 Is it possible to develop a data model for the Web document system that has the potential to solve or alleviate problems concerning data management and maintenance, resource discovery, referential integrity, customisability; and especially, adaptability and evolvability, which have not been addressed?

RQ2 More particularly, is it possible to apply object-oriented technology and biological concepts to that model?

In order to address these primary research questions, the thesis describes a design and a data model with its prototype implementation. As the design and the model must capture both the static and dynamic characteristics of the Web, the following sub-questions arise:

RSQ1 What are the essential *static* features of the Web that must be captured in the model? How can they be represented using object-orientation?

RSQ2 What are the essential *dynamic* features of the Web that must be captured in the model? How can they be represented using object-orientation and biological metaphors?

RSQ3 What are the evolutionary entities in the Web? What features are required of them, and what evolutionary mechanism and evolutionary path can be designed for them to evolve themselves?

1.3 Thesis outline

This thesis is organised as follows.

Chapter 2 provides an analysis of current major problems of the Web. It first summarises the main features of the Web, its architecture and the protocols upon which it operates. Problems with these features are identified and analysed at both the deployment and design levels, and the connections between problems at the two levels are established. This problem analysis confirms the problematic areas, points out where the roots of the problems are, and crystallises the solution direction: an object-oriented data model to solve problems at the design (root) level, that can capture, among other things, the dynamics of the Web.

Chapter 3 reviews existing trends in Web data models, pointing out what has and has not been addressed in the literature with regard to the issues presented in Chapter 2. These unsolved or partially solved problems are those of statelessness, customisability, and especially, adaptability and evolvability, which may also be used to tackle other problems. By reviewing existing data models, the chapter also makes remarks on their advantages and disadvantages, and the essential static features of the Web that should be captured (in a data model).

The evolvability issue requires a literature review on system evolution, which is provided in Chapter 4. This chapter describes how system evolution has been dealt with in Computer Science, particularly in the areas relating to our proposed solutions, namely data modelling, object-orientation, and biological metaphors. It gives insights into how the same issue may be addressed in the Web, and establishes a literature based on which we can compare our work on Web's evolvability with others' on system evolution.

Chapter 5, based on the remarks in Chapter 3, identifies the essential static elements of the Web, and describes our Multimedia Document Model (MDM), which encapsulates these elements. MDM captures the static and document characteristics of the Web, and is built on established standards in the publishing industry and library systems. It is the first stage of our two-stage system. Another model, which captures the dynamic characteristics of the Web, is constructed on top of it. The chapter also evaluates MDM against comparable document models.

Chapter 6 identifies the essential dynamic elements of the Web and considers different approaches to capture these elements. The chosen approach is presented in the main part of the chapter, which is conceptualised in biological metaphors and implemented by object-oriented technology. Evolutionary entities and their essential characteristics are identified. An evolutionary mechanism and path is defined for them. This approach is formalised in *Life Design*, a design methodology for building a self-evolving system.

Chapters 7 to 9 explain *LifeWeb*, the model of the second stage of our two-stage system. This model, which is built upon MDM and follows the methodology set out in *Life Design*, is described in Chapter 7. Chapter 8 explains how *LifeWeb* can evolve (itself). It also compares *LifeWeb* evolution with evolution in other systems that were reviewed earlier (in Chapter 4). Chapter 9 describes the system implementation, showing how XML, a recently emerging standard, has been integrated into *LifeWeb*. It also presents the implementation details that address the problems set out in Chapter 2, and when applicable, evaluates *LifeWeb* against other comparable systems that have concerns about the same issues.

Our conclusions are presented in Chapter 10.

1.4 Digest of results

Our problem analysis shows that most of the problems with the Web are rooted in its design. Owing to the lack of an appropriate model for the data it represents, the Web has outgrown its initial design and become unmanageable. We have designed a data model for the Web document system, named *LifeWeb*, which has the capacity to make improvements on this problem. This abstraction uses proven object-oriented technology to model the different components of the document and an innovative *Life Design* methodology to capture the dynamic characteristics of the Web.

In the *LifeWeb* prototype, documents can be designed, created, managed and maintained on structural, presentational, and content bases. A prototype tool has been implemented to perform these tasks using a simple "drag-and-drop paradigm". Users can customise online documents and *LifeWeb* is stateful so that these changes can be saved across sessions. Adaptation is generally an automatic process in which a *LifeWeb* document silently accumulates users' inputs and makes changes to itself according to users' preferences or needs. Adaptation can occur recursively through successive levels from instance to schema, or higher meta-modelling levels, throughout the whole population of document objects. This enables a smooth and incremental evolution of the whole system in a bottom-up fashion through all meta-modelling levels. This evolutionary process and the integration of software and tools to support the evolving system can be automated. System interoperability is maintained through levels of meta-modelling. Referential integrity is enhanced through the encapsulation of hyperlinks into objects. It may be ensured using a combined technique of *local link management* and *dynamic link binding*. *LifeWeb* documents hold information describing themselves, which can be deployed by search engines for resource discovery.

Chapter 2 The Web and Its Major Issues

...if future World-Wide-Web development simply consists of ill-considered "bolt-on" extensions to the existing architecture, then problems are likely to result.

L. Relihan [REL94]

2.1 The Web

The Web is a client-server system as depicted in Figure 2-1 [BER94]. In this architecture requested data is served by the server to the client. The data received is then processed and displayed by the client for the human user to read. The processing of data may involve interacting with other *legacy* (existing) information systems such as database, or other organisational systems (using protocols defined for these systems). Web clients can communicate with servers of all existing functionally compatible Internet protocols, including File Transfer Protocol (FTP) [FTP85], Network News Transfer Protocol (NNTP) [NNT86], Wide Area Information Servers (WAIS) [WAI94], and Gopher [GOP93], providing access to a universal information space on the Internet via a single simple interface.

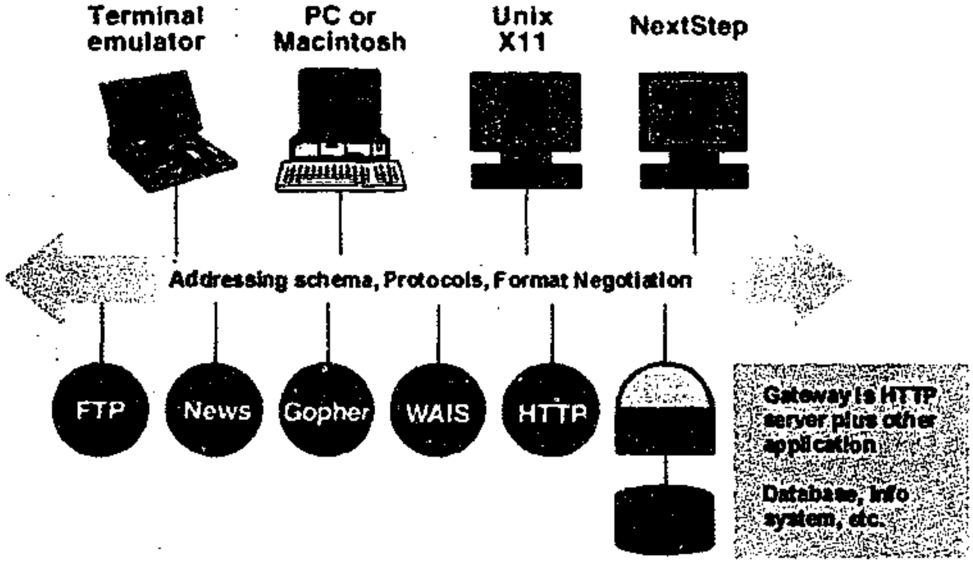

Terminal emulator
PC or Macintosh
Unix X11
NextStep
Addressing schema, Protocols, Format Negotiation
FTP
News
Gopher
WAIS
HTTP
Gateway is HTTP server plus other application
Database, info system, etc.

Figure 2-1 : The Web client-server architecture

(From [BER94])

The WWW defines a number of standards and protocols [BER94]:

- Uniform Resource Identifier (abbreviated *URI*): A scheme defined by the Internet Engineering Task Force (IETF) Working Group on URI [URI98], which assigns to every Web resource a world-wide unique identifier, making it possible for the resource to be universally referenced. URI is designed so that it is generic, appearing to refer to "the same" object although language, data format, or actual location may vary. The selection of language and data format is carried out by a negotiation process between the server and the client as explained below. In practice only a derivative of URI, i.e. the Uniform Resource Locator (abbreviated *URL*) [URL94] is being used. The URL is dependent on the physical location of the resource on the file server, and is therefore largely responsible for the "broken-link" problem described below (see Section 2.2.1.2). Currently the IETF Working Group on URI has concluded its work, and new Working Groups have been formed. Among these the most important one is the Uniform Resource Name (URN) Working Group [URN00]. URNs are designed to be persistent identifiers for information resources globally, which also support other legacy (existing) systems.

- Hypertext Transfer Protocol (abbreviated *HTTP*): An Internet protocol used by Web servers for receiving requests and transferring hypermedia and other kinds of data across the network. HTTP is a *stateless* protocol, that is, the network connection on which it runs is held only for the duration of one operation, so that the result it returns is ignorant of any previous operations performed by the client [BER94]. Statelessness is required, according to Berners-Lee [BER94], for (i) the efficiency needed for retrieving a hyperlinked object regardless of its actual storage (local or remote), and (ii) the fulfilment of the purpose of the URI that it should always refer to "the same" object regardless of the client's actions. HTTP was originally designed so that it could handle any data format and language via the respective format and language negotiation scheme. The client could specify the preferred format(s), or language(s), and the server would select a suitable object. In this case the URI used to identify such an object is called *generic*. This feature, however, is generally not used by clients because "generic URIs have been historically exceptional and this scheme is expensive to implement" [BER94]. HTTP defines a number of methods for use by the Web client. The most commonly used ones are *get* and *post*, which send parameterised queries to the Web server. It is therefore possible to dynamically generate Web pages based on the input provided by the client. Currently HTTP 1.1 [HTT97] is in use and HTTP-Next Generation (abbreviated of *HTTP-NG*) [HNG00] is under development.

- Hypertext Markup Language (abbreviated *HTML*): A language used by Web clients for the storage, display and transmission of hypermedia data over the network. All Web clients are required to understand HTML. HTML is a subset of the Standard Generalized Markup Language (abbreviated *SGML*) [SGM86, SGM00, HER94], a language widely used by the publishing industry. HTML is an extremely simplified version of SGML, so that it can be easily used by the general public as well as parsed by programs. Markup languages use tags to mark the start and end of the various "components" in a document, which are called *elements*. These tags, which are called *element types*, are defined in a Document Type Definition (abbreviated *DTD*). A DTD contains the formal description of the *schema* of a particular *document type*. HTML defines a fixed tag set of a single document type, the first version of which is a collection of primitive document element types such as paragraph break, several levels of headings, lists, menus, and anchors to define hyperlinks [HTM92]. Many more element types have been added to this first HTML specification as the users' and developers' requirements have become increasingly sophisticated. At the time of this writing, the version currently in use is HTML 4.0 [HTM98], and Extensible HTML (abbreviated *XHTML*) [XHT00], has just become a recommended standard.

2.2 Problems with the Web

Since the initial proposal of the Web only very few years ago, this system has seen a steady exponential growth rate throughout the whole world in practically every aspect. In terms of quantity, the statistics collected by Merit Network, Inc. [MER00] for the NSFNET (National Science Foundation Network) backbone [NSF95] shows that the Web has made a giant leap from zero to the highest volume application within only one year (1994-1995), with the traffic in bytes appearing to increase quadratically [GVU95, NGU96]. In more recent datasets, for example, the series of Internet

Domain Surveys conducted at Internet Software Consortium [ISC00], the number of hosts is shown to have grown from 313,000 in October 1990 to nearly 8 million in January 2000 (Figure 2-2).

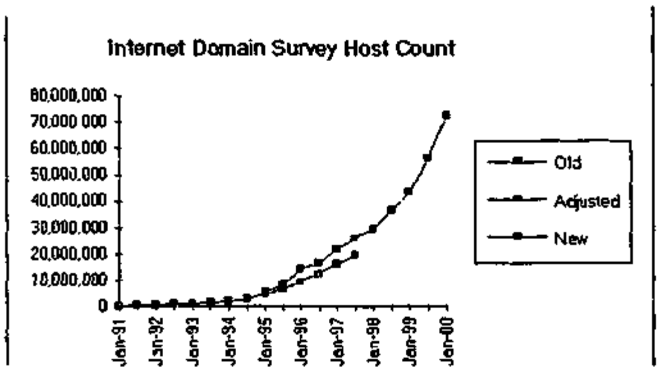


Figure 2-2: Internet Domain Survey Host Count

Source: Internet Software Consortium (www.isc.org)

In terms of quality, the Web has undergone fundamental changes in its functionalities, from (primarily) a simple static document system to one that provides highly dynamic content and support for sophisticated enterprise and distributed computing requirements. Since the Web was not originally designed for these complicated technologies, efforts in Web application development increasingly run into limitations of the Web infrastructure [ING97, MAN98, MAN99]. With its explosion and increasingly complex enterprise demands, serious problems with the system are showing.

2.2.1 At the deployment level

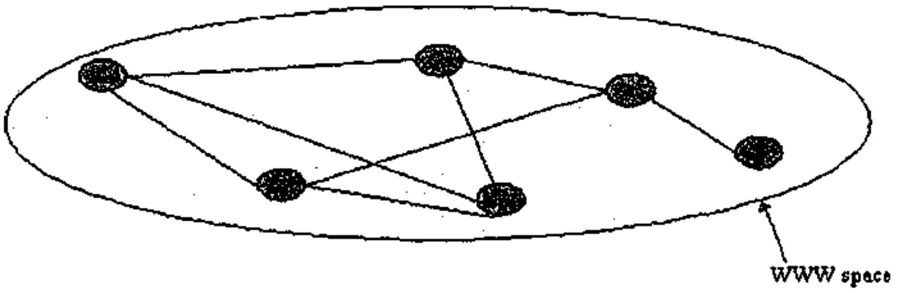


Figure 2-3 : The hypertext data model

The Web was initially designed as a simple network of documents interconnected with each other by hyperlinks (see Figure 2-3). Typically hyperlinks are statically embedded within documents, and activating a hyperlink essentially takes the user from one document to another, similar to the effect of the "goto" statement in some computer languages [HYP00]. This simplicity has gained the Web overwhelming success over the last decade, but as the number of documents and hyperlinks grows, has resulted in a spaghetti-like network with many shortcomings. Some of the most well known problems are:

2.2.1.1 Lost in hyperspace

The complex branching structure of the Web makes it conceptually unmanageable. It creates a heavy cognitive load on users, making navigating Web sites difficult. Even in a well-organised Web site, users are disorientated, lack a sense of size, limits, and current position in the whole. They find it increasingly hard to have an overview of the site, to locate or relocate the desired information – all of it and only it, to recognise new, updated or outdated information, or to discover resources [CUN93, PER98, BOU99, HYP00].

2.2.1.2 Violation of link integrity

The exponential growth of the Web coupled with the complexity of the hypertext network over a distributed system makes hyperlink management impossible. As new resources are added, old ones are required to be removed or migrated to different locations. Thus references to Web resources become no longer valid, constituting the notorious "broken-link" or "dangling-link" problem. This problem has been found to waste users' time and discourage them from visiting the problematic Web site or from using the Web altogether [REL94, ING95, ING96, DAV98a, HYP00]. Another true although less obvious problem is that of "orphan" resources. These are resources that are not referenced by any other resources, and thus become unreachable [HYP00].

2.2.1.3 Difficulties in data management and maintenance

The hypertext data model of the Web does not provide a way to express the structure of the document. Current practice is to directly (hard) code the document components into the source file and use the server file system to express the structure. For instance, one might put different chapters of a book in corresponding directories. Changes to a structured document often need to be synchronised among its components. For example, the removal of a chapter would require all the following chapters to be renumbered, all references to it to be removed, and a new table of contents to be generated. These maintenance operations are typically carried out manually using commands and tools *native* to the server machine, and therefore, orthogonal to the Web. If the file system has been used to represent the document structure, manipulations to Web resources are required in both the file system and the Web system, but these are essentially two disjoint domains. This process is expensive and prone to mistakes and inconsistencies. Its problems are well documented [ING95, GEL97, ING97, NGU98a].

2.2.1.4 Difficulties in document design and authoring

The lack of a way *native* to the Web to express document structure results in a presentation-oriented mindset in Web design. This has been found to distract an author from the overview of the whole document structure in order to worry about presentation details at the design stage [HYP00]. It also induces the author as well as the authoring tool manufacturer into using presentation tricks to represent structural components. For example, it is common to use a combination of bold (<B></B>) and font tags (<FONT></FONT>) to make a data content appear as a (styled) heading. This inappropriate use of presentation tags will, in the long term, make the maintenance task of non-structural elements impracticable.

2.2.1.5 Customisability, adaptability and evolvability

The Web, being stateless (see Section 2.1), cannot by itself make or record changes to Web documents across sessions. Thus it is not possible to have documents tailored to individual needs in a *native* way. Customisation is the problem of "one-size-fits-all", and is becoming increasingly important with the expansion of the Web, which has to serve millions of different users worldwide. The same pressure is experienced at the organisational level, where the same set of data must be served differently or presented in different views to different types of personnel or clients.

Adaptation is a more "advanced" form of customisation, where a Web document "learns" from previous experience to make changes to itself accordingly. Previous experience is typically provided from a user's behaviour or input, so that adaptation is made towards betterment for the user. Adaptive documents are especially useful in the Web, which holds too large a wealth of information for users to find their way around or to filter out useful pieces. They have been used in the Web primarily to help with the lost-in-hyperspace problem (Section 2.2.1.1). Examples are the adaptive search engine [IKE99] and adaptive hypermedia [EKL98, PER98, BRA99a]. Their development and design, however, are difficult and complicated since, like customisation, are not *natively* supported by the Web due to its statelessness [BRA99b].

Evolution is a long-term result of customisation and adaptation. Automatic evolution, as with customisation and adaptation, is not supported because of the Web's statelessness. Besides, no evolutionary path has been defined for changes in the Web to follow. (In this respect, evolvability is itself a design issue.) Most of them have been happening in a non-systematic and unpredictable manner, arbitrarily driven by users' or organisations' needs, so that the current system cannot deal with future modifications. This topic will be discussed in detail in the subsequent chapters of this thesis.

2.2.2 At the design level

Most of the problems mentioned above can be accounted for in the design of the Web document system.

2.2.2.1 Structure

As was pointed out in Section 1.1, the Web was proposed as a solution to the knowledge management problem at CERN due to information being lost because of the high turnover of people. It was necessary to create a shared information space where people could quickly and easily record unrestrained relationships between arbitrary objects without any constraints on data structure [BER89]. That is the picture of the Web today as a cyclic directed graph with documents (essentially files) acting as nodes and embedded links as arcs. It was emphasised in the initial proposal by Tim Berners-Lee [BER89] that this form of "web" organisation offered the freedom necessary to describe a complex and evolving system, whereas others such as hierarchical or relational structures were inflexible and many systems would simply not fit. Such structures were therefore just not considered.

This concept has been reflected in the design of the Hypertext Markup Language where there are no tags to express the (usually hierarchical) structure of the document. The (normally called) "structural" tags such as the heading tags (<H1></H1>), or the division tags (<DIV></DIV>), do however carry some structural semantics for the single content delimited between the start and end tags, but do not indicate any (structural) relationships among these components. They therefore cannot cohesively express the structure of the document as a whole, and are usually treated in a presentation-oriented manner. Other HTML critiques also share the same point of view [BOS97, MAN98, MAN99].

The fact is that, while it is true that the *inter*-relationships between nodes would be most useful if unrestrained for the purpose set out in the initial proposal of the Web [BER94], the *intra*-relationships between components *inside* a node are in most cases, structured. Traditional paper-based documents are almost inevitably hierarchically structured. The same organisational requirement is also found with electronic documents, even in a highly dynamic environment such as the Web. Large amounts of unstructured data are simply impossible for the human mind to grasp, manage or manipulate, especially in the increasingly growing Web. Unrestrained links can be an extension of structural links, but cannot substitute for them. What is lacking in the Web in this respect is a way to express the internal structure of a node. This is found to be directly related to the problems: lost-in-hyperspace, data management and maintenance, and document design, noted in Sections 2.2.1.1, 2.2.1.3 and 2.2.1.4 above.

2.2.2.2 Inextensibility

This problem with the Web is demonstrated in the design of HTML, the language used to represent the Web document system. The tag set defined in HTML is primarily presentation-oriented and fixed to only one single document type. As technology advances, new tags have to be added to incorporate new technologies. As the Web has spread throughout the world to different industries and communities, there arises the need to associate semantics to the tags, and to define different tag sets that are more specific to each application domain. HTML, however, cannot be extended to satisfy the demand. This has two major consequences:

- With regard to the evolution of the Web, the inextensibility of HTML requires that any changes to HTML should be centrally controlled. This generates a global chained effect on related technologies such as browsers and authoring tools, and has invariably inhibited a smooth evolution of the Web at the schema level.
- With regard to the lost-in-hyperspace issue (Section 2.2.1.1), the lack of semantics in HTML makes it impossible to describe the resource captured in a HTML document (without the use of another mechanism), which can be exploited by search engines or web bots to aid users in resource discovery.

The advent of the Extensible Markup Language (XML) [XML98] is especially designed to remedy this problem, as will be explained in Section 3.2.2.

2.2.2.3 Functional behaviour

In contrast to traditional paper-based documents, which are totally static, Web-based documents are highly dynamic and appear to have a functional behaviour. In fact some authors have considered the Web to be a simple object-based system with its objects identified by URLs and "behaviour" defined in the HTTP server [ING95, MAN99]. This point of view however, is not well founded. The "behaviour" mentioned above is not defined internally within the Web document object. It is "borrowed" either from the methods defined in the HTTP server (see Section 2.1 and [HTT97]), or from external technologies such as CGI-SCRIPT [MCC95, RAG97], or embedded OBJECT [RAG96]. As such, Web documents do not support encapsulation, and remain non-functional entities in much the same way as paper-based documents, with an added capability to "borrow" functionalities from external entities. This lack of a functional behaviour of its own disqualifies Web documents from being "objects" in the object-oriented sense. This means that a Web document (as it currently is) is not an encapsulation of a self-contained unit complete with well-defined state and behaviour. This has two major consequences:

- With regard to system evolvability, Web documents cannot function and evolve themselves as independent, complete units.
- With regard to integration with enterprise systems and the high demand for increasingly sophisticated Web applications, Web documents lack an Application Programming Interface (API) for accessing and manipulating their document data and data structure. (The recent release of the Document Object Model (DOM) [DOM98] is an answer to this issue as will be seen in Section 3.2.3).

2.2.2.4 Document properties

It is well established in the publishing industry that a document normally possesses three distinct properties, namely structure, presentation and content. This concept is reflected in the design of major document processing applications such as LaTeX [LAM86] or Microsoft Word. However the distinction between these document properties has not been designed into the Web. A Web document as expressed by the Hypertext Markup Language (HTML) is one in which structure, presentation and content are intermingled with each other. In the example above, the heading (<H1></H1>) and division (<DIV></DIV>) element types not only indicate their structural semantics but also dictate to the browser how to display them. Contents are embedded between tags, which in turn can contain other tags, so that any operation on the document becomes heavy and complicated, as it has to carry the whole bulk of its tags and contents interspersed with each other. All of this has made it very difficult to manage, maintain, or customise the document, as any change to it will affect the whole of itself. Since these aspects of structure, presentation and content are not distinguished from each other, it is not possible to manipulate the document based on any single one of these aspects. This design problem contributes to the difficulties in data management and maintenance, and Web design noted in Sections 2.2.1.3 and 2.2.1.4 above.

2.2.2.5 Embedded hyperlinks

When using HTML, hyperlinks are encapsulated within the document data. As has been pointed out by researches in Open Hypermedia Systems (OHS) [DAV98a], this approach has the advantages of (i) being simple when editing document data, since the source anchor of a hyperlink can be freely moved and edited at the same time to synchronise with the changes made in the document data; and (ii) being efficient and scalable when resolving a hyperlink (for making a hyperlink jump), since the destination anchor is immediately available from within the document data, and there is no need to store links in a database. The cost of these advantages is that links are only uni-directional, because only the node that has the embedded link knows about the link. This has the severely negative consequence of making automatic maintenance of hyperlink integrity in the Web both impractical and impossible. Each movement of a document node requires searching and updating every link to the moving node in all Web documents worldwide, where we may not have write permission. The broken-link problem (see Section 2.2.1.2) stems from this design flaw.

2.2.2.6 Statelessness

As mentioned in Section 2.1 above, the Web is built on HTTP, a stateless protocol, as a requirement for the design of the URI (which should always refer to "the same" object regardless of previous client interactions), and for the efficient retrieval of hyperlinked objects. These have come at the cost of not being able to provide objects customised to the needs of individual human Web user. Consequently adaptation and evolution of Web documents are also not supported (see Section 2.2.1.5).

2.2.2.7 Meta-data

Briefly, document meta-data is information that describes the document's data content. It has been used in the Web for quite some time to aid in the searching and discovering of Web resources. This intention, however, has had two major hurdles over decades. First, there was no standard framework established on the Web for describing document data. Second, there was no unified taxonomy or mechanism to define terms in which documents could be described and classified. Consequently, even though meta-data has long been used on the Web, it is mainly for human readers, and may be only partially supported and understood by some user agents (browsers) and search engines. In addition, authors usually "do not bother" to insert meta-data, mainly because it has not been standardised, and they do not know what standard to follow. This lack of support for meta-data makes it difficult for the Web to provide help with the lost-in-hyperspace problem mentioned in Section 2.2.1.1.

2.3 Conclusion

From the problem analysis above, it can be seen that the Web has outgrown its initial design. Compelling facts about the link between the problems at the deployment and design levels, and the problems at the design level itself, suggest that a solution should be at the design level. The complexity of the Web, which is one of the major causes of the above-mentioned problems, strongly proposes a solution by abstracting the complex entity into a model (or models) [SMA96]. The nature of the problems, which is closely related to the advantages offered by object-oriented technology (maintainability, manageability, extensibility), makes object-orientation very attractive for a solution.

The dynamics and liveliness of the Web have inspired us to investigate some other areas, such as natural life, for new design ideas.

Chapter 3 Data modelling on the Web

The Web has become a very complex entity. And to deal with complexity we need to reduce complexity by a process known as abstraction.

P. Sma'l [SMA96]

Modelling is the process of abstracting the world of our interest, or the Universe of Discourse (UoD), into some model or specification. One particular set of related facts from which the model is induced forms an instance of that model. Going from models to instances is like going from factories to products. Once constructed it can be used to produce an infinite number of instances sharing common characteristics that have been abstracted in the model. To be usable a conceptual model must be represented by a precise and unambiguous notation. In software development models are used mainly for the reduction of complexity, understanding, communication, and testing a physical entity before building it [RUM91, GIU96]. With the level of complexity of the Web information system, data modelling promises a satisfactory solution.

Data modelling, especially object-based and object-oriented (which will be generally referred to as *object*), is among the approaches towards a better Web. It has been recognised as a solution to problems of the Web, in particular those of manageability, maintainability, and extensibility, in the first four years after the coming of the Web [KAP94, ING95, REE95]. Object modelling is becoming a major trend in the development of the Web because of the proven advantages of object technology, the resemblance of the Web to an object system, and the need to integrate the Web with sophisticated enterprise systems, which perhaps can be best done with the extensibility feature of object technology [MAN98, MAN99].

This chapter reviews the Web data models currently available, pointing out what has been achieved and what is yet to be addressed. There are three major groups among the systems providing a data model for the Web:

- The first group is motivated mainly by the need to incorporate complex services and existing enterprise systems into the Web. In this group, the Web is used as a front-end to provide services to users at the inter-net or intra-net levels. Issues such as data structure, flexibility, and manageability are also addressed here, but are not solved for the Web document system *per se*. This group includes such systems as W3Object [ING95], WebObject [WOB00], WebComposition [GEL97], ILU Requester [ILU91, ILU96], CorbaWeb [MER96], and ANSAWeb [REE95]. We shall refer to this group as *service-centred.*
- The second group aims to model the document system intrinsic in the Web and tries to improve its shortcomings such as those regarding data structure, extensibility, manageability, and maintainability. Systems such as Hyper-G [AND95], Extensible Markup Language (XML) [XML98], and Document Object Model (DOM) [DOM98], belong to this category. We shall refer to this group as *document-centred.*
- The third group is concerned with constructing models to describe the data content in Web documents. This group aims at facilitating resource discovery, content rating, cataloguing, and knowledge sharing and exchange. It includes meta-data projects such as the Platform for Internet Content Selections (PICS) [PIC97], the Warwick Framework [LAG96], the Dublin Core [DUB00], the Resource Description Framework (RDF) [RDF99], and the like. We shall refer to this group as *meta-data.*

3.1 Service-centred modelling

Systems in this group focus on providing Web services (W3Objects [ING95, 96, 97], WebComposition [GEL97]), and integrating existing enterprise systems with the Web (WebObjects [WOB00], Web* [ALM95], ANSAWeb [REE95], CorbaWeb [MER96], ILU Requester [ILU91, ILU96]). Their common aim is to remedy the extensibility problem of the Web by means of object-oriented technology. W3Objects [ING95, 96, 97] also looks particularly at the link management issue. ANSAWeb specifically attempts to achieve interoperability between the Web and systems implementing the Common Object Request Broker Architecture (CORBA) [BEN95], a standard for distributed object systems developed by the Object Management Group (OMG) [OMG00]. ILU Requester is directed towards increasing service execution performance caused by the process forking in the Common Gateway Interface (CGI) [MCC95] technology[2].

Except for WebComposition, which is a non-distributed object-oriented system, most of these systems use distributed object technology. In particular Web* [ALM95], CorbaWeb, and WebObject are all CORBA applications. ILU Requester employs a CORBA compatible technology, the Inter-Language Unification (ILU) [ILU91], with its own Interface Specification Language (ISL), to capture HTTP daemons as distributed objects. W3Objects implements distributed object-orientation using C++ and the Remote Procedure Call (RPC) protocol [RPC95], although standards defined in CORBA and ILU can also be used. Commercial applications such as WebObjects can integrate with a wide range of industry standards such as CORBA, COM/DCOM (Component Object Model/Distributed Component Object Model) [BEN95, BEN97, MOW97], and major Database Management Systems (DBMS) such as Oracle, Sybase and Informix.

The data models in these systems, if provided, are basically designed to encapsulate services into objects (CorbaWeb, WebComposition, W3Objects, Web*, WebObjects), and decompose a Web application into screen elements or application-specific components (WebComposition, WebObjects, W3Objects). CorbaWeb also captures databases into objects. ILU Requester is different from the other systems in that it models the HTTP server, not the Web documents or services provided at the client. ANSAWeb does not have a data model, but only provides interoperability between a CORBA system and the Web.

Access from a Web client to objects defined in these systems is usually provided through CGI scripts [MCC95], written in a scripting language defined for each system. CorbaWeb creates CorbaScript to allow for dynamic invocation of operation on CORBA objects. Similarly Web* has TclDII, an extension of Tcl [OUS94], that uses CORBA's Dynamic Invocation Interface (DII) to provide access to Orbix services [ORB00]. (Orbix is a commercial CORBA application.) WebObjects designs its own environment, which includes a CGI Script written in its own scripting language, a HTML Template that may contain WEBOBJECT markup elements, and a Declaration file to specify the binding between WEBOBJECT and HTML markup elements. There are some good design ideas in these systems, for example WebObjects introduces a separation between business logic (CGI Script), presentation (HTML Template), and data access (Declaration file). However the scripting languages defined for these systems may be difficult to use. For example, TclDII requires CORBA knowledge

[2] Every invocation of a CGI script causes a new process to be created, resulting in a performance penalty.

since the user has to precisely define the IDL (Interface Definition Language) [IDL97] type for each parameter. More importantly, each language is specific to a proprietary system so that there is no standard way to access objects in the underlying object system, and objects in different systems cannot communicate with each other.

Some other approaches have been used in other systems. W3Objects provides Web access to its objects through a gateway implemented as a plug-in module to an extensible server such as Apache [APC00]. URLs beginning with /w3o/ are passed to the gateway module, which binds the requested named object to a W3Object within a nameserver, and invokes the appropriate methods. In WebComposition a Web client accesses WebComposition components only indirectly through the normal file system. These files are generated by the WebComposition's Resource Generator and Component Server using "incremental publishing". This technique incrementally maps WebComposition components stored in the Component Server to HTML elements for the requested file, and generates the corresponding HTML file. This is done not on the user's request but at a time decided by the system itself when some change to the file is detected. These approaches also provide valuable input, but as with the scripting approach, cannot be standardised.

Systems in this group offer advantages obtained from object-oriented technology. In particular, they all attempt to repair the manageability, maintainability and extensibility problems of the Web. Their common approach is to "extend" the Web with services, either by developing new applications or by integration with existing enterprise systems. As this is done using object-orientation, the extension itself possesses the well-known advantages of object-orientation. Consequently the Web as a whole appears to be manageable, maintainable and extensible. As a matter of fact, however, these systems never really change the Web system as such, but only apply a "patch" to it. The Web infrastructure still suffers from problems of manageability, maintainability, and extensibility.

As pointed out in Section 2.2.2, these problems stem from a design issue of HTML, which is fundamentally a notation derived from SGML [SGM86], a standard from the publishing industry, to represent a document. A solution for them, therefore, should be an improved design for the Web as a document system. The service-centred systems, however, focus only on providing services using the Web as a front-end. Consequently their models do not capture the Web as a document system, and do very little in solving the design problems mentioned in Section 2.2.2. In addition, when the Web infrastructure is changed, they are greatly affected because there is no proper mapping between their models and the model implied in the Web document system.

Service-centred systems are also limited in terms of extending themselves because of the complication involved in developing Web applications. The Web gained enormous success because of the simplicity inherited from well-understood concepts of a document. Although this simplicity causes us many problems (see Section 2.2), it is necessary to preserve it (as much as practically possible). The data models of service-centred systems, on the other hand, are built on complex software development concepts.

In brief, extending the Web in the service-centred way is both limited and fragmented because it is complicated and cannot be standardised. Service-centred systems do fulfil their objectives of integrating complex services and enterprise computing with the Web, but to solve the design problems of the Web, we need another approach: document-centred.

3.2 Document-centred modelling

The systems in this group provide data models that capture the "document characteristics" of the Web. There are three major systems in this category: HyperWave [HYP00], the Extensible Markup Language (XML) [XML98], and the Document Object Model (DOM) [DOM98].

3.2.1 HyperWave

HyperG [AND95], which was later commercialised into HyperWave [HYP00], was the first system to attempt to improve the document structure inherent in the HTML design. It does this by introducing the concept of an *abstract container*, which can be one of four types: Collection, Sequence, MultiCluster, and AlternativeCluster.

Collections are overlapping sets of items, and an item can be a document (content or material) or a Collection itself. Thus an item can belong to one or more Collections, and Collections can be nested. A Sequence is a Collection whose items are sequentially ordered. Collection and Sequence together map well into the traditional hierarchical document structure of unordered, or ordered, nested structural components, respectively. In addition, due to the "soft" characteristics of an electronic document, an item can also logically belong to (or strictly speaking, be associated with) many Collections. This allows for reusability and the ability to define multiple *views*, where a Collection corresponds to such a view.

MultiCluster and AlternativeCluster act like terminal Collections in the document hierarchy with all their items unfolded at once for display. (As a comparison, a Collection or Sequence is displayed as a menu of closed folders or document items.) A set of (ordered or unordered) paragraphs and images is an example of a MultiCluster. This allows a finer granularity of the document structure in the sense that a document (or structural component) can be broken down into non-structural components, or the "raw materials" that build it. In this respect, a cluster can be considered to have captured another type of document "structure": unstructured documents. In HyperWave clusters are useful in collaborative authoring, where each author can independently supply (relatively small) pieces of materials into a cluster, and let the container assemble and display them as one document. An AlternativeCluster is like a MultiCluster, but returns only one of its elements instead of a combination of all elements. An AlternativeCluster is useful for selecting between available options, for example, languages.

With these abstract containers, a document can be organised into a coherent structure meaningful to its author. Items in a container can be managed and maintained by the system. For example, an item can be inserted into or removed from a Sequence, and the system automatically adjusts the relevant attributes such as numbering, structural links, and table of contents, to reflect the change. This can be done because of the distinction between structure and content, two of the three document properties mentioned in Section 2.2.2.4. Both static and dynamic customisation in structure and content can also be supported by the use of views. Such a view can be a statically designed Collection, or a Collection of structural components selected by their access permissions (so that different users or user groups with different access permissions will see different views), or a Collection that is dynamically generated from a stored query (which is a search query). A stored query can be a member of a Collection, and is itself a (dynamic) Collection, whose members are not statically linked to it, but generated on the fly by evaluating the query. A view generated from a stored query generally does not have any structure but contains only the materials (leaves in the collection hierarchy) that satisfy the search criteria stored in the query. An identified user can also have a personal Collection called *Home Collection*, which acts like a sophisticated bookmark list, where he/she can choose which Collections (including stored queries) to add or remove.

The use of the structural elements also allows HyperWave to control the document presentation to a certain extent by defining different styles for structural layout and navigation. A presentation style can be configured for a user (by the system administrator or the user him/herself), which has a global effect in all Collections. (For example, if "Partial Tree" is selected, all collection listings will be displayed in the browser as partial trees.) In this way, customised presentation can be catered for, although a user or author is limited by the styles pre-defined by the system (or the HyperWave application developer who has modified the system). Customisation can also be provided in HyperWave by the use of *cookies* [COO00]. This technology however, is criticised for privacy invasion [LIN98]. In addition, since *cookies* does not define a model upon which customisation services are performed, the types of services provided are arbitrary and limited, and customisation information cannot be deployed to drive a systematic evolution of the Web.

All HyperWave objects (for example, document contents, collections, queries, hyperlinks, and users) are stored at the server in an object-oriented database maintained by the server. HyperWave can thus keep referential integrity among its components using this database,[3] and has built-in search facilities based on meta-data. These meta-data are simply attribute names and values of HyperWave objects stored in the database. Customisation on hyperlinks is also supported by assigning hyperlinks with different access permissions (similar to what is done with structural elements), so that different users or user groups will follow different navigational paths. HyperWave provides a HyperWave API for application programmers to extend the server's functionalities (using HyperWave's own language).

The major advantage of HyperWave is in solving the problems of document structure and document properties (Sections 2.2.2.1 and 2.2.2.4), and consequently other derived ones, namely document management and maintenance (Section 2.2.1.3), document authoring and design (Section 2.2.1.4), and customisability (Section 2.2.1.5).[4] It also offers advanced features in maintaining link integrity by the use of external links (relating to the referential integrity and embedded link problems described in Sections 2.2.1.2 and 2.2.2.5) and facilitating search by the use of meta-data (relating to the lost-in-hyperspace and meta-data problems described in Sections 2.2.1.1 and 2.2.2.7), albeit with the help of a database. Recently, HyperWave provides the *Document Class*, an extension of the container classes, whose instances can be made persistent and stored in the server (rather than in volatile memory as in conventional object-oriented systems).[5] Programmers can override attributes and methods defined in this class, as well as derive subclasses from it (in the object-oriented way). The Document Class is designed to provide a way to develop special document types, for instance, one with specific behaviour. HyperWave however, does not deal with problems concerning adaptability and evolvability (Section 2.2.1.5). We also note that HyperWave is not an object-oriented system, although it uses an object-oriented database.

3.2.2 Extensible Markup Language (XML)

Extensible Markup Language, abbreviated XML [XML98], is an effort by the World-Wide-Web Consortium (W3C) [W3C] to repair the design problems of HTML. Like HTML, it is also a subset of the Standard Generalized Markup Language (SGML) [SGM86], allowing the use of tags (including nested tags) to describe elements in a document. Unlike HTML, however, which defines only a fixed

[3] Referential integrity across HyperWave servers that belong to a particular *server pool* may also be maintained [HYP00].

[4] HyperWave also deals with other issues such as collaborative authoring and security. These are not included in the scope of our project, and thus are not reviewed in this thesis.

[5] This simply means instances of a Document Class are stored in the server in the same way as those of other types of abstract container such as Collection.

tag set for a single document type, XML is really a meta-language that defines the grammar. It permits the definition of tags and other markup syntax (including the definition of HTML in terms of XML). An XML document for example, may look as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE book System="http://lifeweb.org/book.dtd">
<book>
      <chapter id="chapter1" heading="CHAPTER 1">
            <section id="section1.1" heading="Section 1.1">
                  This is the text of section 1
            </section>
      </chapter>
      ...
</book>
```

Figure 3-1: Extract of an XML document

This example shows the <chapter> and <section> tags that are not present in HTML. Such tags are defined in a Document Type Definition (DTD).[6] The DTD for the above example is named *book*, which is declared in the <!DOCTYPE> clause, and its name is used as the "covering tag" that encloses all other elements in the document. XML calls <book> the document's *root element*. A DTD is a single file, or several files, which formally defines a particular type of document. It describes the schema of the corresponding document type, specifying factors such as what element types there are, their attributes and attribute types, their possible nesting, and so on. The DTD for the XML document in the above example defines the two elements <chapter> and <section> as follows:

```
<!ELEMENT chapter (section)>
<!ATTLIST chapter
      id          ID          #REQUIRED
      heading     CDATA       #IMPLIED>
<!ELEMENT section (section | CDATA)
<!ATTLIST section
      id          ID          #REQUIRED
      heading     CDATA       #IMPLIED>
```

Figure 3-2: Extract of a Document Type Definition (DTD)

When a DTD is declared in an XML document (with the <!DOCTYPE> clause) as the file "http://www.lifeweb.org/schema/book.dtd" in the above example (Figure 3-1), it is used to validate the XML document when the document is parsed. Such a DTD is called an *external* DTD. Element type definitions may be declared locally in the XML document that uses them, as illustrated in the example below (Figure 3-3). In this example, the DTD "greeting" defines one single element,

[6] XML also permits a document without a DTD, called a *well-formed* document. In this thesis we discuss only *valid* XML documents, that is, those *with* their own DTDs.

"greeting", that contains character data (PCDATA). The element "greeting" is used immediately in the same XML document to mark up the text "Hello, world!". A DTD declared this way is called an *internal* DTD.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE greeting [
  <!ELEMENT greeting (#PCDATA)>
]>
<greeting>Hello, world!</greeting>
```

Figure 3-3: Example of an internal DTD

In addition, an XML document may use element types that are defined in several DTDs, via the *namespace* mechanism [NAM99]. Namespace allows an "inline" association of an element name or attribute name (that is being used in an XML document) with the DTD that defines it. Such a name is called a *qualified name*. Thus, an XML document may use element types or attributes that are not defined for its own document type (that is, in the DTD declared in the <!DOCTYPE> clause). It may also use element types or attributes defined in multiple DTDs. In this thesis, we differentiate the DTD declared in the <!DOCTYPE> clause from other DTDs used in a document, and call it the *main DTD*. Figure 3-4 shows an example of the use of XML namespace. In this example, the namespace declaration in the first line binds the "chem" prefix to "http://chemistry.org/schema.dtd" for the "e" element and contents. In the second line, the "molecule" element is qualified by "chem", thus assumes the namespace "http://chemistry.org/schema.dtd".

```
<e xmlns:chem='http://chemistry.org/schema.dtd'>

      <chem:molecule>C6H6</chem:molecule>

</e>
```

Figure 3-4: Example of the use of XML namespace

As can be seen from the above examples, since XML allows nested structures, it can easily be used to represent a document hierarchical structure. Its extensibility comes from the expressive power derived from its meta-level. Various document types can be defined and domain-specific "jargon" specified, meaningful to the respective communities. In addition, XML also forces the separation between structure (that is, structural contents) and presentation, imposing the use of stylesheets to specify how the user-defined element types are presented on the browser. These advantages of XML convert the Web infrastructure into one that can be managed, maintained, and extended.

XML (currently in version 1.0) is th . base member in the XML family, and the first to reach the W3C Recommendation status. Some other important members include:

- XLink: XML Linking Language [XLK00] provides advanced hyperlinking functionality such as links to multiple destinations, bi-directional links, inline and out-of-line links, associating meta-data with links, and databases of links. XLink is represented in the XML syntax. It has been mainly influenced by established standards in HTML, HyTime [HYT97], and Text Encoding Initiative Guidelines [TEI94]. Many other linking systems such as Microcosm [DAV93], FRESS [DER99], Dexter, [DEX94] have also provided input to the design of XLink.

- XPointer: XML Pointer Language [XPT00] is a language used to identify a fragment for any URI-reference that locates an XML resource. It supports addressing into the internal structure of an XML document. This includes traversing the document tree and selecting internal parts based on various properties such as element types, attribute values, character content, and relative position. XPointer and XLink were split from the former XLL project to separate the linking functionality from the addressing of the objects being linked. XPointer is based on XML Path Language (XPath) [XPA00], a language used for addressing parts of an XML document, which is a newly released W3C Recommendation in the XML family. XPointer has been influenced by other standards including HTML, HyTime [HYT97], and Text Encoding Initiative Guidelines [TEI94]. It also receives input from the Open Hypermedia Systems (OHS) such as Dexter [DEX94], FRESS [DER99], Microcosm [DAV93], and Intermedia [MEY86, YAN88].

- XSL: Extensible Style Language [XSL00] defines a language to write stylesheets for XML documents. XSL consists of two parts: XSL Transformations (XSLT) [XSLT00] for transforming XML documents, and an XML vocabulary for specifying formatting semantics. XSL specifies the styling of an XML document by using XSLT to transform the document into another document that uses the formatting vocabulary. At the time of writing, XSLT has reached W3C Recommendation status, but XSL is still a working draft.

- XML Schema: XML Schema [XSC00] defines a rigorous data model (including a language) to constraint and validate XML documents. Currently, the XML Schema project consists of two parts. *XML Schema Part 1: Structure* specifies the XML Schema Definition Language, which provides a superset of the capabilities found in XML 1.0 document type definitions (DTDs). This specification depends on *XML Schema Part 2: Datatypes*, which defines the data types to be used in XML Schemas. XML Schema is still a progressing project.

In addition, XML also has some associated standards. Two important ones are (i) the Document Object Model (DOM) [DOM98] that defines an Application Programming Interface (API) for an XML document; and (ii) the Resource Description Framework (RDF) [RDF99] that provides a foundation for using meta-data to describe resources. These will be described in the next two sections.

It can be seen that, together, members of the XML family and its associated standards have addressed or attempt to address issues concerning data management and maintenance (including document structure and document properties as mentioned in Sections 2.2.1.3, 2.2.2.1 and 2.2.2.4), extensibility (Section 2.2.2.2), referential integrity (including embedded links as mentioned in Sections 2.2.1.2 and 2.2.2.5), functional behaviour (Section 2.2.2.3), and meta-data (Section 2.2.2.7). XML has created a lot of excitement over the Web and many people have seen it as the future of the Web. Usdin et al. [USD98], however, have reminded us that XML is only an "enabling technology". "Well-designed XML can provide a valuable tool... but XML by itself is not a solution to any problem." Besides, the continual growth of the XML family does make a person watching the development of the Web wonder whether it will lose the simplicity that has won the Web such a phenomenal success. These issues are perhaps ones to be left with others that still remain unsolved or partially solved, such as statelessness (Section 2.2.2.6), customisability, adaptability and evolvability (Section 2.2.1.5).

3.2.3 Document Object Model (DOM)

The Document Object Model, abbreviated DOM [DOM98], is another W3C project that defines an object-oriented Application Programming Interface (API) for a Web document. DOM maps well into the XML data model, and thereby defines a complementary behavioural model for XML. It allows a Web application to access and manipulate an XML (or HTML) document's elements, their structural links, and their attributes.

DOM is a generalisation of Dynamic HTML [DHT97] defined by Microsoft and Netscape. DOM level 1 has been stabilised and is currently a W3C Recommendation. The major classes of DOM level 1 Core Specifications are shown in Figure 3-5 in UML [UML98] notation. In this system Node is the base type for all other types that can be found in a document structure tree. Nodes are nested, so that a Node can have several sequentially ordered child Nodes, and usually a parent Node; except for the root Node, which represents the document itself. A Document, Element, or Attribute object represents an XML (or HTML) document, element, and attribute respectively (as shown in Figure 3-5). A Text object captures any non-markup values, for example the data contents between the start and end tags of an XML element. A Comment object corresponds to an XML (or HTML) comment, which will be ignored by a parser. PI stands for *Processing Instruction*, a concept derived from SGML [SGM86]. A PI object identifies by name a helper application that can process an unparsed entity.

As a complementary functional model to XML, DOM specifically addresses the issue of a lack of functional behaviour mentioned in Section 2.2.2.3. Other problems, in particular the meta-data issue, are addressed in the next group of data models as explained below.

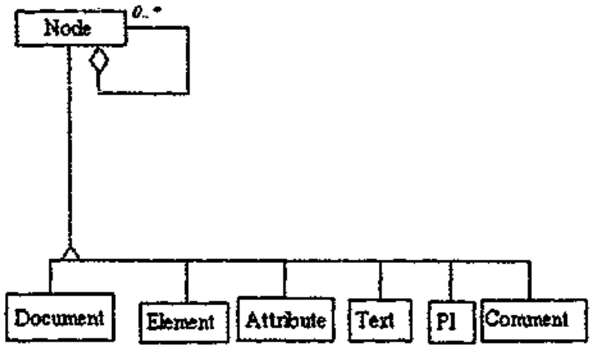


Figure 3-5: DOM level 1 Core

3.3 Meta-data modelling

Systems in this group focus on establishing some mechanism for describing Web resources. Their objective is to help in services such as resource discovery, knowledge exchange, content rating, and cataloguing. The most important project is the Resource Description Framework (RDF) [RDF99], a W3C Recommendation.

RDF provides a foundation for exchanging machine-understandable metadata that describes Web resources, and thus enables automated processing of Web resources. RDF can be used, for example [RDF99]:

- in *resource discovery* to provide better search engine capabilities
- in *cataloguing* for describing the content and content relationships available at a particular Web site, page, or digital library
- by *intelligent software agents* to facilitate knowledge sharing and exchange
- in *content rating*
- in describing *collections of pages* that represent a single logical "document"
- in describing *intellectual property rights* of Web page authors,
- for expressing the *privacy preferences* of a user as well as the *privacy policies* of a Web site
- with *digital signatures* to build a "Web of Trust" for electronic commerce, collaboration, and other applications.

The basis of RDF is a syntax-independent data model for representing named properties and their values. RDF properties can represent both attributes of resources (thus correspond to traditional attribute-value pairs) and relationships between resources (thus correspond to the traditional entity-relationship diagram). The core RDF data model is defined in terms of:

1. A set of *Nodes* (*N*)

2. A set of *PropertyTypes* (*P*), a subset of *N*

3. A set of triples called *Triples*, whose elements are called *properties*. In a Triple {*p, r, v*}, *p* is a member of *P*, *r* is a member of *N*, and *v* is either a member of *N* or an atomic value (for example, a Unicode string).

In this data model, both the resources being described and the values describing them are represented as nodes in a directed labelled graph. The arcs connecting pairs of nodes are labelled by property types. This is represented diagrammatically as:

$$[r] \xrightarrow{p} [v]$$

and can be read as "*v* is the value of property *p* for resource *r*", or "*r* has property *p* with value *v*". For example, the statement "John Smith is the Creator of the Web page http://someweb/somedoc.html can be represented as:

$$[\texttt{http://someweb/somedoc.html}] \xrightarrow{\texttt{creator}} \texttt{"John Smith"}$$

RDF statements can be represented in XML (with the use of the namespace mechanism). For example, the above statement can be written as:

```
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:dc="http://purl.org/metadata/dublin_core#">
    <rdf:Description about="http://someweb/somedoc.html">
      <dc:Creator> John Smith </dc:Creator>

</rdf:RDF>
```

Figure 3-6: An RDF statement

More statements can also be included for the same resource to describe the resource in more details. For example, an XML element such as `<dc:Publisher>Addison-Wesley</dc:Publisher>` can be added inside the element `<rdf:Description>` above to give information about the publisher of the resource. RDF also defines other useful structures such as Bag, Sequence and Alternative to express collections of RDF statements, and a reification model to express statements about statements. (For instance, the sentence "Adam Smith believes that John Smith is the author of the resource http://someweb/somedoc.html" is a statement about another statement).

3.4 Conclusion

The data models described here, especially HyperWave [HYP00] and those of the XML family, have solved, or provided a foundation for solving, a range of problems, primarily manageability, maintainability, and extensibility. This demonstrates the power of the data modelling approach. There are disadvantages to the approaches that have been taken, however. For instance, HyperWave uses a commercial database, which is not an open system like the Web, to manage and maintain documents. XML is only an "enabling technology" [USD98], and therefore only provides a foundation for solving problems, rather than solutions themselves. More importantly, there are problems that remain unsolved or partially solved even with the advent of XML and its associated technologies. These are statelessness (Section 2.2.2.6), customisability, adaptability, and evolvability, which may be summed up in one single issue, evolvability (Section 2.2.1.5). Specifically, the existence of XML itself raises many questions. For instance, XML provides the expressive power that allows for the creation of various DTDs meaningful to differing communities, but how do these DTDs fit together? Can elements of different DTDs interact with one another and to what extent? Is it necessary to develop a separate set of tools and supporting software for each DTD? Is it possible to make the supporting software and tools interoperable across different DTDs, and to what level is this possible? Is it possible to derive a DTD from an existing one, or must each DTD be a completely new entity? If this is possible can the derivation process be automated, and to what extent can automation be supported? These questions can also be summarised in one issue: the self-evolving capability of the Web, or the Web evolvability. This topic will be investigated in the subsequent chapters of this thesis.

Chapter 4 System evolution

There has not been any formal study on Web evolution or Web evolvability that we are aware of to this date. Literature on system evolution in Computer Science, however, can be found in some other areas. Some well-established fields include Database Schema Evolution (DBSE), meta-modelling, and evolutionary algorithms. This chapter reviews work that is relevant to our study in these fields.

4.1 Schema evolution in Object-Oriented Database Systems (OODBS)

A database is a collection of related data, which represents some aspect of the UoD (Universe of Discourse), and is designed for a specific purpose [ELM94]. There are many types of database systems, depending on how data are organised into logical data models. Some of the most well-known data models are [ELM94, HAL95]:

- The *network data model*, which was first defined by the Conference on Data Systems Languages (CODASYL) Data Base Task Group (1971), is fairly complex. Facts are stored either in records or as record links. A record field may contain a single value, a set of values, or even a set of value-groups. A record link connects record types in an owner-member relationship (for example, an employee's record is "owned" by a department record). These connections may form networks, for example, a record type may have many owners as well as owning many record types. A query across two record types has to rely on a record link (which provides an access path) being predefined between them. This makes a database application less flexible and its management complicated. For example, access paths must be added for some new queries and, as the application structure evolves, internal optimisation efforts can be easily undone.

- The *hierarchical data model*, which was developed at IBM, is simpler. It is specifically designed for application to a hierarchical structure (for example, a computer file system). Facts are stored in records or record links as in the network model, but record fields can only hold single values, and record types are linked in a parent-child relationship where a child can have only one parent. This model provides good performance for hierarchical applications, but is less suited to other applications. In addition, as with the network model, it still depends on predefined access paths for some queries.
- The *relational data model*, defined by E.F. Cood in 1970, introduced an even simpler model. Here all the facts are stored in tables, including their constraints and relationships. This allows tables to be treated as mathematical relations. Complex queries can be made across multiple tables in an adhoc manner, and application management is simpler, since there is no need to separately define access paths.

Object-oriented database systems (OODBS) arrived later, in 1986, pioneered by Stonebraker et al. [STO86, KEM87] with their product POSTGRESS. These systems represent a merge between object-orientation and relational databases. They use concepts of class, class hierarchy, inheritance, instance variable (or *attribute* in object-oriented terms) and method from the object-oriented paradigm to describe data in terms of their semantics, behaviour and structure. OODBSs add behaviour to database objects and offer the well-known advantages of object-oriented technology such as manageability, maintainability, reusability and extensibility [BRA93]. OODBSs (rather than other kinds of database systems) are related to our work because we also investigate the use of object-orientation in our solution. In this section we look particularly at the issue of schema evolution in OODBSs.

Database Schema Evolution (DBSE) refers to the ability to change the description that models a particular database system dynamically, and to the consistent management of these changes. In an OODBS it involves such operations as addition or removal of an instance variable, a method, or a class. Two fundamental issues have been identified with DBSE, which are (i) taxonomy and semantics of schema evolution, and (ii) change propagation.

4.1.1 Taxonomy and semantics of Database Schema Evolution

This issue has been approached in current object-oriented database management systems (OODBMSs) in an informal way by using invariants to ensure the consistency of the database schema and rules to guide the maintenance of these invariants [PET97]. The sets of invariants and rules, however, vary in different OODBMSs, depending on the underlying object system and the choices of the system designer. Orion [BAN87], which was the first system to introduce this approach, defined a widely accepted taxonomy and semantics of schema changes, invariants, and rules to be observed when applying change operations.

4.1.1.1 Invariants of schema evolution

Five invariants have been defined in Orion [BAN87], which are presented in Table 4-1 below:

Invariants	Explanations
Class lattice	The class lattice of the system is a singly rooted and connected, directed acyclic graph (DAG), with uniquely named nodes and uniquely labelled edges, where every node is reachable from the root.
Distinct name	All instance variables and methods of a class, whether *native* (defined in that class) or inherited, must have distinct names.
Distinct identity (origin)	Uniqueness and traceability must be ensured for the origin of definition of all instance variables and methods of a class, whether they are directly or indirectly inherited from a single or multiple superclasses.
Full inheritance	A class must inherit all instance variables and methods defined in each of its superclasses, except for when this causes a violation of the distinct name and distinct identity invariants.
Domain compatibility	The domain (or class) of an inherited instance variable must be either the same as the original one in the superclass, or a subclass of the original domain.

Table 4-1: Invariants for Database Schema Evolution in Orion [BAN97]

4.1.1.2 Rules of schema evolution

As there can be more than one way to maintain the invariants, Banerjee et al. [BAN87] established twelve rules to guide the preservation of these invariants. These rules fall into four categories: default conflict resolution, property propagation, DAG manipulation, and composite object.

4.1.1.2.1 Default conflict resolution rules

These rules deal with the distinct name and distinct identity invariants by specifying a selection scheme over the conflicting elements.

Rule 1: A locally defined instance variable or method overrides an inherited one with the same name.

Rule 2: When two or more superclasses of a class *C* have an instance variable or method of the same name but distinct origin, *C* will inherit the variable or method of the first one (i.e. the left-most node among the nodes representing these superclasses in the class lattice) among the conflicting superclasses.

Rule 3: When two or more superclasses of a class *C* have instance variables of the same origin, *C* will inherit the variable of the more specialised domain, or of the first superclass among the conflicting ones if the first case is not applicable (i.e. the instance variables are over the same domain, or their domains are not of superclass – subclass relationship).

4.1.1.2.2 Property propagation and change rules

These rules dictate what changes in the contents of a class should propagate to related classes and to what extent.

Rule 4: Changes to the properties of an instance variable or method in a superclass propagate to all the subclasses that inherit them, unless these properties have been redefined in the subclasses.

Rule 5: The addition of, or name change to, an instance variable or method in a superclass should propagate to only those subclasses that do not introduce new name conflicts. This rule aims at resolving conflicts that may be introduced by Rule 4.

Rule 6 (domain change rule): The domain of an instance variable can only be changed to be more generalised, and can only be as generalised as that of the original instance variable.

4.1.1.2.3 DAG manipulation rules

These rules deal with the modification of classes and their relationships in the class lattice.

Rule 7 (edge addition rule): When a class *C* acquires a new superclass, that superclass will be the last in the list of the superclasses of *C*.

Rule 8 (edge removal rule): If a class *A* is the only superclass of class *B*, and *A* is removed from the superclass list of *B*, then all of *A*'s superclasses will become immediate superclasses of *B* in the same order as they are with *A*. In this respect if the root class is the only superclass of *B*, it cannot be removed from *B* (class lattice invariant).

Rule 9 (node addition rule): A newly added class will assume the root class as the default superclass if no other superclass is specified.

Rule 10 (node removal rule): The removal of class *C* first requires that all edges to it and from it be removed. Rule 8 must be applied if any edge is the only one from *C* to a subclass of *C*. In addition it is not possible to delete a system-defined class.

4.1.1.2.4 Composite object rules

In Orion a composite class is one that is connected to one or more classes with the aggregation (or is-part-of) relationship. Component objects participating in this relationship are dependent on the composite (parent) object in that they rely on their parent for their existence.

Rule 11 (composition link rule): A composite instance variable may be changed to non-composite (by dropping its composite link property), but not vice versa.

Rule 12: If a composite instance variable is changed to non-composite, the parent object *A* disowns the component object *B* (referenced by the instance variable). *A* continues to reference *B*, but the deletion of *A* does not also cause *B* to be deleted.

4.1.1.3 Taxonomy and semantics of schema evolution

Orion expresses the semantics of schema evolution using the invariants and rules specified above for each supported schema evolution operation. These operations can be broadly categorised into three groups: (i) Changes to the class definition, such as adding, removing or modifying instance variables or methods; (ii) Changes to the generalisation/specialisation relationships between classes; and (iii) Changes to the class as a whole such as adding or removing an entire class. The schema change taxonomy is as follows [BAN87]:

1 Changes to the contents of a node (class) (or changes to the class definition)

  1.1 Changes to an instance variable

    1.1.1 Add a new instance variable to a class

1.1.2 Drop an existing instance variable from a class

1.1.3 Change the name of an instance variable of a class

1.1.4 Change the domain of an instance variable of a class

1.1.5 Change the inheritance (parent) of an instance variable (inherit another instance variable with the same name)

1.1.6 Change the default value of an instance variable

1.1.7 Manipulate the shared value of an instance variable

1.1.7.1 Add a shared value

1.1.7.2 Change the shared value

1.1.7.3 Drop the shared value

1.1.8 Drop the composite link property of an instance variable

1.2 Changes to a method

1.2.1 Add a new method to a class

1.2.2 Drop an existing method from a class

1.2.3 Change the name of a method of a class

1.2.4 Change the code of a method in a class

1.2.5 Change the inheritance (parent) of a method (inherit another method with the same name)

2 Changes to an edge (or changes to the generalisation/specialisation relationships between classes)

2.1 Make a class *S* a superclass of a class *C*

2.2 Remove a class *S* from the superclass list of a class *C*

2.3 Change the order of superclasses of a class *C*

3 Changes to a node (or changes to the class as a whole)

3.1 Add a new class

3.2 Drop an existing class

3.3 Change the name of a class

4.1.2 Change propagation

Change propagation considers the strategy employed to update instances affected by schema changes to maintain data consistency throughout the database system. Three major techniques have been used in current OODBMSs, which are *screening*, *conversion* and *filtering*. All of these techniques explicitly *coerce* instances to reflect changes of the schema, but coercion takes place at a different time and in a different way in different techniques. This section presents a review of these techniques [PET97].

In *screening*, each schema change generates a conversion program that can independently convert objects to reflect the new schema definition. Actual coercion does not happen immediately but is delayed until the object is accessed. As it is possible to apply multiple schema changes to one class, multiple conversion programs might be invoked at the same time when an object of that class is being accessed. Screening suffers from performance penalties during access to object, especially if several conversion programs need to be applied. It can also increase the system overhead due to the need to keep track of what objects still need to be applied to a particular conversion program.

The *conversion* technique converts affected objects immediately on each schema change. This approach causes processing delay during the schema modification phrase, but not during the object access.

The *filtering* approach does not propagate changes to objects, but creates a different version of a schema for each schema change, and the old objects remain with the old version while new objects are created for the new version. The filters maintain consistency between versions and handle problems that arise when objects of different versions try to communicate with each other. In this respect filters also play the role of coordinating and inter-linking between different versions of schemata, maintaining the interoperability among their instances. This approach introduces the overhead for maintaining multiple versions and the filters between them.

4.1.3 Conclusion

In brief, schema evolution in OODBSs is about allowing for dynamic changes in the class definition and class lattice of an application while still maintaining data and schema consistency. These changes start at the schema level and propagate down to the affected instances. As a result, an evolution operation can create a complex chain effect to all affected classes and instances. This is complicated and difficult to implement, hard to standardise (since the underlying object models differ across OODBSs) and incurs penalties in performance or overhead. Recently, Peters R. J. et al. [PET97] have developed an Axiomatic Model of Dynamic Schema Evolution (DSE) in Object-Based Systems

(OBS). This model provides a formal foundation to describe the semantics of changes in various OBSs so that they can be compared. But it still does not address problems resulting from the impact that a schematic change has on the system. The taxonomy for schema evolution described in this section however, will serve as an additional guideline in the design of our own evolutionary model, and in comparing our system with systems such as Orion (in terms of support for schema evolution) (see Section 8.8).

4.2 Meta-modelling – a formal basis for information system evolution

DBSE has been made possible because of the availability of a set of information that describes the database schema, which is commonly referred to as meta-data[7], or a data dictionary. Meta refers to stepping back from something to survey its context and make assertions about it and that context [MAE88]. Simply speaking, meta-data is "data about data", or "data that describes data". Access to this set of meta-data allows the OODBMS to maintain the schema consistency and handle version conflicts during schema changes. Automatic manipulation of schema requires meta-data to be formally represented in models, by so-called schema repositories or meta-databases. These repositories can differ across distributed information systems and change over time. To ensure interoperability and adaptability of these systems, it is necessary to have models that describe these models (repositories), or meta-models [CON93, JAR98].

Meta-models are models about models, or in data modelling terms, a model is an *instance* of its meta-model. As meta-models are models themselves, they provide the abstraction and constraints needed to interlink the elements of different data models in a heterogeneous environment and along their evolution. Meta-models can be recursively nested, thus models abstract from instances, meta-models abstract from models, meta-meta-models abstract from meta-models, and so on [JAR98]. It has been shown by Kotterman [KOT84], and widely accepted [JAR98, UML98], that four levels of instantiation are needed for the evolution and definition of complex information system. As [JAR98] has pointed out, the architecture defined in the ISO/IEC[8] Information Resources Dictionary Standard (IRDS) Framework [IRD90] is designed based on the same principle with four levels interlocking

[7] To be more precise metadata should be called meta-information. The term metadata however, has been more popular and will be used in this thesis.

[8] ISO is the International Organization for Standardization. IEC is the International Engineering Consortium.

each other in pairs (see Figure 4-1). [JAR98] has provided an excellent summary of this architecture which is described below:

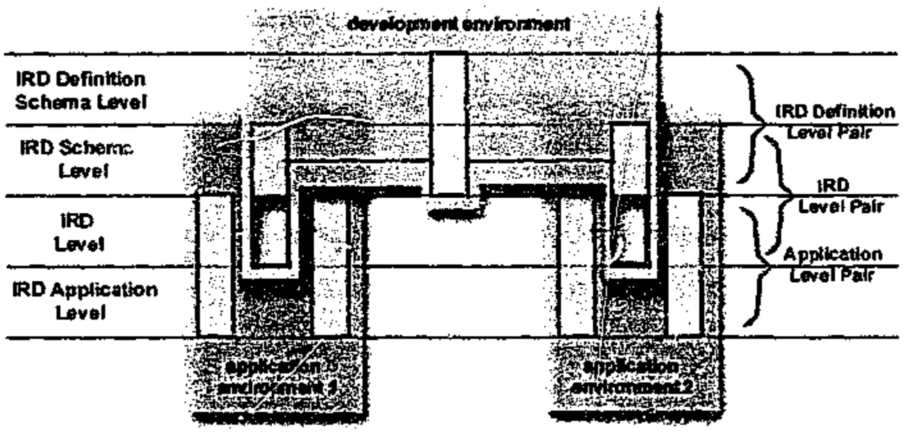


Figure 4-1: The ISO/IEC IRDS architecture

4.2.1 ISO/IEC Information Resources Dictionary Standard (IRDS) Framework: a meta-model architecture

The IRDS architecture is designed to interlock distributed application usage with distributed application development. This is achieved in a four-level meta-model architecture that we describe from bottom to top (see Figure 4-1):

- The *Application Level* includes application data and program execution traces. This corresponds to the instance level of a class-based language.
- The *IRD Level* includes database schemata and application programs, plus any intermediate specifications, and also specifications of non-computerised activities (for example, workflows). It can also contain traces of development processes interlinking these specifications. This corresponds to the class level of a class-based language.
- The *IRD Definition Level* specifies the languages in which schemata, application programs and specifications are expressed, and possibly also include the specification of their static and dynamic inter-relationships, for example design process models. This corresponds to the meta-class level.
- Finally the *IRD Definition Schema Level* specifies a meta-meta-level model (M2-model) according to which the IRD Definition level objects can be described and interlinked.

As Figure 4-1 shows, these four levels are grouped into interlocking level pairs. A level pair can be intuitively understood as a database where the upper level is the database schema and the lower level the database instance. The architecture interlocks level pairs in that the schemata of level pairs at the lower level can be coordinated by the database state of a level pair (dictionary) at the next higher level, thus creating a distributed database:

- *Application Level Pairs* correspond to traditional application databases, consisting of a schema and a database state
- *IRD Level Pairs* correspond to data dictionaries, meta-databases or repositories. At runtime, they can serve as coordinators for distributed systems. At system evolution time, they serve as design databases.
- *IRD Definition Level Pairs* serve the same purpose for distributed evolution environments, linking multiple heterogeneous data dictionaries or design environments.

Interlocked application level pairs and IRD level pairs form a *distributed application environment,* whereas interlocked IRD level pairs and IRD definition level pairs form a *distributed development environment.* Thus, the architecture provides the principal concepts for integrating the usage and evolution of distributed systems. The concepts implied in this architecture are found in the current database context as well as in many meta-modelling tools, although they may be only partially supported.

4.2.2 Conclusion

While schema evolution considers adaptability, and possibly also interoperability, at the schema level for one particular database system, information system evolution is concerned with the same issues at the same level for multiple database systems across a distributed information system. In both cases some description at the meta-level is necessary to maintain system consistency, and to cooperate and interlink elements in a heterogeneous environment. This concept is very useful in the design of an evolvable system. Elements or individuals that diverge in the course of evolution may be able to maintain their compatibility or interoperability through levels of meta-modelling. This idea is exploited in our design as explained in the subsequent chapters.

4.3 Evolutionary Algorithms

Evolutionary Algorithms (EAs) are "a class of direct, probabilistic search and optimisation algorithms gleaned from the model of organic evolution" [BÄC96]. Thus the problem of EAs is *optimisation*, which has a different focus from the issue of evolvability of the Web. On the one hand, the ultimate objective of an EA is to find an optimum in a particular search space by applying an evolutionary strategy. Typically, once such an optimum (or some termination condition) has been achieved, the algorithm will terminate. In the context of artificial life, EAs do *not* answer questions such as, "how is a new species formed?"[9] Or, "how do members of different species interact or communicate with each other?" On the other hand, evolvability is concerned with the *evolutionary mechanism*. Its problem is to identify *what* features (for example, structural, functional, or genetic) are required of an individual or a population (an "evolutionary entity") for it to evolve itself, and *how* the elements of these features work together for the evolution of the involving entities. Evolvability necessarily involves designing an evolutionary path, and defining the behaviour and interaction of evolutionary entities. Thus it can answer the questions above that EAs cannot. In addition, all EAs work on the basis of probabilistic or random mutation or recombination, which may not be applicable on the Web, where changes should most likely be directly controlled. They both involve evolution, however, and knowledge of EAs may be applicable or relevant to our research questions. We therefore present in this section, a summary of a representative and probably the most widely known EA, namely the (traditional) Genetic Algorithm (GA).

The Genetic Algorithm (GA) was invented in 1975 by Holland [HOL75], a computer scientist and psychologist at the University of Michigan. As its name suggests, the GA simulates the Darwinian evolutionary process at the genetic level. It does this by mathematically representing natural genetic information using bits and bitstrings. In nature, genetic information is encoded using a quadruplet system of the four so-called *genetic alphabet* A, T, G, C. In the GA, the coding system is mainly binary with two bits 0, 1. For simplicity we shall assume a binary coding system only. An optimisation problem is represented as a set (population) of bitstrings. These strings are modelled after chromosome strings, where each string represents an individual in the population of strings, and has an associated *fitness* value. Repetitive execution of the algorithm on this population produces strings with varied patterns to form a new population, progressively selecting the strings with the highest fitness values in each loop (generation). This process is based on the Darwinian principle of naturally occurring genetic operations (sexual crossover and mutation) and survival of the fittest.

[9] This question is actually not yet answered in biology at the genetic level.

4.3.1 Representation and fitness evaluation

As stated above, using the GA an optimisation problem is represented as a population of bitstrings of fixed-length. For example, in the "Hamburger Restaurant" problem described in [KOZ92], a manager has to find out the best business strategy among three binary decisions for his four restaurants:

- Price: Should the price of the hamburger be 50 cents or $10?
- Drink: Should wine or cola be served with the hamburger?
- Speed of service: Should the restaurant provide slow, leisurely service by waiters in tuxedos or fast, snappy service by waiters in white polyester uniform?

In this problem each decision has two possible choices, which can be represented as either 0 or 1. There are three decisions, therefore a business strategy can be represented as a bitstring of length 3, and the search space is $2^3 = 8$. This constitutes the *representation scheme* of the problem, which is shown in Table 4-2.

Price	Drink	Speed	Binary representation
High	Wine	Leisurely	000
High	Wine	Fast	001
High	Cola	Leisurely	010
High	Cola	Fast	011
Low	Wine	Leisurely	100
Low	Wine	Fast	101
Low	Cola	Leisurely	110
Low	Cola	Fast	111

Table 4-2: Representation scheme for the Hamburger Restaurant problem

Since there are only four restaurants, four strategies from the above are randomly selected and initially assigned to each of the restaurant as shown in Table 4-3. This is called generation 0, or the *initial random generation*.

Restaurant number	Price	Drink	Speed	Binary representation
1	High	Cola	Fast	011
2	High	Wine	Fast	001
3	Low	Cola	Leisurely	110
4	High	Cola	Leisurely	010

Table 4-3: Initial random generation (generation 0)

The fitness function is a decoding scheme that maps a bitstring to a real number. For simplicity, in this example, the fitness value for each strategy has been made equal to the decimal equivalent of the bitstring as shown in Table 4-4.

	Generation 0	
Restaurant	String	Fitness
1	011	3
2	001	1
3	110	6
4	010	2
	Total	12
	Best	6
	Worst	1
	Average	3.00

Table 4-4: Observed values of the fitness measure in generation 0

4.3.2 Selection

In the GA an individual is selected (to form a mating pool or to participate in a sexual recombination) with a probability p_s proportionate to its fitness. For instance, in the population represented in Table 4-4 above, the selection probability of string X_3 (110) is:

$$p_s(X_3) = \frac{f(X_3)}{\sum_{i=1}^{4} f(X_i)} = \frac{6}{12} = 0.5$$

that is, the string X_3 (110) has 50% chances of being selected.

Applying the GA on the population of generation 0 above, individuals are first selected to form a mating pool. Since the maximum number of individuals that can participate in the mating pool for our problem is 4, we expect that string X_3 (110) will appear 0.5x4 = 2 times in the pool. Similarly we can calculate the selection probabilities of other individuals in the population: p_s (X_1) = 0.25, p_s (X_2) = 0.08, p_s (X_4) = 0.17. We expect to see X_1 and X_4 to appear once each in the mating pool, while X_2 would disappear. The overall fitness of the population has increased (from the previous average of 3.00 to 4.25, and the worst-of-generation from 1 to 2) at the cost of a reduction in genetic variety. Table 4-5 shows the mating pool created after the fitness-proportionate selection process.

i	Mating pool	$f(X_i)$
1	011	3
3	110	6
3	110	6
4	010	2
Total		17
Best		6
Worst		2
Average		4.25

Table 4-5: Mating pool after selection

4.3.3 Recombination

Recombination or crossover in the GA is a sexual operator that selects two parents from the mating pool and recombines them to form two new individuals. Parents are selected proportionate to fitness, or can be based on a uniform random distribution if individuals in the mating pool has already been fitness-proportionate selected. Crossover allows new points in the search space to be tested. It is controlled by the crossover probability p_c, which specifies the percentage of individuals in the mating pool that participate in the operation. For example, in the mating pool shown in Table 4-5 above, suppose p_c = 0.5, then 4x0.5 = 2 parents will be selected for crossover. (Other individuals are simply reproduced into the new generation). Since the mating pool has been selected proportionate to fitness, two individuals can be randomly selected, for example strings X_1 (011) and X_3 (110).

A crossover point is randomly chosen[10] between 1 and the length of the string $L - 1 = 2$. Two offspring are created that contain fragments of the two parents' strings starting from the crossover point. In our example, suppose the crossover point is 2, each parent X_1 and X_3 will contribute fragments 01- and 11-, and give remainders –1 and –0, respectively. An offspring is produced by combining the fragment of one parent and the remainder of the other. Thus one offspring in our example will be 010 and the other is 111. A summary of this result for both generations is given in Table 4-6.

Generation 0				After selection		Generation 1		
i	String	Fitness	Selection	Mating	Fitness	Cross	X_i	$f(X_i)$
1	011	3	0.25	011	3	2	111	7
2	001	1	0.08	110	6	2	010	2
3	110	6	0.50	110	6	-	110	6
4	010	2	0.17	010	2	-	010	2
Total		12			17			17
Best		6			6			7
Worst		1			2			2
Average		3.00			4.25			4.25

Table 4-6: Summary of a possible outcome for generations 0 and 1

After crossover, the fitness of the new population is evaluated, and the GA is executed again to produce new generations of individuals until some *termination criterion* (for e.g. an optimal individual is found) is satisfied. As can be seen from Table 4-6, in our example, the overall fitness has increased for all the best-of-generation, worst-of-generation, and average, from generation 0 to generation 1. This example illustrates how the GA can improve the population and individual fitness from one generation to another.

4.3.4 Mutation

The mutation operator in the GA is only a "background operator" that randomly selects a string in the mating pool, and randomly selects one single bit in the string to invert. The rate of applying the

[10] Later development of GA introduced the crossover operator, which implements different strategies for picking up the crossover point(s). We do not discuss these strategies in this thesis.

mutation operation is controlled by the mutation probability $p_m \in [0,1]$, which is usually very small so that mutation seldom happens in the GA. Mutation was introduced into the GA by Holland [HOL75] to restore the "lost genes" that have become extinct due to the exploitative effect of the fitness-proportionate selection and cannot be regained by crossover.

4.3.5 Conclusion

Operating primarily on crossover and fitness-proportionate selection, the GA can produce populations with improved average and individual fitness. Mutation is used as a secondary operator to restore "genes" that have been lost and cannot be regained by crossover. The GA is typically executed repeatedly until a termination criterion is satisfied: e.g. an optimal solution is found or a maximum number of generations is reached. The problem of GAs in particular, or EAs in general, is optimisation. This is different from ours, which is evolvability. The underlying concepts in EAs, however, which are based on Darwinian evolutionary theory, are closely related to what we propose. EAs provide very good examples of how we can use biological evolution as an analogy to develop powerful methods for solving computing problems.

Chapter 5 The Multimedia Document Model (MDM)

We now start describing our own work for this project. As has been pointed out in Sections 2.1, 3.1 and 3.2, the Web was initially designed as a document-centred system. Although it has developed into a highly interactive system, this feature still holds true, as interactive components are only embedded objects (or scripts) brought to users by means of documents. The document-centred characteristic is preserved in our system, and developed in two stages. The first stage models the traditional paper-based document system. The second stage develops the first-stage model into one with Web-specific characteristics. In this chapter, the Multimedia Document Model (MDM), the first-stage model of our system, is described.

5.1 Design criteria

The Multimedia Document Model is designed to satisfy the following criteria:

- Simplicity: simplicity has been one of the key factors of the success of the Web, and is preserved in our model. Simplicity means that our system is easy to understand and can be used by the general public. The system should be as intuitive as the current Web so that anyone can use it without requiring any special knowledge.

- Compactness: compactness means that the number of classes, attributes and methods in a class is kept to the minimum. Compactness contributes to simplicity since there is less to learn, and also gives room for extensibility and evolvability.
- Completeness: completeness means that the system must cover all existing elements in the traditional document system.

The system also possesses features of an object-oriented system, which are manageability, reusability, and flexibility.

5.2 The Multimedia Document Model (MDM)

This model is designed based on the current library document system and publishing systems such as LaTeX [LAM86], Standard Markup Language (SGML) [SGM86], and Hypertext Markup Language (HTML) [HTM98]. It follows object-oriented design techniques. Figure 5-1 shows the class diagram of the top level of the model, using Unified Modeling Language (UML) notation [UML98].

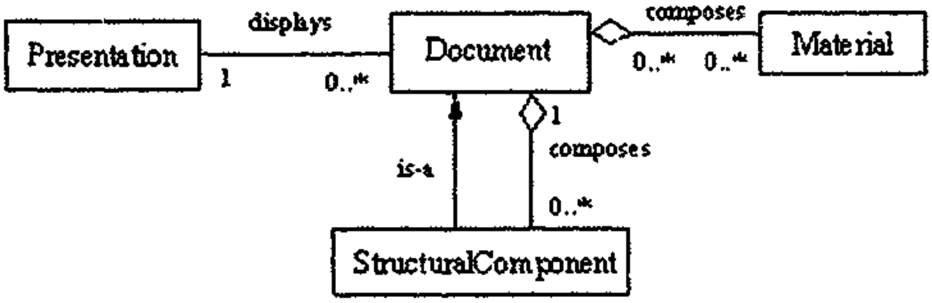


Figure 5-1: MDM class diagram - top level

In this model, a document is decomposed into objects, having internal state and well-defined behaviour. The object's internal state is defined by a set of attributes and its behaviour by a set of methods. A document can be viewed in three ways: *Structure, Material* and *Presentation.* In the *Structure* view (represented by the *StructuralComponent* class in Figure 5-1), the document is broken up into structural components, such as volume, part, chapter, and section. The *Material* view (represented by the *Material* class in Figure 5-1) accounts for the raw materials that compose a document, such as text and graphics (other types of media can be added due to the extensibility feature of object-orientation). The *Presentation* view (represented by the *Presentation* class in Figure 5-1) looks at the various ways (such as table, frame, list, reference entry, and article) information is presented to human readers.

5.2.1 The MDM classes

This section explains the model according to five major classes: *Document* (which represents the whole document) *StructuralComponent, Presentation, Material* (which represents the three views of the document) and *Publication* (which represents a major subclass of document). Figure 5-2 shows the second level class diagram of MDM.

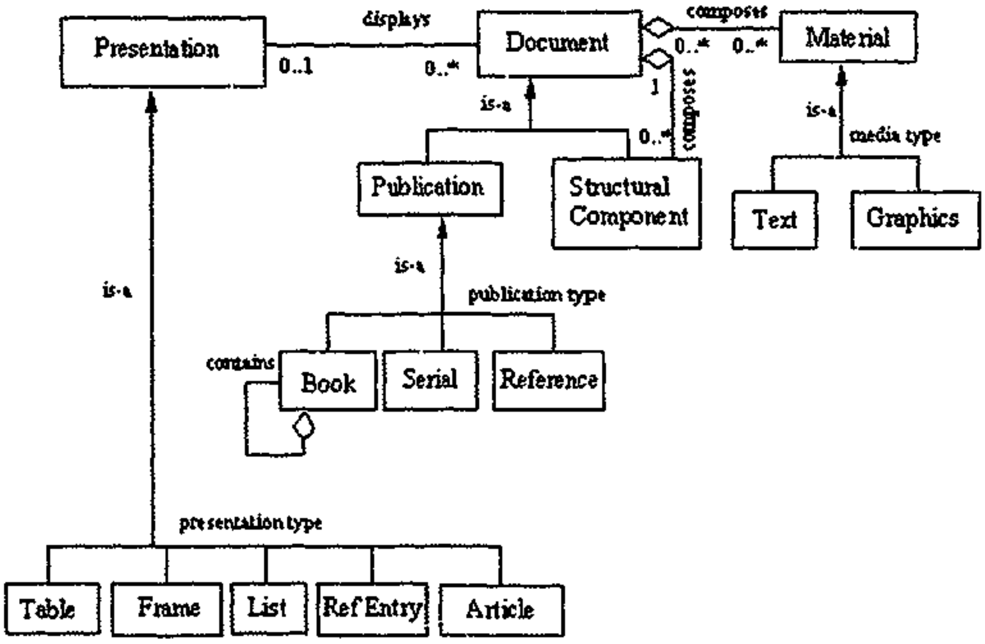


Figure 5-2: MDM class diagram - second level

5.2.1.1 Document

A *Document* object represents the entirety of a publication, and is constructed from the basic building blocks of *StructuralComponent* objects. Instances of the *Document* class correspond to the loosest form of publication. It is designed to completely cover any type of document in the traditional document system. These loose types can be, for example, unbounded documents, drafts, letters, or notes. Objects of the *Document* branch act as the interface of the Web system for human readers. Internally they are composed of other objects which may be only *StructuralComponent* objects (structured documents), or some *StructuralComponent* objects and some *Material* objects (semi-structured documents), or just *Material* objects (unstructured documents). The only two attributes that require non-null values are the object *id*, and *heading*, which underlines the fact that its content is semantically complete, that is, it must not contain just fragments of text or graphics which are not comprehensible to human readers. The *heading* also serves as a mnemonic identifier to human readers, but does not have a significant meaning for the system, and is not required to be unique in any domain.

5.2.1.2 Structure

As the *StructuralComponent* class inherits the composition relationship to itself from *Document*, a *StructuralComponent* object can contain other *StructuralComponent* objects recursively. This forms a flexible tree structure commonly found in paper-based document systems, such as a book containing many chapters, or a chapter containing many sections. This structure allows for the whole document to be modified, extended, or truncated on a component basis, without losing the coherence of its overall structure. *StructuralComponent* is the generic class for all structural components of a document, such as volume, part, chapter and section. Since all these structural components share the same properties and behaviour, it is appropriate to have one single generic class. This design makes the model simpler and more compact, and gives authors a larger degree of freedom when defining their own document, since the model does not differentiate between the various types of structural component. More importantly, the use of generic classes makes it possible for the system to evolve. (System evolution is discussed in Chapter 8.)

StructuralComponent objects hold two special attributes: *numberingScheme* and *view*. *numberingScheme* defines whether a collection of *StructuralComponent* objects should be regarded as a *sequence* (possibly numbered), or a simple unordered *set*, and also specifies the format for numbering them. The system can thus automatically generates numbers for structural components, reflecting their order and level within the document hierarchy, and formats them according to the specified *numberingScheme*. *view* assigns to a *StructuralComponent* object some value(s) that specifies whether that object should be (functionally) included (visible) in the parent object that it (declaratively) belongs to in the document tree. *view* allows for dynamic customisation, which will be explained in Sections 8.2 and 9.4.2. It is possible for a *StructuralComponent* object to have more than one value for *view*, so that it can be combined with other *StructuralComponent* objects in different ways, and different views can be supported for the same set of materials.

5.2.1.3 Presentation

The model also makes presentation components available, by associating objects of the *Presentation* branch with those of the *Document* branch. *Presentation* is an abstract class from which other presentation formats are derived. A *Document* or *StructuralComponent* object may be associated with zero or one *Presentation* objects (at a time), and a *Presentation* object may be associated with zero or more objects in the *Document* branch.[11] It is possible to have a repository of pre-defined *Presentation* objects for an author (or user) to choose from when designing (or viewing) a document. A *Document* or *StructuralComponent* object can be presented in a number of different formats such as *text chunk, table, frame, list, reference entry,* and *article. Text chunk* is a sequence of one or more ASCII characters. It is essentially a section of text, with one or more paragraphs. *Table, frame, list, reference entry* and *article*, are all defined according to the common sense criteria used in the traditional publishing industry. *Text chunk* is the default presentation for an object of the *Document* branch.

5.2.1.4 Material

Material objects hold the *real contents* that populate objects of the *Document* branches. They contain the unstructured, raw materials that are used to fill in the *Document* objects or their structural components. The actual data type of the content will be defined in the offspring classes of *Material*. In this version of MDM, it can be either text or graphics. This content may be semantically incomplete, that is, it may not be meaningful to human readers. For example, it can be a fragment of text from a part of a chapter of a book. This semantic incompleteness is reflected in the object design by the fact that no *title* or *heading* is required for an object of the *Material* branch. Objects of subclasses of *Material*, which represent different media types, can therefore be inserted at arbitrary places in a *Document* object. This concept is applicable to all media types, although currently, it is meaningful to *Text* objects only. This design makes it possible, for example, to have a chapter in a textbook with an image inserted in the middle. If the chapter is defined as one *StructuralComponent* object, then it will consist of three *Material* objects: two broken text pieces and one image (see Figure 5-3).

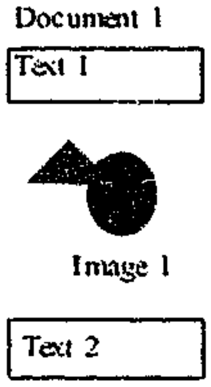


Figure 5-3 : Different media types in a document

5.2.1.5 Publication

Publication is the generic class for all types of published materials. It generally corresponds to any publication, normally bounded in size, that can be identified by an international identification number, usually an ISBN (International Standard Book Number) or an ISSN (International Standard

[11] It is possible to allow for the association of more than one *Presentation* object with a *Document* or *StructuralComponent* object, and to define a scheme to select a suitable presentation. For simplicity, we only allow the association of at most one *Presentation* object.

Serials Number). There are other more specific classes inherited from *Publication*, designed to handle specific features of different document types. All of these classes are based on the existing paper-based document systems found in most libraries. For compactness and simplicity, several library categories are merged into one single class in our model:

- *Book* includes such publications as textbooks, storybooks, manuals (non-reference), the Bible, edited books, and picture books. A *Book* object normally has one or several authors, but all cooperating in the one work, and typically contains items that are closely related to each other. For example, there are normally references from one section to another in a textbook. A book is typically a non-collective, non-recursive entity. Some exceptions are the Bible, a book of short stories, or a condensed book, which can all contain other books.
- *Serial* includes magazines, newspapers, journals, conference proceedings, research reports, bulletins, and newsletters. They are published periodically and all have some method for sequential ordering, either by number or by date, and their contents are usually collectively authored. They typically contain a collection of independent texts or writings, each of which is complete and comprehensible without the need to refer to or understand other items in the collection (although they may use other work outside or within the collection as references). Examples of such texts are articles, technical papers, reports, lecture notes, advertisements, and notices. A collection of serials can be bound into a volume, identified by a volume number.
- *Reference* covers all reference materials, such as dictionaries, handbooks, calendars of events, directories, and reference manuals. They normally contain many reference entries, which may be grouped into sections. Each entry usually consists of a reference term and its explanation or specification. The reference class has a special method, *sortEntry()*.

The design of our model allows the mapping of all existing document types. The *Book*, *Serial* and *Reference* classes cover most printed materials found in libraries. The *Publication* and *Document* classes account for any other types, both published and unpublished, structured and unstructured. Each offspring class of *Publication* is actually a merger of many library categories, and new classes are created only if there are important properties or methods that need distinguishing from each other, such as issue number for *Serial* objects, or *sortEntry()* method for *Reference* objects. The model is thus both complete and compact, since it covers all document types while the number of classes are kept to a minimum. Simplicity is achieved by compactness, and also by the fact that the model is based on familiar and well-understood concepts of the traditional paper-based system. This system has many *defacto* standards, for instance, those that are used to design the structural and presentational classes in our model. This provides a well-founded base, and makes the model robust in terms of changes and simple to use.

5.2.2 Three views of the MDM

In our model a document is viewed in three ways: *Structure, Material* and *Presentation.*

- In the *Structure* view, the document is broken up into building blocks, such as volume, part, chapter, and section, which are generically represented by the class *StructuralComponent* in our model (Section 5.2.1.2). By allowing such a mapping for a structured document, this view gives a solution to the document structure and document properties (these properties are: structure, material and presentation) problems mentioned earlier (Sections 2.2.2.1 and 2.2.2.4). Consequently, other derived issues, namely document management and maintenance (Section 2.2.1.3) and document design and authoring (Section 2.2.1.4) are also addressed. MDM does this by making the structural components of the document accessible, allowing for the document to be designed, authored, managed, maintained and manipulated structurally on a component basis.

- The *Material* view accounts for the various media types that compose a document, such as text and graphics (Section 5.2.1.4). Like the structure view, this view addresses the document properties issue (Section 2.2.2.4), and other derived problems, namely document management and maintenance (Section 2.2.1.3), and document design and authoring (Section 2.2.1.4). This is done due to the encapsulation of *Material* objects, which allows them to be created independently (in some editor), managed, maintained and manipulated on a component basis and according to their specific media types. The document can also be designed and authored separately from content (and presentation) details (see also Section 9.7).

  Another significant benefit is that *Material* objects can be reused in more than one document. This is very useful in a distributed environment such as the Web, where the same materials may be used in many documents (for instance, a company logo), or to produce customised documents. (Customisation in our system is discussed especially in Sections 8.2 and 9.4.2).

  Most importantly, the *Material* view helps solve the problem of statelessness (Section 2.2.2.6). It is noted that a document component hierarchy is very similar to a parse tree (or a Directed Acyclic Graph) of the flat representation of a given document. In this analogy, the materials associated with a document component become the terminals (leaves) of such a tree. In other words, every object of the *Document* branch must be "translated", or "resolved", to one or more objects of the *Material* branch. In this way, the *Document* object can be considered as the *virtual* container of system's contents, holding only the references to *Material* objects, which keep the *real* contents of the system. This allows for virtual contents to be distributed, and makes it possible to retain states at distributed (client) sites. Statefulness in our system is explained in detail in Sections 7.2.3 and 7.2.5.2.

- The *Presentation* view looks at the various ways information is presented to human readers. Presentation is only supported in a very limited way in this version of MDM. Other format and presentation issues have been intentionally left out for the next version of MDM, or for use with other mechanisms such as the Cascading Style Sheet (CSS) [CSS98] or Extensible Style Sheet (XSL) [XSL00]. All the offspring classes of the *Presentation* class, namely, *Table, Frame, RefEntry, Article* (Figure 5-2), represent the presentation view in our model. The presentation view makes it possible to present structural components in different formats, which is useful for document design, management and maintenance (see Section 9.7), and particularly, customisation (see Sections 8.2 and 9.4.2).

5.2.3 The MDM and HyperWave

The data model for the Web closest to the MDM is that of HyperWave [HYP00], described in Section 3.2.1. In the MDM, classes of the *Document* branch are comparable to the *abstract containers* in HyperWave, which are Collection, Sequence, MultiCluster and AlternativeCluster. Here a *StructuralComponent* object corresponds to a Collection or Sequence (depending on the value of *numberingScheme*), and also to a MultiCluster or AlternativeCluster (depending on the value of *view*) if the *StructuralComponent* object contains *Material* objects only. There are some differences between the two data models and/or their respective implementation approaches (which will be described in Chapter 9):

- Like HyperWave, MDM can support both static and dynamic customisation in a similar way, based on pre-defined grouping of structural components, or on their *view* attribute values, or on some (stored) selection criteria that are (dynamically) evaluated against other (possibly combined) attribute values. (These are comparable with HyperWave's Collection, access permission and stored query, respectively.)[12] Customisation in our system is explained in more detail in Sections 8.2 and 9.4.2.
- HyperWave keeps all objects in the server's database, while objects in our system are stored as flat files (or part of such a file) and *Document* objects can be wholly distributed at client sites. The use of a central commercial database does offer advantages such as facilitating link integrity

[12] A slight difference between the two systems is that in MDM such a customised document always retain its structure, whereas in HyperWave a stored query returns an unstructured Collection. Our approach is based on the design principle that the internal data content of a structural component should be private to the component, and user customisation should be carried out on a structural component basis only. We only remark the difference but do not comment on the advantages or disadvantages of either approach here.

maintenance and providing built-in search functionalities, but increases complexity and restricts the openness (public use) of the system. Most importantly, the distribution of *Document* objects allows users to keep multiple customised documents at their client sites (as opposed to only one Home Collection for a HyperWave user). Such distribution has a significant meaning in our design as has been mentioned in Section 5.2.2 above (and can be seen in the rest of this thesis).

- HyperWave does not really capture the presentational aspect of a document in its data model and is thus limited in providing for presentational customisation (see Section 3.2.1). By contrast, in MDM, presentation is modelled as a class that is associated with the *Document* class (and all its subclasses, such as *StructuralComponent*). This gives MDM the potential to allow its users to control not only the global presentation of the whole document (as HyperWave does) but also the local presentation of individual structural components.[13] More importantly, the explicit capture of the presentational aspect has a significant meaning in the adaptability and evolvability of our system (see Sections 7.2.5.2, 8.2 and 9.4.2).

5.2.4 Summary

In summary, the MDM inherits the features of flexibility, manageability and reusability from object-oriented technology. It covers virtually all document types, while remaining relatively compact and simple. This is due to the design principle of using generic classes and a minimal number of attributes and methods. The three distinguished properties of a document, namely structure, presentation and material, are identified, and the model is designed to reflect these views. The three views together make the document components accessible in various ways, and allow for the document to be assembled or disassembled structurally, its layout suitably controlled for components of different media types, and various formats presented.

[13] When a local presentation directive and a more global one are both effective on a document component, the effect of one directive on the other may not be the desired one. However, it is not within the scope of this thesis to deal with these issues (see Section 5.2.2).

Chapter 6 Life Design

The emergence of XML-based formats does not merely represent a slew of new competitors, but an ecosystem of interdependent document species.

R. Khare et al. [KHA98]

6.1 Preamble

It can be seen from the problem analysis in Section 2.2 that most problems with the Web relate to the growth of the Web, both in size (quantity) and in the underlying technologies (quality), over time. Quantitative growth, characterised by the exponential increase in number of Web documents and Web sites (Section 2.2), is directly related to the issues of getting lost in hyperspace (Section 2.2.1.1), violation of referential integrity (Section 2.2.1.2), and difficulties in data management and maintenance (Section 2.2.1.3). Qualitative growth, as indicated by the changes in HTML and the addition of a myriad of technologies (Section 2.2.2.2), underlines the design issues discussed in Section 2.2.2.

It can also be seen that deployment trends in the Web have changed increasingly from static to dynamic ones (Sections 2.2 and 3.1). The Web is no longer simply a shared information space where new nodes and their relationships can be recorded (Section 1.1). It is now being used as an interconnected space where documents can interact with each other and with servers and users, and generate other documents dynamically.

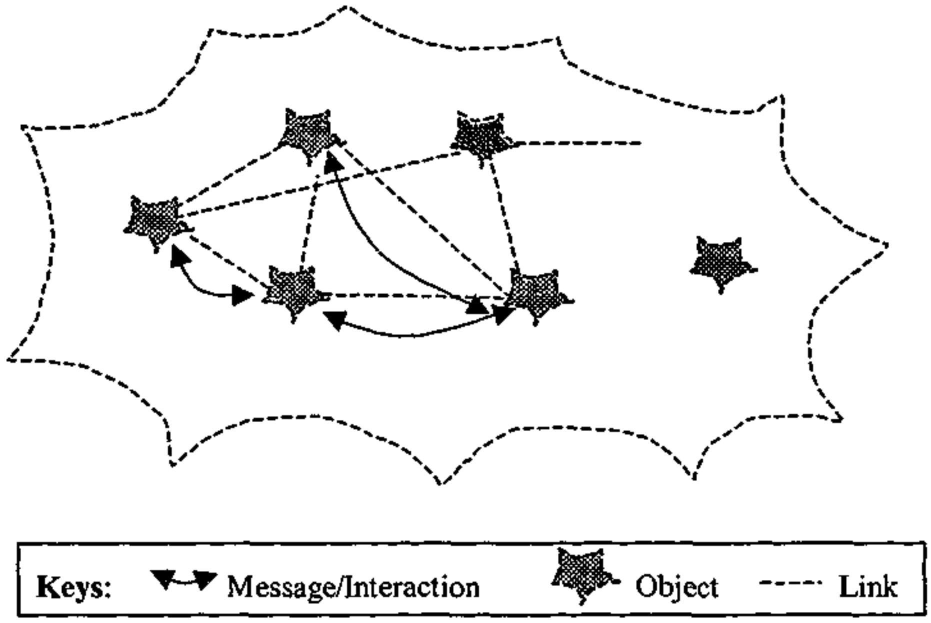


Figure 6-1: A schematic view of the Web

These factors of growth and deployment suggest a new view of the Web. It is not so much a static network of nodes and links (Figure 2-3), but rather a highly dynamic and rapidly growing body, full of active objects, constantly changing and interacting with one another (Figure 6-1). Here the basic dynamic elements are Web document objects, in their interconnected network and interactive environment with users.

As has been pointed out before in several places (Sections 2.2, 2.2.1.5, 2.3, 3.2.2, 3.4 and Chapter 4) this characteristic of the Web suggests a solution that can capture its dynamics, or liveliness, and growth. Taking our modelling approach into consideration (Sections 1.1, 2.3, and Chapter 3) such a solution must be a data model that can deal with future changes, that is, it can *evolve itself* in a *systematic* way.

Two candidate models have been considered for our problem: the socio-economic model, and nature. Using the socio-economic model as a guide [SMI82, SMI99], the Web can perhaps be considered analogous to a community, a Web site to a city or village, a Web document to a household, and a user to a citizen. This model may give an approximation to the reality of the Web, but it naturally captures the Web's social and economical aspects, rather than the technological one that we are working on. Besides, its major drawbacks are the complexity inherent in socio-economic-political issues, and the lack of a well-defined evolutionary theory, which is precisely what is being sought. Nature, on the other hand, has the compelling advantages of a well-understood and well-established Darwinian theory on evolution [DAR59, DAR69], and many related scientific evidences (for example, the discoveries of the *gene* by Gregor Mendel [COR93], and *DNA* (deoxyribonucleic acids) by Crick and

Watson [WAT83]). These provide well-founded knowledge on the mechanisms of biological life. Here, the Web may be considered analogous to Earth, a Web site to a geographical region, a Web document to a living organism, a document type to a species, and the communities of users and developers to the environment.

The use of biological metaphors in computer science is not new. About forty years ago, John von Neumann [NEU58] made one of the first computers by emulating biological systems. The components of his computer were designed after organs, and logic gates after the activity of neurons. The Genetic Algorithm (GA), invented in 1975 by Holland [HOL75], benchmarked the first successful application of Darwinian evolutionary theory in computer science with a simple yet powerful mathematic model (see Section 4.3). Richard Dawkins [DAW76, DAW95], who is considered an "astonishingly influential... revolutionary evolutionist" of this time, viewed life as "a process of digital-information transfer" [SCH95]. Dawkins extends Darwinian evolutionary theory into a unified one, which transcends the boundaries between natural and artificial evolution (by considering nature as an information processing system), and reaches into multidisciplinary areas of psychology, ecology, sociology, taxonomy, culture, and beyond. His ideas have particularly stimulated researchers of artificial life (a-life) and related fields.

With regard to the use of biological metaphors on the Web, Peter Small [SMA98] presented his ideas about *A-Life Avatar*, an artificial host *cell* on the Internet that can take various forms and behaviour, depending on the artificial *life form* that resides in it at one particular time. The behaviour of these *life forms* can be driven by textual commands sent in email messages. In his analogy, Web documents are viewed as *life forms* or intelligent agents. In another book, Small speculated on an analogy between molecules and objects (in object-orientation), with similar message passing mechanisms [SMA96]. Rohit Khare et al. [KHA98] used the term *document species* to denote *document type* in their WWW7 conference paper, which discussed the advent of XML as an "evolution" of Web documents. "Evolution" mentioned there, however, was only at the level of observed facts about changes in document types (in terms of syntax, style, structure and semantics) over time.[14]

Work in these areas has inspired, and given many useful insights into the search for a solution to our problem. In particular, Peter Small's analogies of molecule–object and life-form–document have

[14] Interestingly, it was also during this conference that Berners-Lee talked for the first time about the Web's evolvability [BER98], and our own first papers on this project were presented, proposing the application of biological concepts for the design of a *self-evolving* Web [NGU98a & b].

been used and developed further in our studies (see Section 6.3, in particular, 6.3.1).[15] Peter Small, however, did not consider evolution in his work; and none of the above work has dealt with the problem of evolvability. An evolutionary mechanism and evolutionary path for artificial life forms were not identified. The questions posed earlier (see Section 4.3) about *what* features are required of an individual or a population for it to evolve itself, and *hrw* the elements of these features work together in the course of evolution, have still not been answered.

In this chapter, we explore the many parallels between the Web and the ecological system in which life forms exist, interact and evolve. The differences between the two systems are simultaneously investigated, which help us to learn from nature, the most complex yet perfect system known to us in terms of evolvability. Section 6.2 summarises findings from relevant biological studies. Section 6.3 presents our observations of the Web as a "living" entity and shows how object-oriented technology can be applied to bring "life" to the Web. Missing factors in object-oriented design that are essential for this task are also identified. Section 6.4 attempts to present these concepts in an accurate way in *Life Design*, a software design methodology.

6.2 Biological life

Life is the term used to summarise the activities characteristic of living beings and living organisms, as opposed to dead ones and inorganic matters. These activities fall into two major categories: reproduction and metabolism. Generally an organism is considered alive if it is capable of performing both reproduction and metabolism [ALB83].

6.2.1 Reproduction

Reproduction is the process whereby a living organism produces its offspring. This is controlled by genetic information processing through which the hereditary material contained in the deoxyribonucleic acids (DNA) is passed from one cell or organism to another [ALB83, DEN83, KOR80].

Gene is the name given to a unit of heredity, found thanks to the work of the Austrian monk and scientist Gregor Mendel in 1866 (see [COR93]). The gene, which exists in the nucleus of the cell, consists of DNA molecules [WAT83], which, in turn, contain coded information capable of directing

[15] To be precise, Peter Small's analogy between "life form" and document is different from ours. In his concept of *A-Life Avatar*, a *life form* is really analogous to an embedded object in a document. In our design, a document itself is analogous to a biological life form.

the synthesis of specific proteins, and substances that make up the cell itself. A DNA molecule consists of two strands of four types of nucleotides (subunits of the DNA molecule), linked together like a twisted rope ladder. The order of these nucleotides determines the order in which amino acids are synthesised and ultimately all characteristics of the living organism. Through a process known as *template replication*, the DNA molecules duplicate themselves during the cell division process to form two new cells with identical genes. Just before the cell divides, the DNA unwinds and the nucleotides on opposite strands, which are only weakly bound, pull themselves apart from their pairs, exposing two separate single strands of nucleotides. Free-floating nucleotides come to combine with the nucleotides just separated. When all nucleotides have been paired up, the cell division process begins.

The discovery of DNA reveals its role as both a *model* from which specifications of proteins are defined, and as a *manufacturer* producing these proteins, which subsequently construct and define the characteristics of the living organism. The DNA is called the *code of life* since it defines the type (*species*) of the living organism and controls its creation and reproduction.

6.2.2 Metabolism

Metabolism refers to the sum of the chemical processes by which the cell transforms energy to perform life activities such as reproduction, growth, response to stimuli, and elimination of waste materials. There are two types of metabolic process: anabolism and catabolism.

Anabolism or constructive metabolism synthesises complex molecules from simpler ones with the storage of energy. This process is required for the growth and reproduction of new cells and the maintenance of tissues. Catabolism or destructive metabolism performs the reverse of anabolism, and breaks down complex molecules into simple ones with the release of energy. This process provides the energy needed for all external and internal physical activities (including response to stimuli), and is also responsible for maintaining the body temperature and the degradation of complex molecules into waste substances ready to be removed [ALB83].

6.2.3 Evolution

Evolution is the complex process of gradual change in the characteristics of a population of living organisms over generations to withstand changes in the environment. The evolutionary process was not understood until Charles Darwin, in his famous book, *On the Origin of Species By Means of Natural Selection* (1859) [DAR69, DAR77], enlightened human understanding on the matter. Since then, Darwinian theory has undergone many studies and there is still much debate on the evolutionary process. It has been generally agreed, however, that evolution is a process of repetitive selection of the fittest (called "survival of the fittest") among a variety of genes, through reproduction. The genetic variety comes from the introduction of new genes into the gene pool of a population of a species. These new genes may be the result of a random genetic combination from each parent in sexual reproduction, or a series of small *gene mutations*. (Gene mutations are mistakes in the ordering of the nucleotide strings when genes are being copied in the reproduction or cell division process.) If these genetic changes increase the survivability of the species, they will be retained through the natural selection process, and multiplied throughout the population, otherwise they will be disadvantaged and disappear [AYA79, AYA84, SMI93].

It can be seen that evolution, which starts at individual living organisms, can happen only if the following three conditions are satisfied: (i) a genetic change to seed the process, (ii) a factor to provide the driving force for the selection of the fittest, and (iii) a population in which individuals can connect, communicate and interact with each other to spread the change.

6.2.4 Species and speciation

Species is the basic category in the biological classification system, and is an attempt to classify organisms into groups that reflect the evolutionary processes underlying the similarities and differences between them. A species represents a distinct type of living organism, with characteristic shape, size, behaviour and habitat. A species remains stable over a long period of time. Members in a species can mate and produce offspring with one another, but do not breed with members of other species. Closely related species are grouped into a genus (plural genera). Genera are then grouped into families, families into orders, orders into classes, classes into phyla, and phyla into kingdoms [JEF78, SMI93].

Speciation is the complex process of forming a new species, and is believed to happen along the evolutionary process. Speciation probably starts first with extrinsic isolation, where a species becomes subdivided because of some extrinsic event such as a climatic change or a geographical separation. Secondly, the isolated population becomes genetically differentiated, possibly because its individuals multiply more quickly than there are chances to mate with individuals in other populations. Genetic divergence can happen as a result of natural selection or randomly. Thirdly, intrinsic isolation, some form of isolation within individuals among the differentiated populations, develops. This could be the result of some preferences in courtship or some genetic incompatibilities, making offspring of mating between individuals of differentiated populations no longer viable or fertile. Finally, the independence of a species is confirmed when newly separated populations continue to evolve independently, and may subsequently invade each other's habitats without hybridisation [DAR77, SMI93].

It can be seen that speciation is an extreme case of the evolutionary process, where extrinsic and intrinsic isolation maximises genetic divergence, leading to the formation of new groups whose individuals are no longer genetically compatible for mating.

6.2.5 Life at the molecular level

The above biological characteristics of life forms can be realised thanks to the support provided at the molecular level. Our body is made up of cells, which in turn, are made up of what are called *organic compounds*. All of these organic substances have one common constituent in their composition, which is *carbon* (C), an element that exists in all forms that have life. Carbon possesses two special features that cannot be found in any other elements [AND62, pp.725-749]:

- The first is the ability of carbon to combine with other elements to form numerous complex and branching chain structures, which are connected together either by *skeletal links* or as *side-chained functional groups*. The presence of a functional group alters characteristic properties of an organic compound. The –OH group (hydroxide radical) for example, makes the compound exhibit characteristics of an alcohol.
- The second is that the connections between elements may only be *weakly bound*. They can unwind to allow for connections to other elements to be made under certain conditions. In a molecule, a single bond is stronger than a double bond, which in turn, is stronger than a triple bond.

It is the composition of these compounds and the complex arrangement of their constituent atoms that create an almost endless variety of cells differing in hereditary components. The weak bindings in their structure play a crucial role in all metabolic reactions to form new substances, as well as in other life processes such as reproduction. The separation of the two strands of the DNA molecule at the weakly bound base as explained in Section 6.2.1 above illustrates the significance of weak bindings.

6.2.6 Summary

At the molecular level, the element that is central to life is carbon (C), due to its unique ability to form numerous *complex and branching chain structures* that are more or less stable, but can also *be broken and changed* under certain condition. It is the complex ways of combining molecules that supplies the vast genetic variety in cells, and the flexibility of these links (weak bindings can be broken and new ones formed) that allows for life processes such as reproduction and metabolism to take place.

At the genetic level, life is guaranteed by complex information processing through which *deoxyribonucleic acid (DNA)*, "the code of life", controls chromosomes and their genes to be copied exactly from cell to cell in the reproduction process.

At the biological level, life manifests itself in the capacity to perform two major functional activities, *reproduction* and *metabolism*. Reproduction is the process whereby a living organism produces its offspring, and is controlled by genetic information processing. Metabolism refers to the chemical processes by which a living organism transforms energy consumed from outside world to use in activities such as *reproduction, growth,* and *responsiveness to environment.* Metabolism is made possible through contact with the external environment and an internal network of molecules interacting with each other via complex chemical reactions.

Over time living organisms *evolve* into diversified and modified forms to withstand changes in the living conditions. It can be said that the evolutionary process, which takes place in each individual living organism, is enabled by internal genetic change, driven by the external forces of natural selection, and realised by the population's adaptation capabilities. *Speciation* is the extreme case in evolution, where individuals of genetically differentiated populations are no longer (genetically) compatible for mating.

6.3 Bringing "life" into the Web

As was pointed out in Section 2.2, Web documents are continually mounted with rich behaviour, and static contents are increasingly being replaced by highly interactive and dynamic contents. The Web, as it is deployed nowadays, in this way, bears many resemblances to the living world. The Web space is populated with Web objects uniquely identified by their URLs in much the same way our planet is populated with living creatures, each of which is a unique entity in the universe. Web objects are created and destroyed, and similarly, living creatures are born and die. The interconnectivity of the Web inherent in the Internet allows for a network of objects communicating and interacting with each other through various protocols; likewise, living creatures make connections, communicate and interact with each other through their own standards. And as living creatures respond to the environment in which they live, so Web objects can also respond to user inputs.

The Web as a whole appears to be a very lively system, but it is not "alive". In contrast to the Earth's inhabitants, who live and thus "breathe" life into the planet, current Web objects never "live". They do exhibit some "living" qualities, but being stateless (see Section 2.2.2.6) they cannot grow on their own, nor can they reproduce themselves. Consequently, Web objects cannot evolve themselves (without human intervention) and direct the evolution of the whole Web in a systematic way, as biological living organisms do to their biological world. The analogy between Web documents and living organisms (introduced in Section 6.1) on the other hand, allows for the design of Web objects capable of these "life" functionalities (grow, reproduce, and self-evolve). This highlights another important discrepancy between the two systems. In nature, the evolutionary process happens in a bottom-up fashion, starting in each individual living entity, whereas on the Web only quantitative growth (change in the number of Web objects) happens at the object level. On the other hand, qualitative (technological) changes are made to the Web as a whole, in a top-down fashion from the system level. Examples of these global changes are changes to the HTML tags set, or the emerging transition from HTML to XML, that affect the whole Web (see Section 2.2.2.2). Changes taking place at the level of the entire Web will probably result in major alterations (not "evolution" in the proper sense), and "quick and dirty" additions to existing features. Changes taking place at the level of individual object will result in gradual development of the whole population, which is the same as the evolutionary process in the biological world.

Biological world	Web
Each individual is uniquely identifiable	Each Web object can be uniquely identified
Individuals populate the earth	Web objects populate the Web space
Individuals are born and die	Web objects are created and destroyed
Individuals can interact with each other and respond to stimuli	Web objects can interact with each other and respond to user inputs
Individuals live in groups or communities where they can communicate with each other through protocols	Web objects belong to sites or communities, are interlinked on a communication network and can communicate with each other through protocols

Table 6-1 : Similarities between the biological world and the Web

Biological world	Web
Individuals are **stateful**	Web objects are **stateless**
Individuals **can** reproduce, grow, and evolve themselves	Web objects **cannot** reproduce, grow, and evolve themselves
Growth, change, and evolution happen at **individual** level	Growth, change and evolution happen at **system** level

Table 6-2: Differences between the biological world and the Web

Table 6-1 and Table 6-2 summarise the similarities and differences between the biological world and the Web, respectively.

If the Web can be regarded as an entity like Earth, then it is possible to populate it with species. The class hierarchy in the object-oriented paradigm can be compared to the species hierarchy in biology. To bring "life" to Web objects, it is necessary for them to change, grow and reproduce themselves. Observations of biological life provide very helpful input for projecting "life" into our model. This section discusses the requirements for a "living" object. It explains how object-orientation and the Web can support this design and where there is still a need for further work. It also points out how evolution and speciation of these objects are possible in the Web environment.

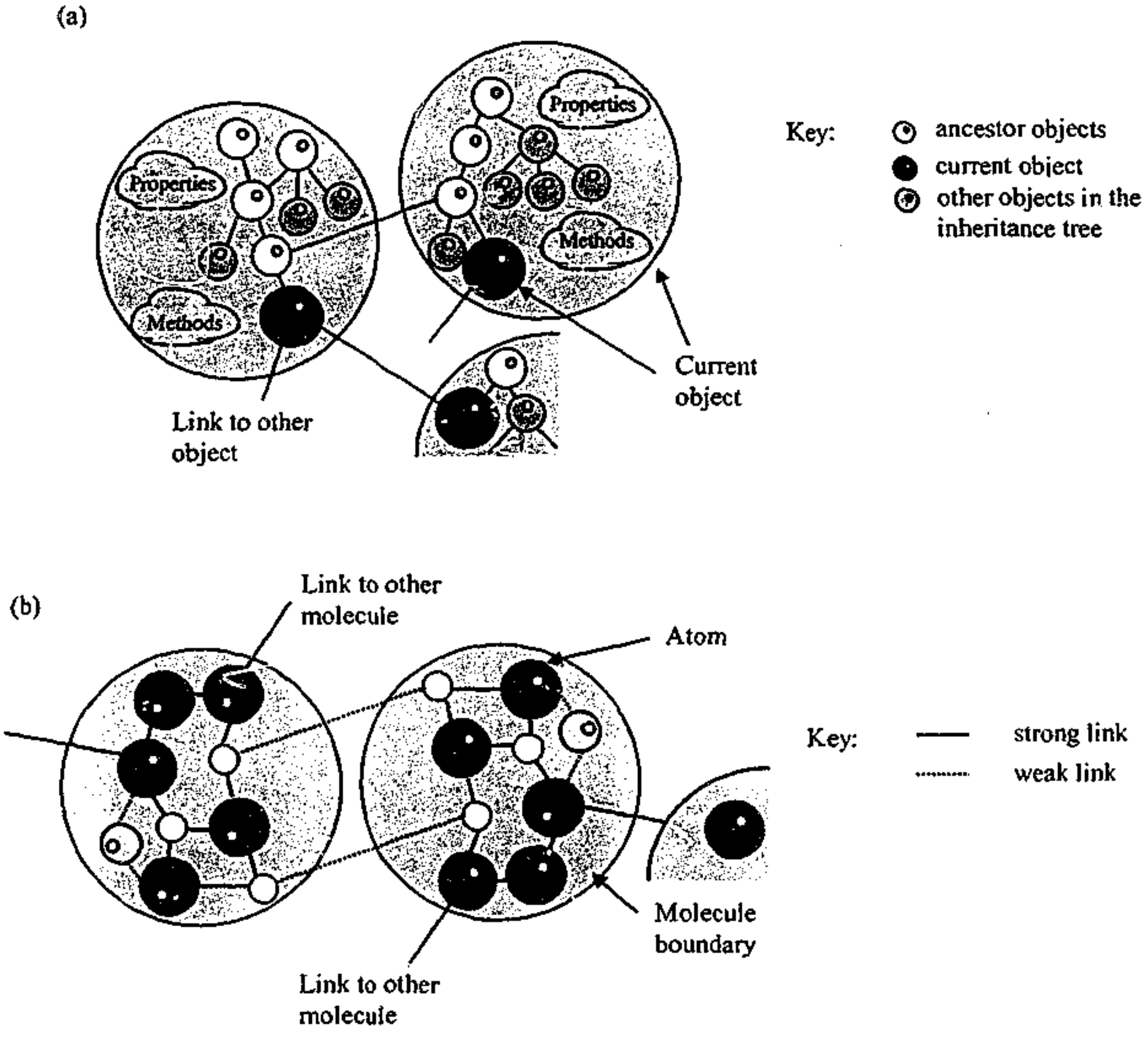


Figure 6-2 : (a) Object structure; (b) Molecular structure

6.3.1 Flexible modular structure

A "living" object should have a flexible modular structure. This requirement is inspired by observing the characteristics of the carbon element (see Section 6.2.5), the chemical reactions among molecules (see Section 6.2.2) and the reproduction process at the genetic level (see Section 6.2.1). It is this structure that allows for an organic compound to change around the flexible connections on a component basis. In the object-oriented world, things are very similar. Objects are analogous to molecules, combined in complex arrangements through relationships such as inheritance, association, and composition. They also possess properties like molecules, and function through a message passing mechanism like that of molecular messages passing in chemical substances, which trigger cascades of chemical reactions. This resemblance has also been noted by Peter Small [SMA96]. In this respect, modularity can be obtained directly from object-orientation (with some special considerations, which will be presented in Section 6.4.2.1).

In nature, link flexibility, or in terms of molecular chemistry, chemical bond stability, is the basis of all life activities. As indicated in Section 6.2.5, it is the weak bonds that allow for the unwinding and reforming of the two strands of the DNA in the reproduction process, and the numerous metabolic reactions that happen in an organism. It was also observed that there are several levels of chemical bond stability, which are dependent on the types of the bonds. For instance, single bonds are stronger than double bonds, which in turn, are stronger than triple bonds.

In object-oriented technology, link flexibility is a concept that has been formalised and implemented with such notions as *final* class in Java [DEI98], or *readonly* attribute in Sather [STO96, SCH91]. Link type, or relationship, however, is still a debatable issue, and does not relate to link flexibility [HEN97]. The most commonly used relationships are *association, aggregation* (or *composition*), and *generalisation* (*specialisation,* or *inheritance*) [RUM91, UML98]. Even these relationships may be defined differently in differing object systems, and their semantics often does not give enough information to assign a degree of flexibility for the links of its type. Association, for example, may itself be considered a generic type of many subtypes, such as *use,* or *define. Use,* as in the case of "Calculator *uses* Memory", and *define,* as in the case of "Genome *defines* Organism", however, would both be categorised as association, but cannot have the same degree of flexibility because of their semantic differences. While it is possible to remove or replace the *Memory* instance from a *Calculator* instance (to make a non-memory calculator or to upgrade memory), it is not possible to remove or replace the *Genome* instance from an *Organism* instance (because the removal would make the *Organism* instance undefined and the replacement would lose its identity).

Formalisation of relationships is not within the scope of this thesis. For the purpose of experimenting with the concepts proposed in this study, some levels of link flexibility will be defined in Section 6.4.2.1.3. Figure 6-2 depicts some similarities and differences between the object's and the molecule's architectures.

6.3.2 Genes

In order for an object to reproduce, it must "know" its own specifications. In order to grow and evolve, it must be able to alter its own specifications. These processes should be carried out during the object's lifetime. This suggests a scheme in which the object carries its own specifications, similar to a protein carrying its own genetic code of life (DNA). This scheme may also be relaxed so that the object may have access to its specifications without actually carrying them. In either way, objects can reproduce, change, grow, and evolve by themselves when certain conditions are satisfied or some events occur. In a similar way, living organisms reproduce, grow, and evolve in response to stimuli (Sections 6.2.1, 6.2.2 and 6.2.3). Besides, each individual living organism has its own specific genes, but all belonging to the same species share the same genome (set of genes). This suggests the notions of *individual gene* that is specific to an individual object, and *group gene* that describe a class of objects.

The genetic requirement entails that a "living" object must be defined by, and has access to, its own individual "gene" and group "gene". Intuitively, an individual "gene" is a special element that is embedded in, or linked to, an object. The individual "gene" contains the (compact) core information specific to the object, which is necessary and sufficient for the creation and reproduction of that object. It controls these (creation and reproduction) and other "life" processes (growth and evolution) that take place in the object. A group "gene" defines the common features of similar objects, including the essential functionalities for the objects to perform "life" activities (see Section 6.3.4), and plays the role of a framework, model, or schema, for the objects it describes.

Under this view we can see the strong parallels between the object-oriented and the biological worlds. An object is an instance of a class in the same way as a creature is an instance of a genome. What needs further work in object-oriented technology is to design individual "genes", to make the object carry or have access to its "genes", and to ensure that these "genes" are modifiable and transferable across objects of the same class (during reproduction). It is also essential to define methods for "life" functionalities (see Section 6.3.4). For transferability, "genes" are desirably represented in the most compact form possible.

6.3.3 Meta-genes

As can be seen in Section 6.2.4, from the genetic point of view, biological speciation happens when individuals from two populations are no longer genetically compatible, that is, the differences between their genes have exceeded a certain limit. We do now know of any scientific study of to what extent such differences will trigger the speciation process, or how to measure these differences. However, it is not impossible to imagine the existence of a *meta-gene* that controls the process. This assumption stands up well when applied to the meta-modelling concepts presented in Section 4.2. Thus, in the same way that a genome defines an individual, a meta-genome defines a species, a meta-meta-genome (M2-genome) a genus, and so on. Speciation would start when a genome has so diverged that it no longer conforms to the specifications defined by the meta-genome. Similar processes would also happen at higher meta-levels.

The recursive abstraction to higher meta-levels of genes forms a pyramid structure in which the higher the level, the more abstract it is, and the fewer distinct groups it contains. The lowest level at the bottom of the pyramid, corresponding to the *concrete level*, is non-abstract (specific) and contains not groups but (concrete) individuals. The highest level at the top of the pyramid, corresponding to the *union level*, is the most abstract, and unifies all groups into one single model. This design, which we call *recursive gene*, is reflected fairly well in the biological classification system, where individuals are grouped into species, species into genera, genera into families, and so on, until there are only a few kingdoms, and finally one single large group of living organisms (see Section 6.2.4).

With regard to the concept of "living" object, assuming a design for "gene" is in place, it would be possible to use meta-modelling technique for the construction of object meta-genes and object genes at higher meta-levels. Recursive gene design also generally agrees with studies of meta-modelling (Section 4.2). Researchers and practitioners in this field generally accept that four levels of instantiation are needed to control the evolution of an information system [KOT84, IRD90, UML98]. Meta-modelling technology, however, does not have the concept of the *union level*, which unifies "the whole world". In this sense, our recursive gene design extends the architectures proposed and used in meta-modelling. In our opinion, in practice, the number of necessary meta-levels need not be restricted to four, but depends on the need to link, unify and cooperate elements across multiple heterogeneous and distributed systems.

6.3.4 Functionalities

As has been pointed out in Section 6.2.2, biological life is characterised by activities such as reproduction, growth, metabolism, and response to stimuli. In addition, a life form must also be born (created), evolve, and die (be destroyed). These activities establish the functional requirements in our design. An object gene must define functions for the "living" object to perform the aforementioned life activities. Here, object-orientation *natively* provides for this requirement by means of (appropriate) methods [RUM91].

We have identified a set of *genetic functionalities* to support the biological life activities aforementioned (Table 6-3). These form the basic functionalities that an object gene must define, and a "living" object must perform.

Genetic functionalities	Life activities
N/A	Metabolism
N/A	Response to stimuli
Self-construction	Creation
Self-duplication	Reproduction
Self-evolution	Evolution
Self-modification	Growth
Self-destruction	Destruction

Table 6-3: Genetic functionalities and life activities

Regarding the life activities listed, we make following notes:

- *Metabolism* and *response to stimuli* are not supported by life functionalities. This is because these activities are already natural behaviour of computer software. *Metabolism,* in the software engineering context, is analogous to the total of data processing processes in an object by which inputs (from users or other objects) are transformed into outputs and for use in other software functionalities (including "life" activities). *Response to stimuli* is the production of output in response to a given input.
- *Reproduction* and *evolution* are meaningful only in an interconnecting space in which "living" objects are created, function, interact with each other, and are destroyed.
- It is also noted that *growth* requires another condition, *statefulness*, as explained in Section 6.3.5 below.

6.3.5 Statefulness

Growth is the process of changing from one state to another that is considered an increase or development. For an object to grow therefore, it is necessary that it be stateful. Object-oriented technology provides a *native* way to define object state by a set of attributes and values, and state information can be held inside an object [RUM91, UML98]. The current Hypertext Transfer Protocol (HTTP) on which the Web is built, however, is a stateless protocol (see Section 2.2.2.6). The question here is of producing persistent objects, that is, how to maintain object state across Web sessions. This is an issue specific to the Web and will be addressed in our *LifeWeb* system (see Chapter 7, in particular, Sections 7.2.3 and 7.2.5.2)

6.3.6 Evolution

The recursive gene design permits the establishment of an exact evolutionary path. Modelled after biological evolution (Section 6.2.3), evolution in a system based on our design can be conceptually described as the process of gradual change over time, from lower to higher meta-levels of genes, on a population of "living" objects, to withstand changes in the environment.

As has been pointed out in Section 6.2.3, three essential elements have been identified for the evolutionary process to happen:

- A gene pool with a variety of genes and a supply of new genes
- A driving force to select the fittest gene or set of genes for a given requirement
- A population of interconnected and self-reproducing individuals in which the favourable genes can multiply and unfavourable ones become extinct.

Applying these conditions to the Web, provided that suitable designs for "living" objects, object genes and object genes at higher meta-levels are in place (that is, they satisfy the requirements outlined from Sections 6.3.1 to 6.3.5 inclusively), it can be seen that the Web is almost ready as an environment for the evolutionary process to happen. Here, using the analogy between the ecological system and the Web mentioned in Section 6.1, the driving forces would be users' and developers' requirements. The communication network on which the Web is built provides for an interconnected space in which "living" objects, that is, Web documents, interact, reproduce, grow and evolve. The population(s) of "living" objects (Web documents), each with its own gene, changeable under the aforementioned driving forces, constitutes the gene pool and is the supply for new genes.

6.3.7 Species and speciation

The formation of a new "species" on the Web would happen as a consequence of the evolutionary process. Patterned after biological speciation (see Section 6.2.4), and using the same analogy between the ecological system and the Web as in the previous section, species and speciation on the Web may be conceptually described as follows:

Species on the Web, or *document species*, is a distinct type of Web document, with characteristic features pertaining to the community or situation from which it is formed, and which remain stable over a long period of time. Instances of a species are interoperable and compatible at the genetic level. In terms of our recursive gene design (see Section 6.3.3), their genes are defined by meta-genes that conform to each other. Instances of differing species, on the contrary, are genetically non-interoperable and incompatible, and their genes are defined by non-conforming meta-genes.

Speciation on the Web is the process of forming a new type of Web document that no longer conforms with the type it derives from. This process would start when some extrinsic factor, such as the formation of a new Web community with new user requirements, causes the formation of a new population of Web objects with slightly different genetic traits. Users' preferences and developers' needs then continue to diverge, leading to a genetic differentiation among the newly formed populations. A new species would form when "living" objects (documents) belonging to differentiated populations are no longer interoperable or compatible at the genetic level. In terms of recursive gene design (see Section 6.3.3), this means the genome of the new species no longer meta-genetically conforms to the original one from which it was derived. (It is noted that interoperability and compatibility between species can still be achieved at higher meta-levels of genes.)

It can be seen that the recursive gene design makes it possible to indicate exactly the level of compatibility and interoperability between "living" objects.

6.3.8 Summary

Biological life	Object-oriented technology
Species hierarchy	Class hierarchy
Species genes	Class specifications
Meta-genes	Meta-model
Molecule	Object
Molecular links	Relationships
Flexible molecular structures	Flexible modular structures
Weak/strong links	Derivable/final classes, or read-write/readonly attributes
Chemical messages	Messages
Cascaded chemical reactions	Cascaded reactions

Table 6-4 : Similarities between object-orientation and biological life

Object-orientation is found to have large potential for designing a system with biologically life-like characteristics. The Web network establishes an interconnected space, necessary for "living" objects to interact, reproduce, change, grow and evolve. Some new concepts still need to be introduced, and adjustments made to object-oriented design. The many parallels between biological life, object-orientation and the Web, however, make it possible for the design of a special object-oriented data model. This model, using biological metaphors, promises to capture the dynamic characteristic of the Web, and address the issues of the Web mentioned in Section 2.2.

Biological life	Object-oriented technology
Individual and species genes	Only class specifications
An organism contains its own genes	An object does not contain its own specifications
Genes in organisms are functional	Class specifications are non-functional

Table 6-5 : Differences between object-orientation and biological life

Table 6-4 and Table 6-5 summarise the similarities and differences between object-oriented technology and biological life respectively.

6.4 Life Design

The observations described in previous sections are used to build a design methodology, called *Life Design*. It is not within the scope of this thesis, however, to build a formal software design methodology. This section presents the core features of *Life Design* based on object-orientation. This design will be used as the guideline to build and evaluate the *LifeWeb* prototype (and like systems), an implementation constructed to experiment with the concepts proposed in this study. All the biological terminology used in this section is within the context of *Life Design*, unless otherwise stated.

Life Design is centred around the elements that constitute natural life at different levels: molecular, genetic and biological, where each level contributes aspects of structural, genetic, and functional, respectively, into our design. These ideas are implemented using object-oriented technology. We use the concepts of object-orientation described in the book *Object-oriented Modeling and Design* by James Rumbaugh et al. [RUM91], and later refined in the *Unified Modeling Language (UML)*

[UML98].[16] Only minimal features in object-orientation are used to describe the core concepts of *Life Design*. This design encompasses two key concepts, *life form*, or *form* for short, and *gene*, which are explained below.

6.4.1 Object-oriented foundation

We define *forms* and *genes* (in the context of *Life Design*) based on the notions of *objects* and *classes* in object-orientation. A *form* includes all characteristics defined for an object, and a *gene* a class.

6.4.1.1 Object and Class

An *object* represents "a concept, abstraction, or thing with crisp boundaries and meaning for the problem at hand" [RUM91]. *(The) CarPGC123*, *(the) EngineVIN345*, *John Smith*, *MyBook*, *Chapter1* (of *MyBook*), are all examples of objects. An object has an implicit *unique identity*. It is always possible to distinguish one object from another.

A *class* describes a set of objects with similar properties (*attributes*), common behaviour (*operations*), common *relationships* to other objects, and common semantics [RUM91, UML98]. In this respect, we say that a class defines the *type* or *schema* of an object, or conversely, an object is an *instance* of its class. For example, the classes for the objects in the above example can be *Car, Engine, Person, Book,* and *StructuralComponent*, respectively. An object "knows" its class.

The notion of class in object-orientation also includes abstract classes, parameterised classes, and interfaces [RUM91, UML98]. For simplicity, *Life Design* is restricted to non-parameterised, and instantiable classes (that is, classes that can be instantiated into objects) only.

Entity is a generic term for either object or class.

6.4.1.1.1 Object lifetime

Object lifetime is a new concept that is not defined in object-orientation.

An object comes into existence when it starts occupying some physical space, such as computer memory or disk space, and acquires its unique identity. An object ceases to exist when this physical space is (considered) permanently released without replacement, or its identity is permanently lost. We define *object lifetime* as the duration of the object's existence.

[16] UML is a cooperative effort between the three leading object-oriented methodologists: Rumbaugh [RUM91], Booch [BOO94], and Jacobson [JAC93], and has become a widely accepted industrial standard.

In most existing object-oriented systems, an object is typically created directly into the computer memory, and its lifetime starts from the time it takes up some memory space and is assigned a unique identity, until the time it is permanently removed from the memory (and thus cannot be retrieved by its identity). In a system such as the Web, an object, for example, a Web document, is realised from a representation (currently its HTML source code) on a disk. Its lifetime starts from the time this representation is instantiated from a document type (in this case, HTML), takes up some disk space, and is assigned a unique identity (currently a URL). Its lifetime ends when it is permanently removed from the Web, or can no longer be retrieved by its identity (for instance, it has migrated to another space and does not leave any path from the old to the new URL).

6.4.1.1.2 Properties

The properties of an object are described by a set of *attributes*. The term *attribute* refers to a named slot within a class that describes a range of values that an instance of that class may hold [UML98]. For example, the attributes for *Book* are *title, ISBN, publicationYear*, and the valid values for them are *string* for *title* and *ISBN*, and *integer* for *publicationYear*. All objects of the same class share the same set of attributes (names), but the values for a given attribute are specific to each object, which may be the same or different for differing objects. Each attribute name is unique within a class. An attribute value should be a pure data value (such as *string* or *integer* as in the above example), not an object. Unlike an object, a pure data value does not have an identity [RUM91].

6.4.1.1.3 Behaviour

An object's behaviour is described in terms of a set of *operations*. An operation is a function or transformation that may be applied to or by an object of a class. For example, *print, display, generateTableOfContents*, are possible operations on objects of the class *Book*. All objects of the same class share the same set of operations. Each operation has a target object as an implicit argument, and may have additional arguments. A *method* is the implementation of an operation for a class. The behaviour of the operation depends on the class of its target. An object "knows" its class, and hence the right implementation of the operation. The same operation may apply to a different class. Such an operation is *polymorphic*, that is, the same operation can take on different forms in different classes [RUM91].

6.4.1.1.4 Relationships

Relationships among objects (or classes) are established by means of *links*. A *link* is a physical or conceptual connection between objects (or classes). For example, "*Chapter1 constructs MyBook*", "*Person owns Car*", "*Book is-a Document*". Mathematically, a link is a tuple, that is, an ordered list of objects (or classes). Most commonly, it is a pair, that is, a two-tuple. *Life Design* deals with binary links (pairs) only. In the example above, the link between *Chapter1* and *MyBook* can be mathematically represented as (*Chapter1, MyBook*).

A *relationship* describes a set of links with common structure and common semantics. It does this by establishing a (new) link between the entities that describe those participating in the (existing) link. The relationship of a link between two objects is a link between their corresponding classes. For example, the relationship of the link between *Chapter1* and *MyBook* is "*StructuralComponent* ***constructs*** *Book.*" In this respect, we say that a relationship defines the *type* or *schema* of a link, or conversely, a link is an *instance* of a relationship.

We introduce the term *relation* as a generic term for either link or relationship.

Association and generalisation relationships

A relationship is itself a link. As mentioned above, the relationship of a link between objects is a link between their corresponding classes. In this respect, a relationship may be further described by another (more abstract) relationship.

The relationships of links between classes are *association* and *generalisation.*

- ❖ *Association* defines a semantic relationship between classes. An association link between classes, in turn, defines an association relationship between objects. In this respect, an association link between classes can be instantiated into an association link between objects. An association link must be defined by an association relationship (unless it cannot be abstracted further, as explained in Section 6.4.1.2). Association (at both class and object levels) can be either of two categories: (i) Ordinary Association, and (ii) Aggregation (or Composition).

    - *Ordinary association* is primarily binary and inherently bi-directional. For example, "*Car* (is) ***Owned-by*** *Person*", or "*Person* ***Owns*** *Car*" are both examples of association links between classes. Examples of the corresponding links between objects are, "(the) *CarPGC123* (is) ***Owned-by*** *John Smith*", and "*John Smith* ***Owns*** (the) *CarPGC123.*" *Life Design* deals only with binary and uni-directional associations, where the single direction usually implies a subordinate-primary relationship, or direction of navigability (having access to). This restriction makes association a transitive (if A is subordinate to B and B is subordinate to C, then A also is subordinate to C), and asymmetric relationship (if A is subordinate B and B is subordinate to A then A is equal B) between two classes. Between objects, it is a transitive and antisymmetric (if A is subordinate to B then B is not subordinate to A) relationship. Association at the object level is thus more restricted than the one at the class level. (It does not allow recursive links, that is, an object cannot associate to itself).

- *Aggregation* (or *composition*) is a tightly coupled form of association, representing the "part-whole", "part-of", or "has-a" relationship. Aggregation is a binary, transitive (if A is part of B and B is part of C, then A is part of C), and asymmetric (if A is part of B and B is part of A then A is equal B) relationship between two classes. Similar to ordinary association, aggregation between objects is also more restricted in that it requires antisymmetry rather than asymmetry. An example of an aggregation link between classes is, "*StructuralComponent* ***constructs*** *Book*", where ***constructs*** implies that *StructuralComponent* is a ***part-of*** *Book*. An example of a corresponding link between objects is, "*Chapter1* ***constructs*** *MyBook*." In UML [UML98], aggregation and composition are two different kinds of association. *Composite aggregate* (composition) has stronger "ownership" than *shared aggregate* (aggregation). For example, an entity that is part-of a shared aggregate may participate in another aggregate, whereas one that is part-of a composite aggregate may not. This distinction is not significant in *Life Design*, and the terms *aggregation* and *composition* may be used interchangeably in this thesis.

Since aggregation is a special case of (binary, uni-directional) association, it is possible to consider the part-whole relationship as a specific type of the subordinate-primary one. A mixture of both relationships therefore, is also transitive and asymmetric (in the sense of a subordinate-primary relationship). They are two disjoint sets, which either individually or together, define a *Directed Acyclic Graph (DAG)* (at the object level) or a *DAG with self-loops* (at the class level). "Self-loops" here means an entity may have a link to itself.

The notion of association defined by Rumbaugh [RUM91] and UML [UML98] also includes *qualified association, derived association, association class, constraint,* and *multiplicity*. For simplicity *Life Design* does not consider qualified association, derived association, and association class. Multiplicity and constraints specify semantic rules to be observed, and they are treated in *Life Design* in the same way as prescribed by object-orientation. For simplicity, we generally do not discuss these rules here, except in some illustrations that relate to them.

❖ *Generalisation* (or *specialisation*, or *inheritance*) defines a taxonomic relationship between classes only [UML98]. This means, a generalisation link between classes cannot be instantiated to links between objects. Generalisation is an evolutionary mechanism for classes (see Sections 6.4.2.1, 6.4.2.4.4, and Chapter 8). It is assumed to be always defined for classes, and need not be defined further by another relationship. It is a binary, transitive and antisymmetric relationship between a more general class and a more specific class. The general class is called *superclass*, and the specific one *subclass*. For example, a possible superclass of *Book* is *Publication*. Generalisation represents the "is-a" relationship, that is, an instance of a subclass is simultaneously an instance of its superclass. A subclass inherits all the features (attributes, operations, and association and aggregation links) of its superclass, and may contain additional features. In this respect a subclass is fully consistent, or *conformed*, with its superclass. A subclass can inherit from more than one superclass, in which case the relationship between them is also called *multiple inheritance*. The terms *ancestor* and *descendent* are used to refer to generalisation of classes across multiple levels. The generalisation relationship defines a DAG [RUM91, UML98], which may also be referred to as the *inheritance tree*.

In *Life Design*, we consider that the subclass-superclass (generalisation) relationship implies the subordinate-primary (association) one, and restrict the directions of a mixture of their links so that they conform to the transitivity and asymmetry of association (and aggregation). That is, if A is a subclass of B, then B must not be subordinate to A. (It is acceptable if B is primary to A). Thus a mixture of specialisation, association, and aggregation relationships among classes, can also be considered transitive and asymmetric (in the sense of a subordinate-primary relationship). They may overlap, and either individually or together, define a DAG (specialisation, or association and aggregation at the object level), or a DAG with self-loops (association and aggregation at the class level).

Since antisymmetry is more restricted than asymmetry (the former logically implies the latter), and this restriction is not important in *Life Design*, we shall use asymmetry as the generic condition for all the generalisation, association, and aggregation relationships at both object and class levels, and a DAG with self-loops as the generic structure defined by these relationships.

The definitions of links and relationships here are different from those described by Rumbaugh [RUM91] and UML [UML98]. In both Rumbaugh's and UML's methodologies, a link is defined only between objects, not classes; and a relationship is only a (abstract) term of convenience without any specific semantics. Consequently, association and aggregation links are defined between objects only (not classes). These differences arise because of a more fundamental difference between *Life Design* and traditional object-oriented design: the recursive meta-model architecture (which has been informally described in Section 6.3.3).

6.4.1.2 Recursive meta-model architecture

The recursive meta-model architecture in *Life Design* consists of recursive levels of entities and relations (that is, objects – links, and classes – relationships). In this way, as a relationship can also be a link and described further by another relationship, a class can similarly be an object, and described further by another class (in the same way as it describes its objects). In this architecture, in any two adjacent levels, entities and relations in the higher level define classes and relationships for entities and relations in the lower level. Equivalently, entities and relations in the lower level are objects and links to entities and relations in the higher level. We call the higher level the *schema level* (relative to the lower level), and the lower level the *instance level* (relative to the higher level). Figure 6-5 shows an example of such architecture across four meta-modelling levels. In this figure, the entities (represented as labelled boxes) and relations (represented as labelled arrows) in diagram (a) are described by those in (b), which in turn, are described by those in (c), which in turn again, are described by those in (d).

There can be many distinct sets of entities and relations at each level. A given set of entities and relations at the schema level may have zero or more corresponding sets of entities and relations at the instance level. A set of entities and relations at the instance level may have at most one corresponding set at the schema level. We call the set at the schema level the *system schema* of the set at the instance level; or equivalently, the set at the instance level a *system instance* of the set at the schema level. The terms *schema* and *instance* may also be used for short, if there is no danger of confusion. If an instance refers to the schema of its schema, the term *meta-schema* may be used. If references to higher meta-level schemata are required, the prefix *meta*, or *M<n>*, where *<n>* is the number of the occurrences of the word *meta*, may be added for each successive higher meta-level. For instance, the word *M2-schema* is equivalent to the word *meta-meta-schema.*

We postulate that there is a lowest level in the architecture in which entities and relations cannot be instantiated further, that is, they form a schema whose instance is an empty set. This level is called the *concrete level*. There is also a highest level in the architecture in which entities and relations cannot be abstracted further, that is, they form an instance whose schema is an empty set. This level is called the *union level*. Graphically, this architecture forms a pyramid structure of schemata, in which the higher the level, the more abstract they are, and the fewer distinct instances they define. Schemata at the concrete level are concrete instances. The single schema at the *union level* is the most abstract, and unifies all instances into itself.

A system schema defines most classes and relationships for a system instance of it, but not all. Such a system schema is called the *main system schema*, or *main schema* for short, relative to its system instance. A system instance may have an entity (and its respective relations) defined in another system schema. Such an entity is called a *hybrid object*, its class a *hybrid class*, and the corresponding schema a *hybrid system schema*, or *hybrid schema* for short. A hybrid class must be able to form links with classes in the main system schema. For instance, a hybrid class may be a class derived from an existing class in the main system schema, but is not included in the main system schema. According to our postulation about the union level, hybrid objects are not allowed at the level immediately below the union level (because there is only one schema at the union level).

UML defines a similar architecture, called the *Four-Layer Metamodel Architecture* [UML98 pp.2-4 – 2-5], resembling the ISO/IEC IRDS architecture [IRD90] described in Section 4.2. However, there are differences between the UML meta-model architecture and that of *Life Design*.

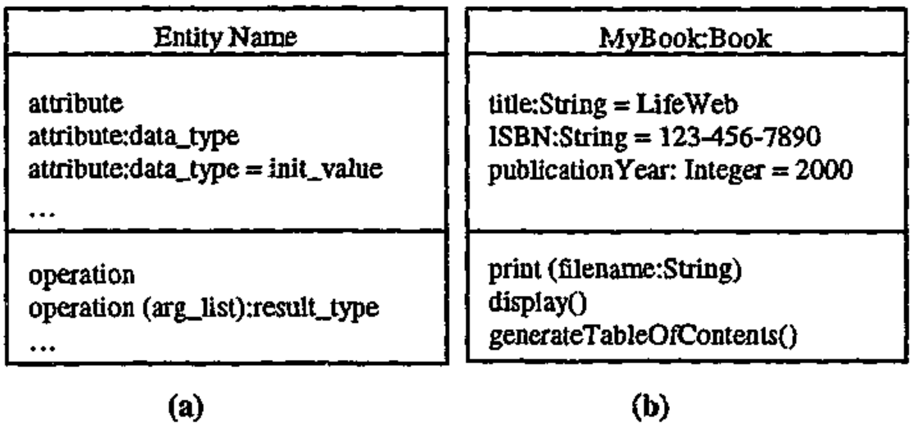
- First, UML does not have the concept of hybridisation. This concept permits entities to systematically and incrementally evolve themselves, by allowing for a new class to be introduced into the main system schema. This is done first through one single object in one of its (the main system schema's) system instances. Through the *Life Design* evolutionary mechanism, the new class may multiply in other system instances, and if passing the "survival of the fittest" test, will be propagated to a higher meta-level (schema). System evolution is explained in Sections 6.4.2.1.3, 6.4.2.4.4 and Chapter 8. In contrast, UML meta-model architecture can offer extensibility, but not evolvability.
- Secondly, *Life Design*'s architecture is more generic. It defines two generic levels (schema and instance) and four generic constructs, two for each level (class and object; relationship and link). This makes it possible to apply *Life Design* concepts on any two adjacent levels recursively and uniformly. UML, on the other hand, defines levels and constructs specific to its problem. For instance, the four specific levels in UML are: user objects, model, metamodel, meta-metamodel. Some of the specific constructs in its metamodel layer are: Class, Attribute, Operation; and in its meta-metamodel layer: MetaClass, MetaAttribute, MetaOperation. As such, the generic architecture defined in *Life Design* may be used as a reference model to describe specific architectures such as the one defined in UML. This explains the differences between *Life Design* and UML in the definitions of links and relationships. In *Life Design*, they are generic terms across meta-levels, whereas in UML, they are specific to a particular level.[17]

6.4.1.3 Notation

We use the notation defined for UML Class Diagrams and Object Diagrams [UML98] with some modifications.

[17] Actually, *relationship* in UML is only a term for convenience, and does not belong to any level.

6.4.1.3.1 UML notation


Entity Name
attribute
attribute:data_type
attribute:data_type = init_value
...
operation
operation (arg_list):result_type
...
MyBook:Book
title:String = LifeWeb
ISBN:String = 123-456-7890
publicationYear: Integer = 2000
print (filename:String)
display()
generateTableOfContents()

(a) (b)

Figure 6-3: UML Class and Object diagrams – (a) General, (b) Object

In UML, an entity is represented as a box with three compartments. The first compartment shows the identifier of the entity. The second compartment contains the set of attributes (optionally with data types for attribute values and initial values). The third compartment holds the set of operations (optionally with argument lists and result types) defined on the entity. In the first compartment, if the entity is a class, the class name is shown. Otherwise, if the entity is an object, its name can be optionally shown in the format of <object name>:<class name>. Either or both of the second (attribute) and third (operation) compartments may be suppressed. Figures 6-3 (a) and (b) show the general UML notation of an entity and an object, respectively, with all three compartments.

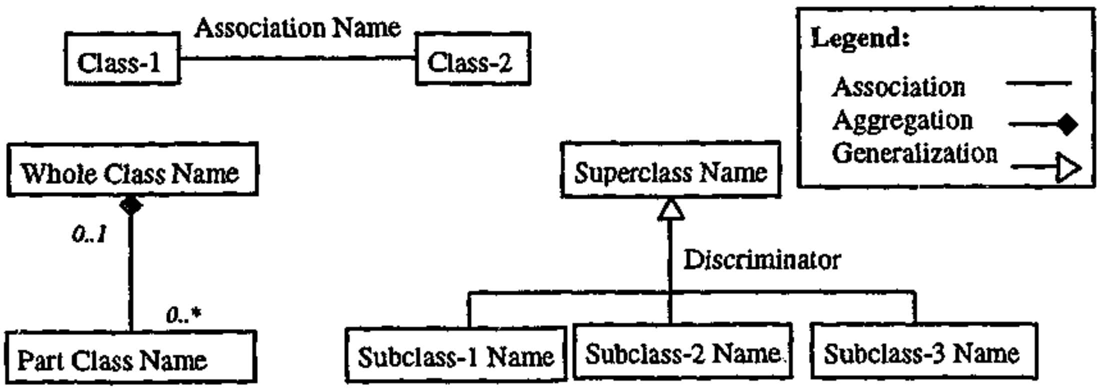


Figure 6-4: UML Class diagram (with relationships)

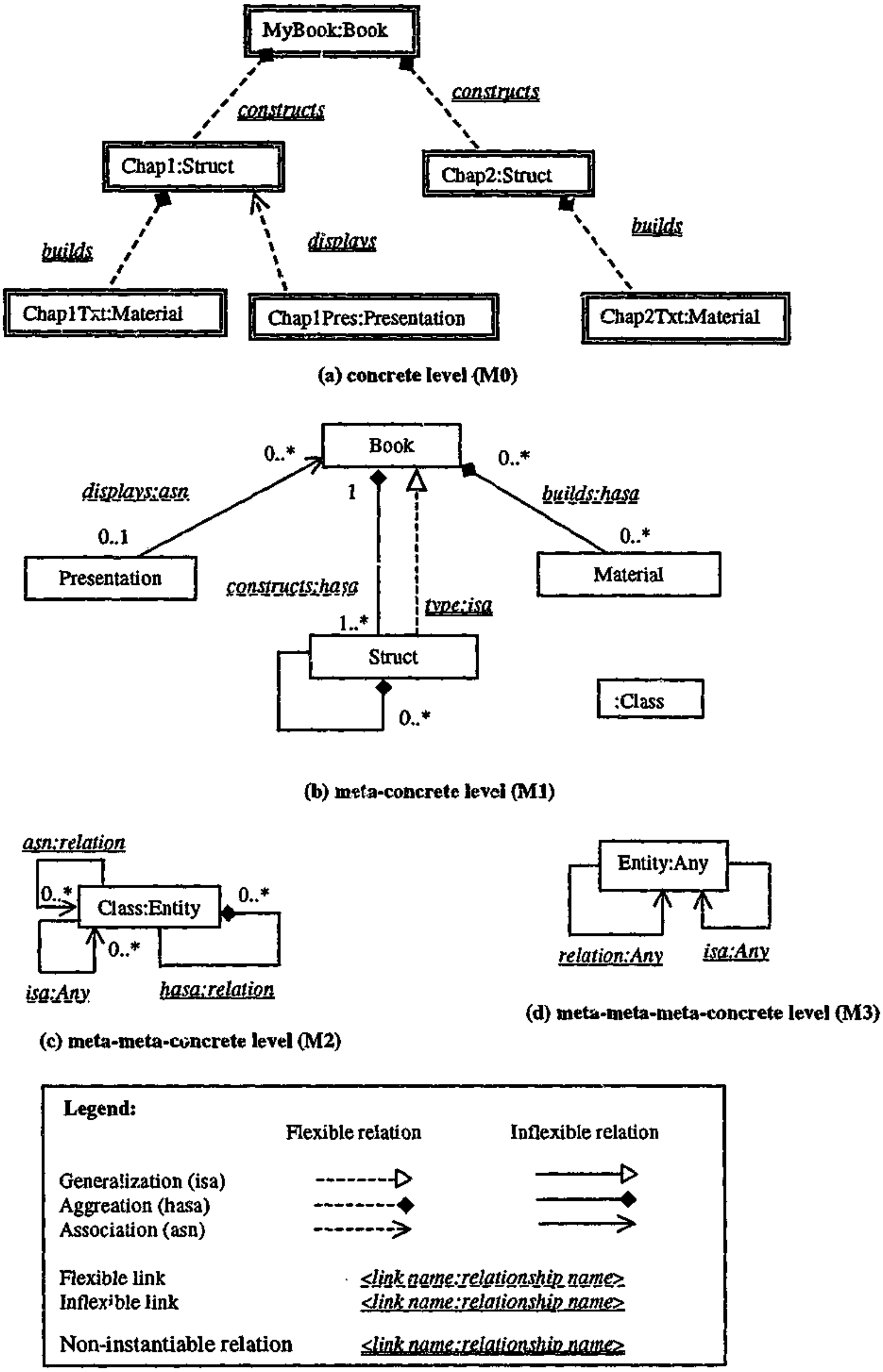
A relation is shown as a line (association), or an arrow of a distinct shape (aggregation and generalisation) (see Figure 6-4). Relations are optionally labelled. When shown, a label corresponds to an association name (for an association), or a discriminator (for a generalisation), which is the name of a partition of subclasses. Aggregations are not named. (See Figure 6-4). An association may also have a role name at each association end (not shown in Figure 6-4). Role names are treated in *Life Design* as they are in object-orientation, and are not discussed here.

6.4.1.3.2 Life-UML notation

Unlike UML, in *Life Design* both entities and relations always have labels. A label is in the format <instance>:<type>, where <instance> is the name of the object or link, and <type> is the name of the class or relationship, respectively. (For example, the entity labels in Figure 6-5 (a), (c), and (d), or the relation labels in Figure 6-5 (b), (c), and (d)). If the names for both <instance> and <type> are the same, then only one of them may be shown (for example, the relation labels in Figure 6-5 (a)). If all objects have the same class, then the entity labels may be abbreviated to <instance>, and the common class name is shown in a separate box in the format of :<type> (Figure 6-5 (b)). If there is no type defined for an entity or a relation, the default type, "Any", is used (for example, the entity and relation labels in Figure 6-5 (d)). In a given class diagram (with relationships), an entity label is naturally unique, but a relation label may be repeated.

A relation in *Life Design* is always shown as a labelled, uni-directional arrow. Since a link label includes both the link name and relationship name, it is not necessary to use arrow shapes to distinguish among different relationships. By convention, however, the different arrow shapes are still used as they are defined in UML, except for association, which is represented in *Life Design* as a plain arrow (instead of a line as in UML), as shown in Figure 6-5. The directions of aggregation and specialisation links are the same as those in UML. The direction of an association link is from subordinate (tail) to primary (head) (Figure 6-5). A link can be either flexible, which is indicated as a dashed arrow, or inflexible, which is indicated as a solid arrow (see the Legend in Figure 6-5). The flexibility of a link is dependent on its semantic, and described in its relationship. A dashed line under a relationship label (link label between classes) indicates that the corresponding link (between corresponding objects) is flexible. Otherwise, a solid underline on a relationship label indicates an inflexible link (Figure 6-5 (a), (b), (c), (d)).

Life Design incorporates recursive levels of meta-modelling (see Section 6.4.1.2). For this purpose, a box with a double-lined border indicates that an entity cannot be instantiated further, and similarly, a double line under a relation label means the same for a relation (Figure 6-5 (a)). On the other hand, the default type "Any" indicates that an entity or link cannot be abstracted further.



(a) concrete level (M0)

(b) meta-concrete level (M1)

(c) meta-meta-concrete level (M2)

(d) meta-meta-meta-concrete level (M3)

Figure 6-5: Examples of Life-UML diagrams across four levels of meta-modelling

6.4.2 Life Design constraints

This section defines *form* and *gene*, the two key concepts in *Life Design*.

A *form* possesses all the characteristics of an object. Either an object or a class (considered as an object relative to the next higher meta-level) can be a form if it satisfies the following four constraints: structural, genetic, stateful and functional.

6.4.2.1 Structural constraint

The structural constraint is modelled after the molecular structure. It requires that a form should be recursively decomposed into a *flexible modular structure* of forms, connecting with each other via *links*, unless it is *atomic*.

6.4.2.1.1 Flexible modular structure and links

A form's structure is constructed by a set of (component) forms and their links. The relationship of the links between forms (in such a structure) is called *subform. Subform* is a transitive and asymmetric binary relationship between two forms. Subform relationship defines a Directed Acyclic Graph (DAG) with self-loops.

If a form *l* is in a subform relationship with *m*, we call *l* a *subform* of *m*, or equivalently *m* a *superform* of *l*. For instance, in the example in Figure 6-5 (a), *Chap1:Struct* is a subform of *MyBook:Book*, and conversely, *MyBook:Book* is a superform of *Chap1:Struct*. Mathematically, a subform link is represented as a pair, for example, (*m, l*), where *m* is the superform and *l* is the subform. It is noted that the names of the relationship and the thing (in this case, form) participating in that relationship are the same (subform). Usually the context clears any ambiguity. If precision is required, the phrase *subform relationship* is used to explicitly indicate the relationship.

Subform is a purely structural relationship and is realised by the association, aggregation, or specialisation relationships (described in Section 6.4.1.1.4). This realisation is possible since subform has the same generic structure as any of those relationships, or any combination of them. If a subform relationship is realised as an association or aggregation, it is called a *first-order subform (relationship)*. Otherwise, if it is realised as a generalisation, it is called a *second-order subform (relationship)*. Subform (or superform) from now on will be used as a generic term to denote either a first-order or second-order subform (or superform).

We also call a subform that is immediately below a form a (first or second-order) *direct subform* (of the given form). That is, if *A* is a first-order (or second-order) direct subform of *B*, then there is no intermediate first-order (or second-order) subform between *A* and *B*. A form *has access* to its direct subforms, for instance, by embedding or containing references to them.

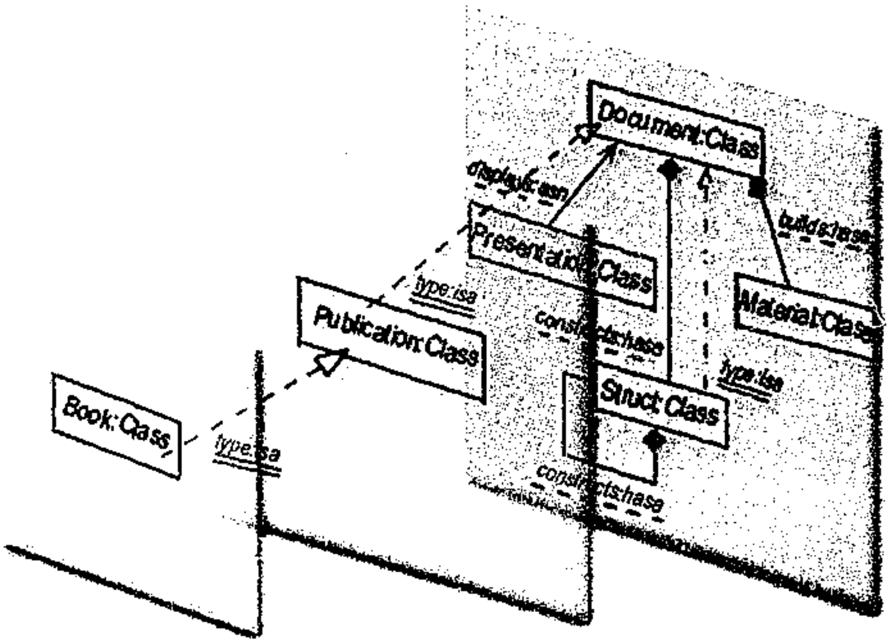


Figure 6-6: First-order and second-order subform relationships (without multiplicity)

In this way, at a particular meta-modelling level, the set of first-order subform links defines the relationships for the corresponding set of first-order subform links at the lower level, if any. The set of second-order subform links, if any, forms an inheritance tree and defines a mechanism to evolve the entities and links at the current level (see Sections 6.4.2.1.3, 6.4.2.4.4 and Chapter 8 for evolution). Graphically, a second-order subform may be seen as being elevated on another plane (Figure 6-6).

Since these two sets (first and second-order subform links) may overlap, we define that if a link belongs to both of them, it is considered as belonging to the first set only. This rule is established because in this case, a generalisation link may be simply considered as a "short-cut" way to include the inherited association or aggregation links in the form at the subclass end. In this sense, the taxonomic semantics of the generalisation link becomes unimportant. For instance, in Figure 6-6 the generalisation link (*type:isa*) between (*Document:Class*, *Struct:Class*) may be replaced by two links *displays:asn* and *builds:hasa* between (*Presentation:Class, Struct:Class*) and (*Material:Class, Struct:Class*), respectively.

The set of (first, second, or all-order) subform links emanating from a form constructs the set of (first, second, or all-order) subform links in that form. For example, in Figure 6-6, the set of first-order subform links in *Document:Class* is {(*Document:Class, Presentation:Class*), (*Document:Class, Struct:Class*), (*Document:Class, Material:Class*), (*Struct:Class, Struct:Class*), (*Struct:Class, Presentation:Class*), (*Struct:Class, Material:Class*)}. The set of its second-order subform links is {(*Document:Class, Publication:Class*), (*Publication:Class, Book:Class*)}.

A form is said to have the *capacity to form a (first, second or all-order) subform link* (or the capacity to be a first, second, or all-order subform of some form) if there is a corresponding (first, second, or all-order) subform relationship defined for it in the higher meta-level, and semantically this capacity can be used. A form does not have this capacity when at least one of these conditions is false. For example, in Figure 6-5 (a), *Chap1:Struct* has the capacity to form a first-order subform link since there is a corresponding first-order subform link for its class, that is, (*Book:Class, Struct:Class*). (The link between *Struct:Class* and itself also qualifies for this). *MyBook:Book*, on the other hand, does not have this capacity, since there is no link defined for *Book:Class* in which it is a first-order subform. Also, *Book:Class* does not have this capacity, although there are two first-order subform links between *Class:Entity* and itself (Figure 6-5 (c)). This is because semantically, *Book:Class* cannot be a first-order subform of any form.

6.4.2.1.2 Atomic form, root form, living organism, and free form

Given a set of forms, all connecting with each other via the subform relationship:

We postulate that there exists in the set at least one form which either does not have any subform, or has only itself as a subform. In the first case, the form is called *atomic*. In the second case, it is called *recursive atomic*. For example, in Figure 6-5 (a), *Chap1Txt:Material, Chap1Pres:Presentation*, and *Chap2Txt:Material* are atomic. In Figure 6-5 (b), *Presentation:Class*, and *Material:Class* are atomic, and *Struct:Class* is *recursive atomic.*

There must exist in the set at most one form which is neither a first nor a second-order subform of any other form (except itself), and does not have the capacity to be a first-order subform of any other form (except itself). Such a form is called the *root form.* There also exists in the set at least one form which is not a first-order subform of any other form (except itself), and does not have the capacity to be a first-order subform of any form (except itself). Such a form is called a *subroot form.* (This implies that if there is only one subroot form in a given set, it is the root form). For example, in Figure 6-5 (a), *MyBook:Book* is the root form (and also the only subroot form). In Figure 6-6, *Document:Class* is the root form, and *Publication:Class* and *Book:Class* are both subroot forms.

The set itself is called a *living organism*, or *organism* for short. An organism contains, and therefore, has access to all forms in its set.

A form that exists outside an organism is a *free form*. This implies that a free form has the capacity to be a subform of some form, and none of that capacity is used. A free form may be a form disconnected from its organism, or created elsewhere.

The condition that a root or subroot form does not have the capacity to be a first-order subform of any other form ensures that an organism always represents the *whole* of a system described at the higher meta-level. For instance, in the example shown in Figure 6-5, it is not possible to take any subset of the set of forms shown in the diagram in (a) to construct an organism.

6.4.2.1.3 Link flexibility

Link flexibility is an attribute of a subform link. There can possibly be several degrees of link flexibility. For simplicity, we specify only two of them: *flexible* and *inflexible*. A flexible link can be removed or added during the lifetime of the participating forms (possibly under certain conditions or constraints), and an inflexible link remains throughout their lifetime. The 'emoval and adding of flexible links are carried out via the *genetic functions*, which are explained in Section 6.4.2.4.3. There must exist at least one flexible link in an organism.

The value of link flexibility is dependent on the semantics of the link. We require that a first-order subform link may be flexible only at the concrete level. A second-order subform link is always flexible with the constraint that the addition of a second-order subform link also adds a new form (to a given organism), and the removal of such a link also removes an existing form (from a given organism).[18] In object-oriented terms, this means that a class cannot change its superclass(es),[19] but the class inheritance tree can be extended or shrunk. In this way, generalisation or second-order subform relationship defines a system evolutionary mechanism (see Section 6.4.2.4.4 and Chapter 8).

We call the set of all flexible subform links emanating from a form the set of flexible subform links in that form. The set of inflexible subform links in a form is the set of its subform links subtracted by the set of its flexible subform links.

[18] This constraint implies that if the form being removed has second-order subform links with other forms (as in the case of multiple inheritance), these links too, have to be removed.

[19] This is because due to the semantic of the generalisation relationship, removing or adding a superclass alters the internal definition of the subclass. The internal definition of a class here is understood in the entirety of the class, which consists of its attributes, methods, including inherited ones, and its "subordinate" (in the generic sense explained in Section 6.4.1.1.4) classes.

6.4.2.2 Genetic constraint

The genetic constraint is modelled after the same constraint in biological life forms. It requires that a form should *have an individual gene and a group gene*, and *have access to its own genes and main genomes* (see Figure 6-7). (It is noted that if the form is not a hybrid, having access to its main genomes also provides access to its own genes).

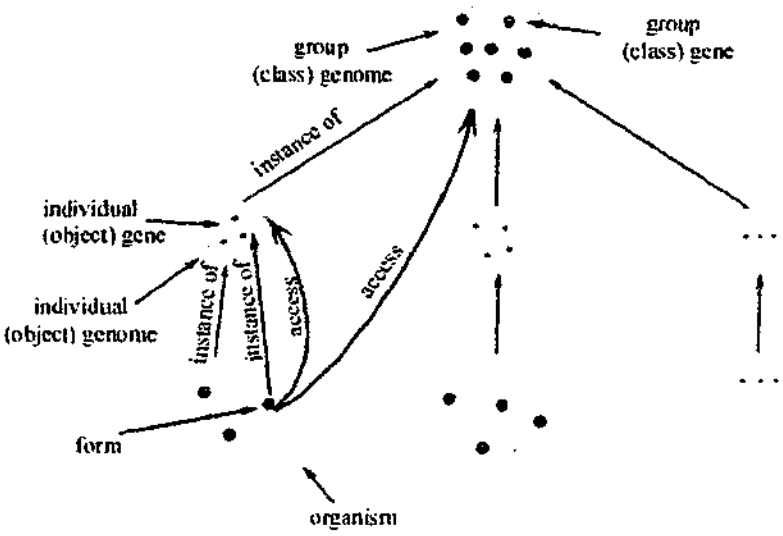


Figure 6-7: Genetic constraint (connections between forms or genes not shown)

Intuitively, an individual gene is a representation of the (specific) form it describes. For this purpose, we introduce the notion of *object representation*, which is also applicable to objects, as below.

6.4.2.2.1 Object representation

Object representation is a new concept in *Life Design*.

An object may have one or several representations. For example, the object *MyBook* may be represented as *MyBook(Title: LifeWeb, ISBN: 123-456-7890, PublicationYear: 2000, Contents: Chapter1, Chapter2)*, or as an electronic image in the computer memory, or both. We call these *object representations*, or *representations* for short (if there is no danger of confusion). The object is presented to the world in one and only one of these representations. We call it the *final object representation*, or *final representation* for short (if there is no danger of confusion). The process of creating a specific representation, with a unique identity, for an object from its class is called *instantiation*. The process of mapping from a representation to the final representation is called *realisation*. (These two processes will be defined in more details in Section 6.4.2.4.1). If an object has only one representation (thus it is final), then its representation need not be realised.

The choice of the final representation is dependent on a given problem. For example, in a system analysis problem, the final representation for an object may be in some notation such as in the first example above. In a system implementation (computer application), on the other hand, it may be an electronic image in the computer memory as in the second example.

A notable object representation is the *syntactic object representation* or *syntactic representation* for short (if there is no danger of confusion). A class describes its objects in a machine-understandable language. We call such a description for a specific object a *syntactic representation.* The first example above, in which *MyBook* is represented as *MyBook(Title: LifeWeb, ISBN: 123-456-7890, PublicationYear: 2000, Contents: Chapter1, Chapter2)*, illustrates a syntactic representation for *MyBook.* In this example, the class *Book* describes its objects as *Book(Title, ISBN, PublicationYear, Contents).* A syntactic representation is static, textual, declarative, and typically more compact than other representations.

We also call the representation of an object in the computer memory a *functional object representation*, or *functional representation* for short (if there is no danger of confusion). In most existing object-oriented systems, an object is instantiated from its class directly into the computer memory. In this case, the functional representation is its only representation, and the syntactic representation is not required. (To be more precise, the syntactic description of the object is interspersed with that of its class and program code. For this reason, it is not a whole and is not considered a syntactic representation). In a system such as the Web, an object, for example, a Web document, has its (currently) HTML source code as its syntactic representation.

6.4.2.2.2 Gene

A *gene* describes its form(s) in terms of its (or their) features (properties, behaviour, and direct subform relationships), including inherited ones. (Implementation-wise, second-order superforms may, and should, be included in a gene instead of an explicit listing of all inherited features.) A gene must exist during the whole lifetime of the form(s) it describes. (This means, a form exists implies that its gene exist.) A gene must satisfy all constraints applied on a form, that is, *a gene itself is a form.* It is desirably and typically more compact than the form(s) it describes. There are two types of genes:

- *Object gene* is a description of an individual form (in terms of its features). It contains the information *necessary and sufficient* to *realise* the form it describes.
- *Class gene* is a description of a set of similar forms (in terms of their common features). It contains the information *necessary* to *instantiate* the forms it describes. This information can be used to create the object gene for a particular form, with the supply of attribute values and

subforms specific to the form being instantiated. It can also be used to validate a given form, for example, to check the form's type (see Section 6.4.2.2.3 for validation rules).

Gene is a generic term for either object gene or class gene.

In this respect, an object representation of a form (see Section 6.4.2.2.1) can be the object gene of that form, and its class its class gene, if they (i) exist during the whole lifetime of the form, and (ii) satisfy all the constraints on form.

The first condition implies that the choice of the gene is dependent on the (predetermined) lifetime of the form. For example, if a form is an object in a computer application that lasts only during the execution time of that application, then its object gene and class gene can be the functional representations of itself and its class, respectively. If a form is a Web document that lasts from the time its syntactic representation (HTML source code) is created and assigned a URL until the time this representation is removed, then its object gene and class gene can be the *syntactic* representations of itself and its class, respectively.

The second condition means that a gene must satisfy the following constraints:

Structural constraint

A gene has the same structure that a form has (see Section 6.4.2.1). Given a set of genes, all connecting via the subform relationship, the equivalent of an atomic form and recursive atomic form (in a similar set of forms) are called *atomic gene* and *recursive atomic gene*, respectively. An atomic gene always generates an atomic form. The equivalent of root form and subroot form are called *root gene* and *subroot gene*, respectively. Only a subroot gene can generate a root form. The equivalent of organism is *genome*. These terms are applicable to both object genes and class genes.

Genetic constraint

In the same way as a form, a gene must have its own object gene and class gene, which are accessible to it. An exception to this is a gene that cannot be abstracted further, for example, one at the union level. Such a gene is called an *ultimate gene*.

A class or an object representation can satisfy this genetic requirement in the recursive meta-modelling architecture described in Section 6.4.1.2:

- A class is also an object (in relation to its own class at the next higher meta-level), which can have its own object representation that can be its object gene. Its class gene can be its own class (meta-class). That means, a class in the *Life Design* recursive meta-modelling architecture can satisfy the genetic requirement to be a class gene.

- An object representation has the same class as the object it describes. That means, its class can be its class gene. It does not need its own representation (being a representation itself), but in order to satisfy the genetic constraint, we can consider it the representation of itself, that is, it is the object gene of itself. Thus, an object representation in the *Life Design* recursive meta-modelling architecture can also satisfy the genetic requirement to be an object gene.

The genetic constraint constructs a recursive gene architecture like the recursive meta-modelling architecture described in Section 6.4.1.2. In this architecture, in any two consecutive levels the instance level consists of forms and links (represented by their object genes); and the schema level, class genes and subform relationships (represented by their class genes). The term *genetic level* (and hence, meta-genetic, M2-genetic, and so on) may also be used interchangeably with the term *schema level*. The equivalents of the terms *hybrid object, hybrid class, hybrid schema*, and *main schema* are *hybrid form, hybrid gene, hybrid genome*, and *main genome* respectively.

6.4.2.2.3 Validation rules

The recursive meta-modelling architecture in *Life Design* allows the establishment of a set of rules to conveniently validate an organism against its main genome and hybrid genomes. These rules can easily be checked in Life-UML diagrams.

Rule 1 (Unique gene). Given a set of organisms and a set of genomes. For each entity (form) at the instance level, there must exist one and only one corresponding entity (gene) at the schema level.

Rule 2 (Unique entity). Each entity, represented in a Life-UML diagram by its box label, in an organism or a genome is unique.

Rule 3 (Relationship between gene and form).

Rule 3.1. Given two forms *A* and *B* of an organism *O*. *A* is a first-order direct subform of *B* in one of association or aggregation if and only if the gene of *A* is also a first-order direct subform of the gene of *B* in the respective relationship. This relationship (between the genes) may be inherited.

Rule 3.2. If a gene is atomic then its form is also atomic. (It is noted that the reverse is not true. An atomic form may have a non-atomic gene which has an optional subform.)

Rule 3.3. A form is a root form if and only if its gene is a subroot gene.

Stateful and **behavioural constraints** are exactly the same for genes as they are for forms, which are described below.

6.4.2.3 Stateful constraint

The stateful constraint requires that a form should have a persistent state that can change.

In *Life Design*, the state of a form consists of its *internal state* and *relational state*. The internal state is defined by the set of attribute names and values of the form. The relational state is defined by the set of both first and second-order subform links of the form.

The function to change a form's state is called *self-modification*, which is defined by its class gene (see Section 6.4.2.4).

It is noted that, since an object gene is an exact image of a form, the state of a form is equivalent to that of its object gene. Furthermore, since an object gene must also be stateful (whereas other object representations of a given form may not be stateful), the stateful requirement on a form is practically applied on its object gene, the "persistent representation" of the form. That means, when considering the statefulness of a form or an object gene, it is sufficient (and practical) to consider it on the object gene and the object gene only.

6.4.2.4 Behavioural constraint

A form must support the following functionalities: self-construction, self-duplication, self-modification, self-evolution, and self-destruction (Section 6.3.4). These form the set of *genetic functionalities*.

6.4.2.4.1 Self-construction

Self-construction consists of two operations for creating a form from its class and object genes.

Instantiation

Instantiation is an operation that, from a class gene, creates an object gene for a specific form, with a unique identity, and specific features (attribute values and direct subforms), which are given as parameters, or generated internally by the operation itself, or set by default.

Instantiation implementing the first approach takes as parameters, attribute values, and references to the direct subforms of the object gene being created, and assumes that these subforms are already created. Instantiation implementing the second approach takes attribute values internally computed or directly (hard) coded in itself, and recursively generates the object gene for each direct subform of the current form. Instantiation implementing the third approach takes, from the class gene, default attribute values and default references to the object genes of its direct subforms (which may also be *null*). A particular implementation of the instantiation operation may combine all three approaches in different ways.

A definition for the instantiation operation that implements the first approach is:

> *Instantiation* is an operation defined on a class gene g belonging to a class genome G, that takes as arguments a set of attribute values and a set of object gene references. It generates a new object gene g_l (for a form l) based on g, whose properties are assigned to the given attribute values, and direct subforms are assigned to those referenced by the given object gene references.

In existing object-oriented applications, a *constructor* is an approximate implementation of the instantiation operation. The only difference between a constructor and an instantiation method is that the constructor does not create an object gene, but the final representation, for the object being instantiated.

Realisation

> *Realisation* is an operation defined on an object gene g_l, that maps g_l to the final representation of l.

In most existing object-oriented applications, the functional representation is the only and final representation of an object, thus realisation is not necessary. In systems where a form has more than one representation, it may be instantiated once but realised many times during its lifetime. For example, the HTML (or XML) source code of a Web document (object gene) may be authored (instantiated) only once, but the document itself (form) may be displayed (realised) many times on a client's machine.

6.4.2.4.2 Self-duplication

Duplication of a form l is an operation that produces a new form l' identical to l, but different in identity. If the form has more than one representation, this is done via the form's object gene. A definition of the duplication operation for a form that has an object gene is:

> *Duplication* is an operation defined on a form l in an organism O, that retrieves its object gene g_l, and produces an object gene $g_{l'}$ identical to g_l but different in identity. It does this by recursively duplicating g_l and each of its direct subform. It then realises a form l' from $g_{l'}$ (which is identical to l but different in identity). Duplication may take, as parameters, some space references which the new form and object gene will occupy.

In most existing object-oriented applications, this operation is called *deep clone*. If the form to be duplicated has only a functional representation, the cloning process happens entirely in the computer memory. If the form has, or is, an object gene, the object gene is also duplicated.

6.4.2.4.3 Self-modification

Self-modification consists of operations for adding or removing subform links in a form (changing its relational state), and for changing its internal state. If the form subject to modification is at a level higher than the concrete level (that is, it is also a class gene), such addition or removal may have a chained impact on related forms (on the same level), or instances that it defines (on the lower level). A class gene is said to be free of dependency if this is not the case. That is, if it can be safely added or removed without affecting the internal structure or behaviour of related class genes and forms. We introduce the concept of *dependency clearance*, applicable to classes (hence, class genes) below.

Dependency clearance

Dependency clearance ensures data and schema consistency when a system (for example, a genome and its corresponding organism) is changed (for example, as a result of system evolution).

Schema dependency

A class is *free of schema dependency* if and only if:

- It is a leaf node in the class inheritance hierarchy, that is, no class is derived from it, and
- It cannot participate in an association or aggregation link with any class in which it is a "subordinate" (in the generic sense explained in Section 6.4.1.1.4). In implementation terms, no objects of other classes should hold a variable of its type. It is possible, however, for it to participate in an association or aggregation link as a "primary" (in the generic sense explained in Section 6.4.1.1.4). It is also possible for it to have a recursive link to itself.

Clearance of schema dependency is necessary to ensure schema consistency while the system evolves.

Instance dependency

A class is *free of instance dependency* if and only if its instances cannot be found in any instance of any class in the entire system. Clearance of instance dependency is necessary to ensure data consistency while the system evolves.

Dependency clearance

A class is said to be *free of dependency* if and only if it is free of both schema and instance dependency.

AddLink

AddLink is an operation that establishes a link between two forms. If the form has more than one representation and the new link is to be made permanent, a corresponding link is also established between the object genes of the two forms. A definition for AddLink is:

> *AddLink* is an operation defined on a form *l* belonging to an organism *O*, that takes as argument a form *m*, and adds a new pair (*m*, *l*) (or (*l*, *m*), depending on whether *l* or *m* is the subform) to the set of flexible subform links in *m* (or *l*). *m* can be a free form or a form within O. If the change is permanent, the object gene of *l* is also retrieved (from *l*) and the same change is made on it.

The following conditions must be satisfied:

- *l* and *m* have the capacity to form a flexible subform link with each other
- Such formation of a subform link between *l* and *m* does not violate any rules and constraints defined in the association, aggregation or generalisation link that is used to realise the subform link (see example below).
- Such formation of a subform link between *l* and *m* does not violate any *Life Design* constraints. For instance, in the example shown in Figure 6-5 (a) and (b), suppose that there is a free form *RumbaughBook:Book*. It is not possible to create a subform link between this form and *Chap2Txt:Material*, for instance, even though forms of types *Book* and *Material* have the capacity to construct a flexible subform link, and such a construction does not violate any rules defined at the schema level. This is because forming such a link would violate the structural constraint in *Life Design* that there is only one root form in an organism (see Section 6.4.2.1.2).

RemoveLink

RemoveLink is an operation that removes an existing link between two forms. If the form has more than one representation and the change is to be made permanent, the corresponding link is also removed between the object genes of the two forms. A definition for RemoveLink is:

> *RemoveLink* is an operation defined on a form *l* belonging to an organism *O*, that takes as an argument a form *m* belonging to *O*, and subtracts the pair (*m*, *l*) (or (*l*, *m*), depending on whether *l* or *m* is the subform), if exists, from the set of flexible

> subform links in *m* (or *l*). If the change is permanent, the object gene of *i* is also retrieved (from *l*) and the same change is made on it.

The following conditions must be satisfied:

- The pair (*m*, *l*) (or (*l*, *m*)) are in a flexible subform relationship
- Such a removal of the subform link between *l* and *m* does not violate any rules and constraints defined in the association, aggregation or generalisation link that is used to realise the subform link. For instance, in the example in Figure 6-5 (a) and (b), it is not possible to disconnect both *Chap1:Struct* and *Chap2:Struct*, because a form of type *Book* must always contain at least one form of type *Struct*.
- Such a removal of a subform link between *l* and *m* does not violate any *Life Design* constraints (see example above).
- The form to be removed is free of all dependency.

The removal of a subform link may exclude a form (with its subforms) from its organism, if that form does not have any other subform links. Such a form may be considered permanently destroyed (cease to exist), or become a free form, depending on the rules specified in its gene. For instance, in the example shown in Figure 6-5 (a) and (b), if *Chap1:Struct* is disconnected from *MyBook:Book*, it is destroyed, because according to the multiplicity constraint between *Struct:Class* and *Book:Class*, a form of type *Struct* cannot exist outside an aggregate of a form of type *Book* or *Struct*. If a disconnected form is destroyed, its direct subforms also become disconnected, and may be destroyed or become free forms in the same way. If a disconnected form becomes a free form, it retains all its subform structure.

A free form may be connected or reconnected with a form in the organism via a flexible subform link, using the AddLink operation.

StateChange

StateChange is an operation that modifies the attribute value of a given attribute in a given form. A definition for the StateChange operation is:

> *StateChange* is an operation defined on a form *l* that takes as parameters an attribute name and an attribute value, and assign one (value) to the other (name) in the given form. If the change is permanent, the object gene of *l* is also retrieved (from *l*) and the same change is made on it.

It is also possible to have a set of StateChange operations defined for each attribute name of a given form, that takes as parameter an attribute value and assigns it to the attribute name being operated on.

In existing object-oriented systems, methods that implement the StateChange operation are called *mutators*. It is also convenient to define the corresponding *accessors*, which are methods that retrieve attribute values for given attribute names.

6.4.2.4.4 Self-evolution

Self-evolution is an operation defined on a form that belongs to an organism in a set of organisms of the same class genome. It *conditionally* propagates an *evolutionary change* from the form to its class gene, that is, from lower to higher meta-levels.

An *evolutionary change* is an irreversible (that is, the change happens both at the form and its object gene) addition or removal of a subform link in a form belonging to a given organism. This kind of change may result in an irreversible addition or removal of a subform in the given form (see Section 6.4.2.4.3), which in turn, may propagate to the higher meta-level and result in an irreversible addition or removal of the corresponding class gene in the class genome of the given organism. Thus evolution in *Life Design* happens incrementally in a bottom-up fashion, starting at individual forms, and propagating up to higher meta-levels.

Since only generalisation links are flexible at a schema level (see Section 6.4.2.1.3), adding or removing a subform link at this (and any schema) level can happen only in the set of second-order subform links. Addition of a second-order subform link is prohibited between forms in the same organism, and removal of such a link is prohibited if the participating forms remain in the same organism afterwards. Under these constraints, such operations (addition and removal of second-order subform links) mean extending and truncating, respectively, the class inheritance tree (via the generalisation relationship). In this respect, the generalisation, or second-order subform relationship, defines a mechanism to evolve the system at the schema level. Schema evolution happens only on a *component* (class gene) *basis* through deriving new class genes from existing ones, or truncating existing ones.

The propagation of an evolutionary change may happen on two *conditions*:

- The subform added is the first one of its type in the given organism and its type does not exist in the organism's main class genome, that is, it is a *hybrid subform* (see Sections 6.4.1.2 and 6.4.2.2). Alternatively, the subform removed is the last one of its type in the given organism.
- The propagation is controlled by a factor named *evolutionary threshold*. When the evolutionary threshold is reached, the change will be carried to the next higher meta-level, resulting in a "more

genetic" change. The evolutionary threshold is expressed in terms of the *fitness of gene*, which is measured by the rate of occurrence (or absence) of a gene in a set of individual organisms defined under the same class genome.

The evolution operation may be defined as follows:

> Given a set *L* of organisms having the same class genome, the *evolution* operation is defined on a form *l* that belongs to an organism *O* of the set *L*. It inspects evolutionary changes in the organisms in *L*, and detects whether the rate of occurrence (or absence) of a particular class gene *g* has reached the evolutionary threshold applicable for *g*. If this is true, gene *g* will be considered for irreversible addition (or removal), via appropriate subform links, in the class genome (using the respective AddLink or RemoveLink operations).

System evolution will be explained in detail through the example of our *LifeWeb* system in Chapter 8.

6.4.2.4.5 Self-destruction

Destruction is an operation that completely removes a form and its object gene from the universe.

> *Destruction* is an operation defined on a form *l* that simultaneously releases the space occupied by *l* and its object gene, and completely erases its unique identity.

For example, a Web document is destroyed when it is deleted from the file system, or when it is migrated to a new file system and loses its identity (URL).[20]

[20] This is true only if URL is considered the unique identity of a Web document.

Chapter 7 The LifeWeb model

As has been stated in Chapter 5, our system is designed in two distinct stages: the Multimedia Document Model (MDM) and the *LifeWeb* model. *LifeWeb* is the MDM augmented with features specific to the Web environment (electronic, distributed, and hyperlinked). These features include other media types (for example, sound and video), hyperlinks, behaviour, schema, and external services. *LifeWeb* is an implementation of *Life Design*. This chapter describes *LifeWeb* and explains its features in the light of *Life Design*. All the biological terminology used in the rest of this thesis is within the context of *Life Design*, unless otherwise stated.

7.1 LifeWeb classes

Figure 7-1 and Figure 7-2 show the design of the core model (top level), and (part of) the second level respectively, using Life-UML notation (Section 6.4.1.3). The second level is an extension of the core model, for the system is expected to be extended as it evolves. *LifeWeb* is based on the Multimedia Document Model (MDM) described in Chapter 5, amounted with features specific to the Web environment. These features are captured in the classes: *LifeWebObject, Behaviour, Schema, Hyperlink, Evolutor, Service*, and classes of new media types (*Audio, Video, Script*).

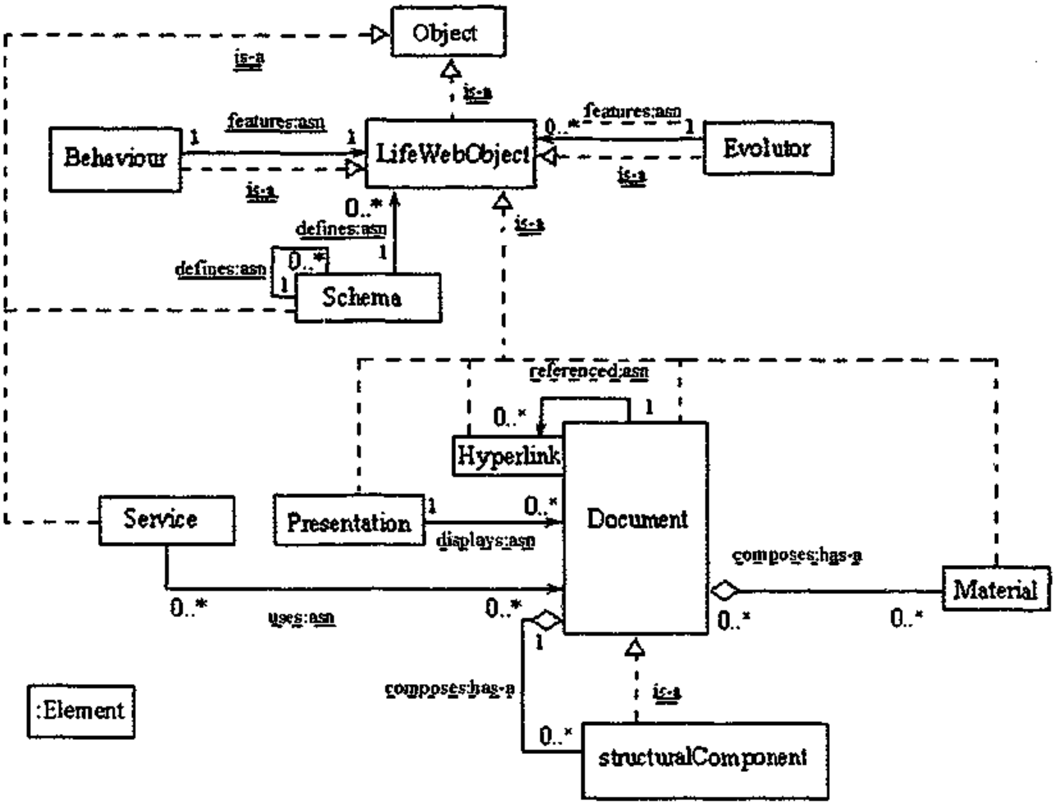


Figure 7-1: LifeWeb object diagram - Core model

LifeWebObject

All classes, except for *Service* and *Schema* (see below), are derived from the *LifeWebObject* class, either directly or indirectly. This class captures the common properties and methods of *LifeWeb* objects. *LifeWebObject* is assumed to be derived from a class, *Object*, which is the common superclass for all classes.

Behaviour

Behaviour represents the Application Programming Interface (API) implementation of a given *LifeWeb* class. (It is through this API that *LifeWeb* objects exhibit their "behaviour".) A *LifeWeb* object is linked to its behaviour in a 1:1 (one-to-one) inflexible association relationship (*features:asn*), whi provides the *LifeWeb* object with access to its class implementation. In traditional object-oriented systems, where functional representation is the only object representation (see Section 6.4.2.2.1), an (runtime) object "knows" its (runtime) class and does not have access to the (static, or design time) class implementation. Having explicit access to the class implementation is not a requirement in *Life Design*. However, a *LifeWeb* object also has a syntactic representation (see Section 6.4.2.2.1), which may be (physically and/or syntactically) different from, or the same as, the class implementation. If they are different, having access to the class implementation allows dynamic linking of *LifeWeb* classes, thus making it possible to automate the evolutionary process (see

Section 9.4.3). In any cases, having references to, rather than embedding, the class implementation makes the syntactic representation of the object more compact.

Behaviour is a feature essential to satisfy the behavioural requirement in *Life Design* (Section 6.4.2.4), and to address the problem of lack of functional behaviour of the Web (Sections 2.2.2.3).

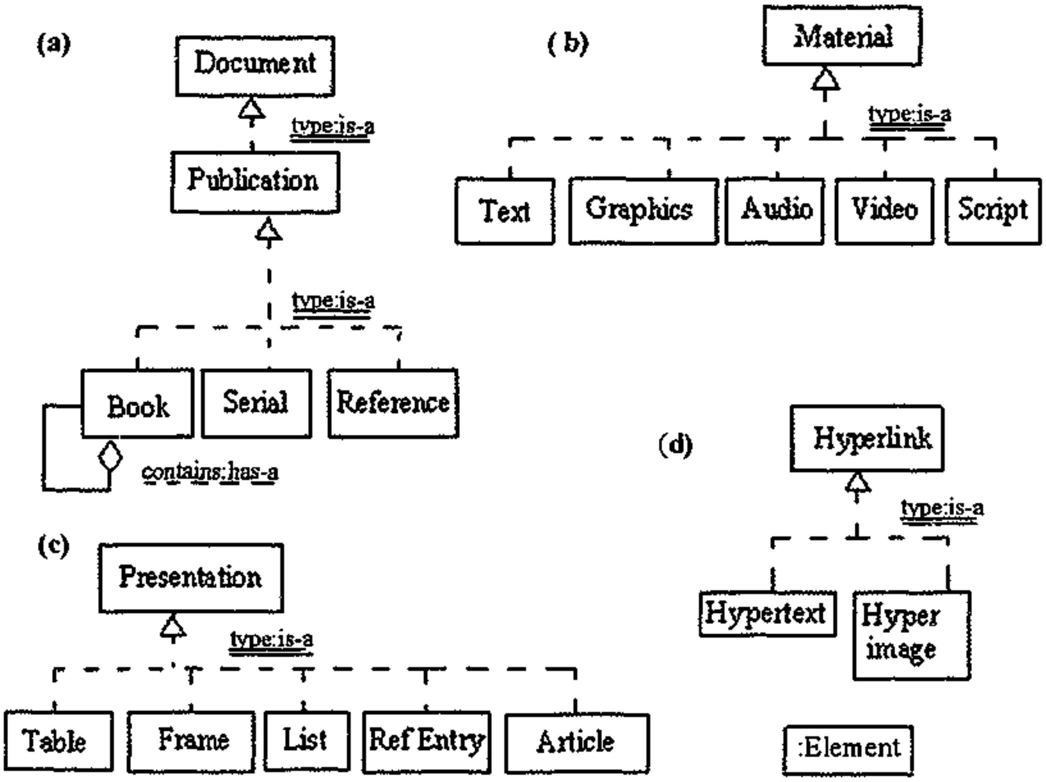


Figure 7-2 : Second level - Extending some LifeWeb classes

Evolutor

Evolutor is a class that captures the evolutionary engine which *LifeWeb* objects use to evolve themselves. An *Evolutor* class is an implementation of the self-evolution operation (described in Section 6.4.2.4.4). This operation could have been implemented simply as a method (like other genetic functions), but its complexity justifies a separate encapsulation. Furthermore, such encapsulation allows for slightly different evolutionary models to be supported. For instance, different implementations may use different ways to calculate the fitness value of a gene (see Sections 6.4.2.4.4, 8.1 and 8.5). An *Evolutor* object is associated with a *LifeWeb* object through an inflexible 1:M (one-to-many) association link (*features:asn*). (This means, a *LifeWeb* object may not change its evolutor during its lifetime).[21]

Schema

Schema is really a meta-class that captures class genes of *LifeWeb* objects (see Section 6.4.2.2). A *Schema* object represents a *LifeWeb* class gene that is linked with a *LifeWeb* object through an inflexible 1:M association link (*defines:asn*), giving the *LifeWeb* object access to its own class gene. (This means, a *LifeWeb* object may not change its schema during its lifetime). *Schema* has a recursive link to itself, that is, a *Schema* object is associated with its own (meta) schema in the same way a *LifeWeb* object is associated with its *Schema* object. This constructs the recursive meta-modelling architecture described in Section 6.4.1.2.

Hyperlink

A *Document* object and its *Hyperlink* qualifier together define a single reference to a *Document* object, which is a flexible association relationship (*referenced:asn*). A document can be referenced by zero, one or more documents. *Hyperlink* is a class encapsulating the source and an anchor (text or image) from which the link emanates, a reference to which the link terminates, and a scope to specify how wide and how deep the link is defined. *Hyperlink* is subclassed into *Hypertext* and *Hyperimage* (see Figure 7-2). Section 9.5 explains in more details about *Hyperlink* and the maintenance of referential integrity in *LifeWeb*.

Service

Service is an abstract class to be inherited by external services possibly provided by third-party software developers. As shown in Figure 7-1, a *Service* object is connected to a *Document* object via a M:M (many-to-many) flexible association link (*uses:asn*). This provides the gateway to extend the system functionalities with various services acquired from external resources. The use of *Service* is explained in detail in Sections 8.2.4, 9.4.2 and 9.4.3.

Audio, Video, Script

Audio, Video, and *Script* are classes that capture the different media types that a *Document* object may contain. They are included in the model for completeness purposes, but do not belong to the core model of *LifeWeb*, and will not be designed or implemented in the *LifeWeb* prototype.

[21] This decision is subjective. Other implementations may allow the link to be flexible.

Other classes in *LifeWeb* that are not explained here have been described in the Multimedia Document Model (see Chapter 5).

7.2 LifeWeb and Life Design

As has been stated before in the introduction of this chapter, *LifeWeb* is an implementation of *Life Design*. This section explains how *LifeWeb* supports each constraint in *Life Design* at two adjacent levels, object and class (in object-oriented terms), or concrete and genetic (in *Life Design* terms). It also gives comments on the significance of this design in the context of the Web. Figure 7-3 shows an example of a *LifeWeb Document* object, using Life-UML notation (see Section 6.4.1.3).

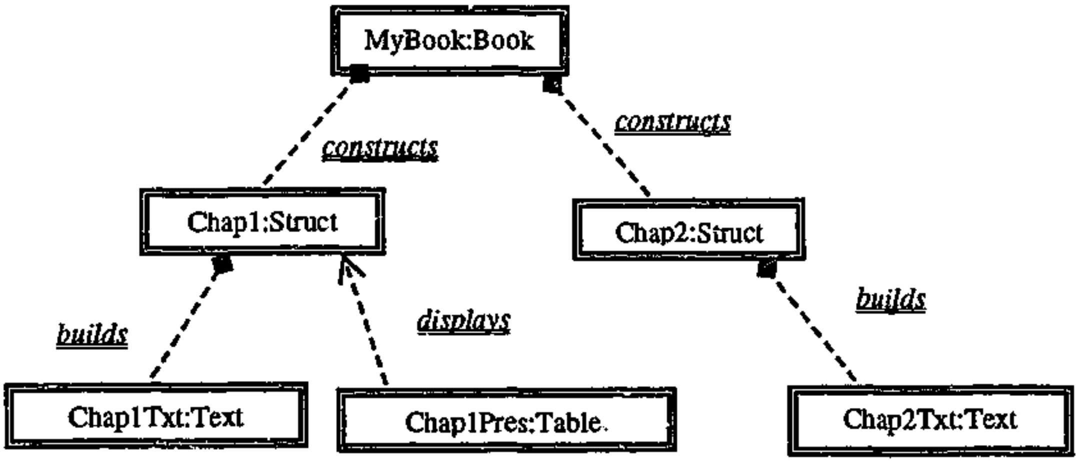


Figure 7-3: A LifeWeb *Document* object

7.2.1 Structural constraint

The structural constraint can be considered with regard to the following conditions (applicable on both levels): (i) the ability to create subform links; (ii) the existence of at least one atomic form and at most one root form; and (iii) the existence of at least one flexible link.

Ability to create subform links

The ability to create subform links is inherently included in the object-oriented design of the system. At both object and class levels, a first-order subform link is realised as an association or aggregation link. At the class level, a second-order subform link is realised as a generalisation link. Figure 7-4 depicts the decomposition of a part of the *LifeWeb* class system in first and second-order subform links (link labels are not shown). In this structure, the first-order subform links are contained in one plane, and the second-order ones connect one plane to another.

Existence of at least one atomic (or recursive atomic) form and at most one root form

- Atomic form: the existence of an atomic (or recursive atomic) form is inherently included in the subform structure (DAG with self-loops) of the system. For example, at the object level, the document component hierarchy (captured by the MDM as described in Chapter 5) is a DAG in which the materials associated with document components are the terminals, as depicted in Figure 7-3. In other words, objects of the *Material* branch (*Text, Graphics, Audio, Video,* and *Script*) have no components (subforms) of themselves, and are atomic. At the class level, these classes (*Text, Graphics, Audio, Video,* and *Script*), by virtue of their semantics, are also atomic (Figure 7-2 (b)).

- Root form: at the object level, a *Document* object (or an object of any class in the *Document* branch, except for *StructuralComponent*), being a subroot gene at the class level (Figure 7-1 and Figure 7-2 (a)), is also the root form at the object level (Figure 7-3). It represents *the whole* of a document, where all other document components (structure, presentation, material, hyperlink, service, behaviour, schema and evolutor) are either contained in it (through aggregation), or associated with it (through association). At the class level, *Object*, the common superclass of all classes, is the root form of the system.

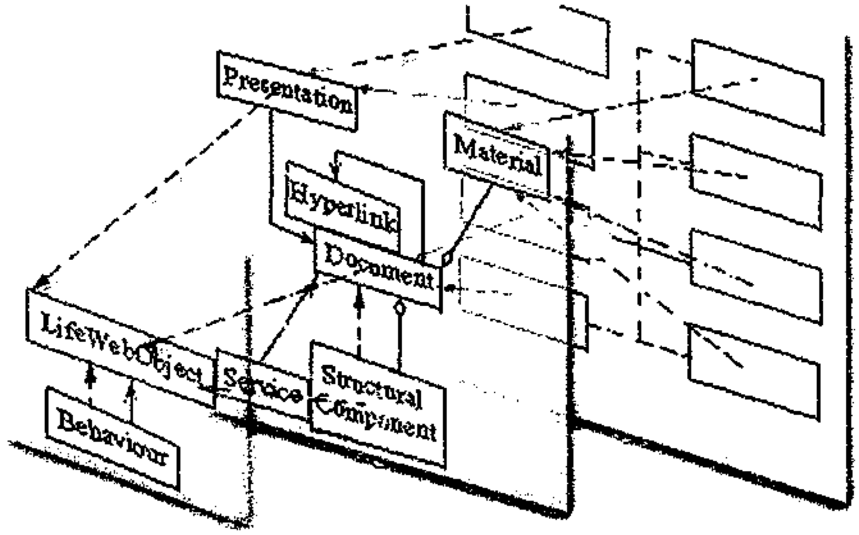


Figure 7-4: First and second-order subform links in LifeWeb class system

Existence of at least one flexible link

A number of flexible links have been defined in our system at the object level (see Section 7.1). For instance, links between *Service* and *Document* objects are flexible. At the class level, all generalisation links are flexible, as has been defined in *Life Design* (see Section 6.4.2.1.3).

7.2.2 Genetic constraint

The genetic constraint can be considered with regard to the following aspects: (i) the existence of an object gene and a class gene (for a given *LifeWeb* object), which are accessible to the object they describe; (ii) the satisfaction of constraints on a life form; and (iii) the lifetime of the genes.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE LifeWeb System="http://lifeweb.org/lifeweb.dtd">
<LifeWeb id="http://lifeweb.org/lifewebbook.xml"
behaviour="LifeWeb">

        <table id="Chap1Pres" row="1" col="2" behaviour="LWTable"/>
        <book id="MyBook" heading="LifeWeb and Life Design"
        behaviour="LWBook">
                <struct id="Chap1" heading="INTRODUCTION"
                present="Chap1Pres" behaviour="LWStruct">
                        <text id="Chapter1Txt"
                        src ="../mybook/data.htm#intro"
                        behaviour="LWText"/>
                </struct>
                <struct id="Chap2" heading="LIFE DESIGN"
                behaviour="LWStruct>
                        <text id="Chap2Txt"
                        src="../mybook/data.htm#lifedesign"
                        behaviour="LWText"/>
                </struct>
        </book>
</LifeWeb>
```

Figure 7-5: The object genome of a LifeWeb document (simplified)

Existence of an object gene and a class gene

A *LifeWeb* object has the syntactic representation of itself as its object gene and its class its class gene. For instance, the object gene of a *LifeWeb Document* object can be its XML source code, and its class gene the *LifeWeb* class system (Figure 7-1). Figure 7-5 shows exemplary object genes, using XML syntax, for the objects depicted in Figure 7-3.

In Figure 7-5, the object genes for *MyBook:Book, Chap1:Struct, Chap2:Struct, Chap1Txt:Text, Chap1Pres:Table* and *Chap2Txt:Text* are the XML elements with identifications (shown in the attribute *id*) matching the respective object names (*MyBook, Chap1, Chap2, Chap1Txt, Chap1Pres,* and *Chap2Txt*). In this representation, an attribute is shown as a pair <attribute name>"="<attribute value>, for instance, *heading="INTRODUCTION"*. The object behaviour, that is, its class API implementation, is linked to the object in a read-only attribute, for instance, *behaviour="LWBook"*. An aggregation link (for instance, between *MyBook* and *Chap1*) is represented by nesting the elements in one another (*Chap1* is nested in *MyBook*). An association link (for instance, between

Chap1 and *Chap1Pres*) is represented by including the reference of one element (*Chap1Pres*) in another (*Chap1*).

In this example, the object genes are contained in an XML file, which is the object genome. An object gene (and hence the object it describes) has access to its object genome via the value of the *LifeWeb* document's *id* (`http://lifeweb.org/lifewebbook.xml`), and to itself via a combination of the document's *id* and its own *id* (for example, `http://lifeweb.org/lifewebbook.xml#Chap1`). It has access to its class genome by the value of the <!DOCTYPE> declaration (`http://lifeweb.org/lifeweb.dtd`), and to its own class gene by a combination of this value and its own element type name (for instance, `http://lifeweb.org/lifeweb.dtd#struct`).

Life form constraints

These constraints include: structural, genetic, stateful and behavioural.

Structural

As can be seen from the above example, such an object genome (as represented in XML) makes it possible to represent and preserve the structure of the entire system it describes, in this case, a *LifeWeb* document (although with limited typing information).[22] Thus, if the system has been designed to satisfy the structural constraint (as is the case of a *LifeWeb* document, explained in Section 7.2.1), its object genome will also satisfy the same constraint.

The *LifeWeb* class genome, that is, the *LifeWeb* class system, also satisfies the structural constraint as already explained in Section 7.2.1.

Genetic

The genetic constraint is applied recursively on genes, that is, a class gene should have its own object gene and class gene.

If the object gene of a *LifeWeb* object is an XML element (as described above), then the object gene of a *LifeWeb* class is the corresponding element type definition in the *LifeWeb* Document Type Definition (DTD). Figure 7-6 shows exemplary object genes of some *LifeWeb* classes, using XML

[22] One exception is the *Schema* object, which is not represented in XML in an association relationship with any elements (as designed in the *LifeWeb* data model), but as a global, independent entity in the <!DOCTYPE> declaration. This however, is only a design choice of XML, and does not invalidate the claim that XML makes it possible to represent and preserve the entire structure of an object-oriented system, with limited typing information.

syntax. (The element types in this example correspond to the elements used in the example shown in Figure 7-5.)

```
<!ELEMENT LifeWeb (book|struct|text|table)*>
<!ATTLIST LifeWeb
      id           CDATA        #RFQUIRED
      behaviour    CDATA        #FIXED  "LifeWeb">
<!ELEMENT book (book|struct|text)*>
<!ATTLIST book
      id           ID           #REQUIRED
      heading      CDATA        #IMPLIED
      behaviour    CDATA        #FIXED  "LWBook">
<!ELEMENT struct (struct|text)*>
<!ATTLIST struct
      superclass   CDATA        #FIXED  "book"
      present      IDREFS       #IMPLIED
      behaviour    CDATA        #FIXED  "LWStruct">
<!ELEMENT text EMPTY>
<!ATTLIST text
      id           ID           #REQUIRED
      src          CDATA        #IMPLIED
      behaviour    CDATA        #FIXED  "LWText">
<!ELEMENT table EMPTY>
<!ATTLIST table
      id           ID           #REQUIRED
      row          CDATA        #IMPLIED
      col          CDATA        #IMPLIED
      behaviour    CDATA        #FIXED  "LWTable">
```

Figure 7-6: The object genome of the LifeWeb class system (simplified)

In Figure 7-6, the class genes for *MyBook:Book, Chap1:Struct, Chap2:Struct, Chap1Txt:Text, Chap1Txt:Text,* and *Chap1Pres:Table*, are the element types whose names match those of the respective class names (*Book, Struct, Text,* and *Table*). In this representation, a class describes the attributes of its objects in the corresponding <!ATTLIST> clause(s). Behaviour, which is "described" (that is, coded) in the class API implementation, is made accessible to the class (hence, its objects) in the *behaviour* attribute. Aggregation links between classes are captured in an explicit listing of all element types (classes) that can be nested in a given element type. For instance, the element type *book* has the list *(book | struct | text)* next to its <!ELEMENT> clause. An association link is represented by an attribute whose value is of type IDREF or IDREFS (which stands for "id reference(s)"), for instance, the attribute *present* in the element type *struct*. A generalisation link is denoted by a special read-only attribute, *superclass*, whose value is the name of the element supertype (superclass) of the current element type. (In this case, the element subtype need not explicitly list the attributes inherited from the element supertype, unless they are redefined).

The class gene for the *LifeWeb* class system can be the *LifeWeb* meta-class system. It should be possible to use some meta-modelling technique to build such a system. However, it is not within the scope of this thesis to explain and design genes at higher meta-modelling levels. Briefly, any class at level *n* is also an object to level *n+1*, which may be represented in its own object gene, and described further by another class at the next higher level (*n+1*). This higher meta-level class may again be designed so as to satisfy the *Life Design* constraints applicable to it.

Stateful

The state of a gene can naturally be kept within itself (by the defined notation), and changed (by its genetic functions). In this respect, a gene has the capacity to be stateful. The Web, however, is a stateless system, because it does not remember any changes made by the client (see Section 2.2.2.6). These changes are thus lost before they can reach a gene to be recorded there. This problem is solved in *LifeWeb* with the distribution of object genes, which is explained in Section 7.2.3 below.

Behavioural

The behavioural constraint can be inherently satisfied since a gene is itself an object with well-defined behaviour. A set of methods for each genetic function required can be implemented for it (see Section 6.4.2.4).

Lifetime of genes

Such genes as described above can be stored permanently on a disk, and their lifetime can be made as long as that of the objects they describe.

7.2.3 Stateful constraint

As has been mentioned elsewhere, a *LifeWeb* object (or equivalently, its object gene) is capable of being stateful (having a well-defined state representing by a set of attributes), but cannot record changes made to itself (while being online) because of the statelessness of the Web. This problem is solved in our system by the concept of *object gene distribution.*

As can be seen from Figure 7-1, on the one hand, the separation of contents (via the use of *Material* objects), functional implementation (via the use of *Behaviour* objects), document type (via the use of *Schema* objects), and external services (via the use of *Service* objects) from other aspects of a document (structure and presentation), leaves the *LifeWeb Document* object with a *bare* framework of the document. On the other hand, following *Life Design*, the *Document* object is also the root form that captures the *whole* organism. This means, despite the possibly large document the *Document* object represents, only a minimal amount of information is kept in it. It is simply a holder of virtual contents and references to other objects. This design makes it practically possible for the *Document* object, which is represented by its object gene, to be wholly distributed at client sites. Such a distributed *Document* object alone is sufficient for the reconstruction of the whole document. This concept, which is termed *object gene distribution*, is about distributing objects with minimal but sufficient information to reconstruct the object fully.

As object genes can be fully distributed and permanently kept at client sites, changes to their state (as a result of client actions) can be retained. That is, they are stateful.

7.2.4 Behavioural constraint

The behavioural constraint can inherently be satisfied in the object-oriented design of the system.

7.2.5 Implications on the Web

Life Design in general, and *LifeWeb* in particular, have remarkable relevance and significance on the Web, and specifically address the issues mentioned earlier in Section 2.2. This section comments briefly on some major implications that *Life Design* and *LifeWeb* have in the setting of the Web, considering them in terms of three of the four *Life Design* constraints: structural, stateful and genetic. (The behavioural constraint is significant in enabling the whole system, designed after other constraints, to function the way it is supposed to. Its meanings can be seen in those of other constraints and need not be considered separately.)

7.2.5.1 A flexible modular structure for manageability, maintainability, customisability, adaptability and evolvability

The flexible modular structure (DAG) of a *LifeWeb* document, inherited from object-orientation and *Life Design*, permits easy management and maintenance of the document on a component (object) basis, both during design time and runtime. Easy manipulation of document components also allows the document to be changed for customisation, adaptation or evolution purposes. In addition, the specific structure of *LifeWeb* documents makes it possible for structural links to be automatically generated by the system during runtime. In this way structural referential integrity can be ensured and large maintenance costs on structural links eliminated.

The flexibility of links also has a meaning. They preserve the semantics and internal structure of the evolving entities, whilst giving these entities the necessary "freedom" (flexibility) to change, grow and evolve. For instance, at the object level, the association link between a *Document* object and its *Schema* object cannot be broken (because doing so will alter the schematic definition, hence semantics, of the *Document* object). A link between a *Document* and a *Service* object, on the contrary, can be freely added or removed. At the class level, both association and aggregation links cannot be removed if the removing class participates in the internal structure of the other one (that is, if the removing class is "subordinate" – in the *Life Design* sense as defined in Section 6.4.1.1.4 – to the other one). Generalisation links, on the other hand, can be extended or shrunk, allowing the system to evolve schematically (see Section 6.4.2.1.3). In this way, schema evolution in *LifeWeb* occurs only on a component basis (subclass/superclass), which is different from traditional schema evolution (see Chapter 4). In traditional schema evolution, changes can happen to the internal structure of an evolving entity, which makes it difficult and expensive to maintain system consistency.

Sections 5.2.2 and 9.7 explains the manageability and maintainability aspects of *LifeWeb*. Customisation, adaptation and evolution in *LifeWeb* are explained in detail in Chapter 8, and in Sections 9.4.2 and 9.4.3.

7.2.5.2 A stateful Web for customisability, adaptability and evolvability

As a document's object genes can be kept permanently at client sites, customisation information can be maintained for each client. Furthermore, the distribution of these object genes (originating from one server document) allows the formation of a population of Web documents in which individual members may be slightly different from one another as a result of customisation. Adaptation and evolution can thus happen over such population (and many others, originating from other server documents), drawing its options from the varieties presented by each individual, and driven by user preferences (see Section 6.3.6). Customisation, adaptation and evolution in *LifeWeb* are explained in detail in Chapter 8, and in Sections 9.4.2 and 9.4.3.

7.2.5.3 Gene and recursive gene architecture for evolvability

In the Web context, a *LifeWeb* object genome is a *compact* and *persistent* document instance that retains its state across HTTP connections, and contributes to the "genetic variety" in a population of similar document instances. Functionally, it controls the realisation of the final representation (see Section 6.4.2.2.1) of the document instance. A *LifeWeb* class genome defines a *document type*, based on which document instances of a particular type are generated. Functionally, it controls the instantiation of document instances and allows document instances to validate themselves (thus ensuring interoperability in the entire population of its document instances).

As can be seen in the examples above (Section 7.2.2), and earlier in the explanation about XML (Section 3.2.2), the concept of document type has actually been applied on the Web with the use of the Document Type Definition (DTD). This notion is especially meaningful in the Web environment, which has developed into a global network, and must incorporate a wide variety of culture, standards and needs. Giving the Web system the ability to define various document types provides for this requirement. We will explain in more detail in Section 9.2 how work on DTD is used in the *LifeWeb* prototype to implement class genes.

Higher meta-level genes define constructs that are used in genes at lower meta-levels. In particular, in *LifeWeb*, meta-genes define the language in which genes are expressed, and M2-genes define the ontology for meta-genes. This recursive gene architecture allows evolutionary changes to be propagated from lower to higher meta-level, resulting in a "more genetic" change in the entire system. *LifeWeb* evolution is explained extensively in Chapter 8.

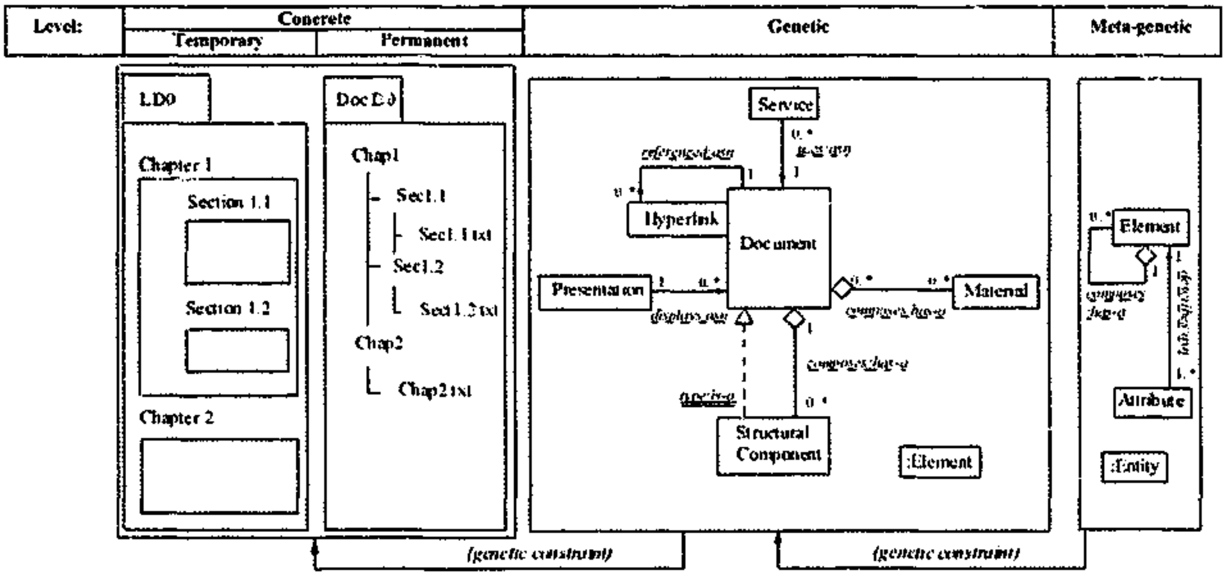
Chapter 8 LifeWeb evolution

Evolution in *LifeWeb* occurs through successive stages, where each stage maps into one corresponding level of meta-modelling described in Sections 6.4.1.2 and 7.2.5.3. The propagation of changes from one stage to another is controlled by a factor named the *evolutionary threshold*. When an *evolutionary threshold* is reached, the change will be carried to the next higher meta-level (of genes), resulting in a more *genetic* change. Such change propagation may result in the formation of a new group of the individuals in the level immediately below, where individuals in these groups are *genetically differentiated*. This is similar to the speciation process in nature [DAR77]. Therefore we have used the term *document species* (and similar terms for higher levels) to denote the basic class of a *document taxonomy*, which is a distinct group of document types that are *genetically interoperable* with each other and conform to the one from which they are derived.

The sections below give a more detailed picture of the evolutionary process in *LifeWeb*. The first section explains the various stages of the process in parallel to the different meta-modelling levels. The second section elaborates the specific types of changes that can occur and propagate to higher meta-levels. The third section uses a scenario to describe the formation of new document types and document species. The fourth section explains the mechanism by which a document type or species becomes extinct or multiplied. The fifth section gives some illustrative examples. Characteristics of *LifeWeb* evolution are drawn up and presented in the sixth section. Finally, the last three sections evaluate *LifeWeb* evolution against other comparable technologies, which include the Genetic Algorithm, meta-modelling, and Database Schema Evolution (see Chapter 4).

8.1 Levels of meta-models and stages of the evolutionary process

LifeWeb evolves through stages that correspond to the levels of meta-modelling. A level may be divided into two sub-levels, *temporary* and *permanent*, if the entities in that level are considered both in their object genes (persistent object representations) and other object representations (usually non-persistent) (see Section 6.4.2.2.1). (For instance, a document displayed in the browser is in its functional, and final, presentation. Its source code, that is, its syntactic representation, is its object gene. The functional representation of a class, however, is not important in our problem, and practically we only consider its syntactic representation.) Figure 8-1 shows the (simplified) *LifeWeb* system through three levels of meta-modelling: concrete, genetic, and meta-genetic, using a *LifeWeb* document represented in XML notation (described in Section 3.2.2) as an example.



Level:	Concrete		Genetic	Meta-genetic
	Temporary	Permanent		

Terms:
-Life Design
-Web
-OO
	living organism
	Web document
	N/A
		object genome
		XML source code
		Document object
			class genome
			Document Type Definition (DTD)
			class system
				meta class genome
				XML specifications
				meta class system

Figure 8-1: Three levels of a LifeWeb document (simplified)

The *concrete level* has two sub-levels: (i) The *(concrete) temporary sub-level* comprises *living organisms* (in *Life Design* terms) or *Web documents* (in Web terms) as they appear in full (in their final object representations, which are functional representations in this case) at the client site (in the browser); (ii) The *(concrete) permanent sub-level* consists of *object genomes* (in *Life Design* terms) or *Document objects* (in object-oriented terms), which are the *XML source code* (in Web terms) of the living organisms at the temporary sub-level. The *genetic level* contains *class genomes* (in *Life Design* terms) or *class systems* (in object-oriented terms), which are the *DTDs* (in Web terms) that define the schemata for different types of *Document* objects (object genomes). The *meta-genetic level* contains *meta-class genomes* (in *Life Design* terms), or *meta-class systems* (in object-oriented terms), which are the *XML specifications* (in Web terms) that define the grammar or notation in which the DTDs (class genomes) are expressed. (Figure 8-1 shows only a simplified version of the XML specifications). Further meta-levels can be specified as required (see below). These triplets of terms may also be used interchangeably in the context of *Life Design*, Web and object-orientation, respectively: *life form, element, object*; and *class gene, element type, class*. For consistency, we will use the Web terms as the main ones in this section, accompanied by the *Life Design* and object-oriented equivalents (in parentheses) for concept mapping.

The following sub-sections describe stages of *LifeWeb* evolutionary process. The role that each stage plays in the whole process is specified, showing what kinds of changes there are, and how they are accumulated and controlled. These sub-sections also explain when the customisation and adaptation processes start and how they can seed the evolutionary process. In this model the *fitness of gene* is measured by the usage of the given gene, or the rate of its occurrence in individual documents.

8.1.1 The concrete level

The concrete level comprises the documents populating the whole Web. This level is divided into two sub-levels: the *temporary sub-level* consisting of Web documents (living organisms or documents in their final representation), and the *permanent sub-level* consisting of document's XML source code (object genomes or *Document* objects).

8.1.1.1 The temporary sub-level

Web documents in this level are realised from their XML source code (object genomes or *Document* objects), fully filled with formatted materials, hyperlinks and other document components, accessible to human readers through their unique URLs. Customisation and adaptation processes start occurring at this level as a result of users' interaction. These processes supply the changes necessary for the evolutionary process to happen. There can be two kinds of change at this level:

- Individual temporary change: changes that alter a Web document (living organism) only from the time the request for change is made until the time the document is brought offline (that is, when the user ends a Web session).

- Individual permanent change: changes that are recorded at the XML source code (object genome or *Document* object) of a Web document. These changes typically result from users' requests or system adaptive engine and are kept at the client site.

8.1.1.2 The permanent sub-level

This level consists of the (documents') XML source code (object genomes or *Document* objects), which can permanently record changes occurring at the temporary level. It has two sub-levels of itself, server and client. The server sub-level consists of original objects created by content providers, possibly manually. The client sub-level contains copies of server objects generated through the reproduction process, possibly modified for customisation or adaptation purposes as explained above. In this respect the set of modified client objects derived from a given server object form a population of Web documents with a variety of genes differing from that of the original server object. Similar changes in client objects are accumulated over time and can trigger a permanent change in the corresponding server object, so that the change will later be duplicated and distributed throughout the next generation of the population. The threshold at which this process starts is called the *object evolutionary threshold.* There can also be two kinds of change at this level:

- Non-genetic change: changes that affect the document's XML source code (object genome or *Document* object) but not its DTD (class genome or class system). These are typically changes to the presentation of structural components, or to the structure of the document (for instance, sorting its structural components). They do not modify the way elements are defined (for example, by adding new attribute types), nor do they add/remove element types (class genes or classes) in the document. This kind of change will not propagate any further than this level.

- Genetic change: changes that affect not only the document's XML source code (object genome or *Document* object) but also its DTD (class genome or class system). The addition or removal of element types (class genes or classes) exemplifies this kind of change. These changes, if happening at a large number of document instances of a given document type, can trigger the formation of a new document type. The threshold at which this process starts is called the *genetic evolutionary threshold.* If the newly created document type does not conform to the previous document type it is derived from, a new *document species* may have been formed.

8.1.2 The genetic level

This level consists of the DTDs (class genomes or class systems) that describe the genetic specifications or the schemata of the corresponding document types. A DTD (class genome or class system) specifies factors such as what element types there are, their attributes and attribute types, and their possible arrangement and nesting. The schematic constructs defined in this level are used to build the XML source code (object genomes or *Document* objects) at the genetic level. Document type schemata control the diversification of these documents, represented in their XML source code (object genomes or *Document* objects), ensuring interoperability and compatibility between them, unless a new document species or non-conformal document type is formed. Similar to the previous level, changes at this level are also classified as genetic (as explained above) or meta-genetic. Accumulated meta-genetic changes, if reaching a threshold called the *meta-genetic evolutionary threshold* can lead to the formation of a new language or notation in which the DTD (class genome or class system) is expressed. Similar to the speciation process, the new notation may start a new *document genus* if it does not conform to the previous notation.

8.1.3 The meta-genetic level

This level consists of the XML specifications (meta-class genomes or meta-class systems) that define the grammar of the language or notation in which the DTDs (class genomes or class systems) are expressed. (Formal) XML specifications (meta-class genomes or meta-class systems) control the speciation process (formation of new document species), ensuring the conformity of various DTDs (class genomes or class systems) to the specified grammar, unless a non-conformal notation or *document genus* is formed. As at the meta-genetic level, there are also two kinds of change at this level, meta-genetic (as explained above) and M2-genetic. When a threshold called the *M2-genetic evolutionary threshold* is reached, accumulated M2-genetic changes can lead to the formation of a new ontology based on which concepts of the Web document system are formed. As in the previous level, a new *document family* may be formed if the newly created ontology does not conform to the previous ontology.

8.1.4 The M2-genetic level

This level specifies the ontology or the total sum of the concepts used to conceptualise the Web document system. The ontological constructs defined in this level are used at the meta-genetic level to define the grammar of the language used to describe the document type schemata, and to ensure the semantic compatibility between the various languages in which the system is represented. Changes happening at this level are assumed to cause what is called "paradigm shift", in which the Universe of Discourse (UoD), or the Web in this particular case, is viewed under a different set of concepts.

There can possibly be more meta-modelling levels above M2-genetic, but they are deemed not necessary to include in *LifeWeb*. Thus we assume the M3-genetic is the *union level* that unifies all ontological systems, and there are no more changes at that level.

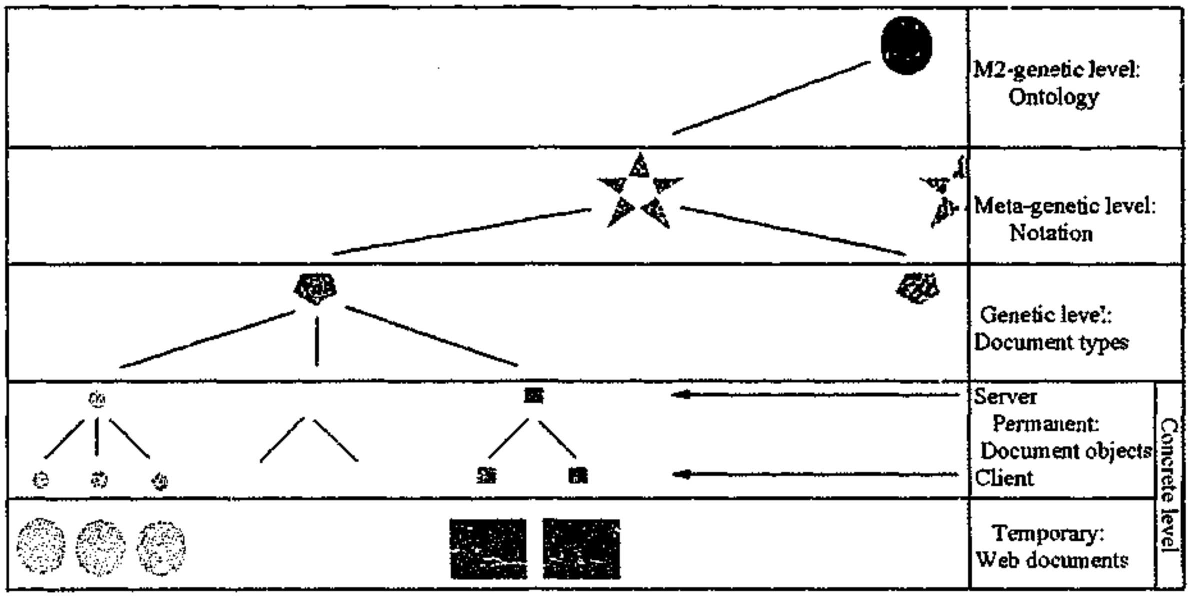


Figure 8-2 : Levels of meta-modelling (evolutionary stages)

In the reality of the Web, some changes at high levels may happen without accumulated changes from lower levels, independent from the ongoing "life" of the Web. These changes are typically made by top-level Web developers or administrators as needs arise. This process is currently common in most information systems (for example, in database management systems such as Orion [BAN87], described in Section 4.1), and we call it *top-down evolution*. It is opposite to the one described above, which we call *bottom-up evolution,* where the system evolves as a result of local changes. In both cases the system develops in a controlled environment under the supervision of higher meta-levels with predictable results, giving itself features of maintainability, manageability, and evolvability. The top-down evolutionary process, however, needs human intervention and is largely dependent on the system designer for the integrity and consistency throughout the system. It will generally not be discussed in subsequent sections of this thesis, unless explicitly mentioned. Most discussions will deal with the bottom-up evolution, which can be automated.

Figure 8-2 depicts different levels of meta-modelling. Entities in each level are defined, controlled, linked and coordinated by the entities in the level immediately above them.

8.2 Change and change propagation

Since explanations in this section are related mainly to object-oriented technology, for readers' convenience, we will, in this section, extensively use object-oriented terminology with the implicit, or in some places, explicit, *Life Design* and Web equivalents (that are defined in the previous section).

As an overview, in *LifeWeb*, except for state (attribute or pure data value) change, all changes (in bottom-up evolution) happen only on a component (object or class) basis at the flexible links. The addition, removal or modification of a feature (attribute, method or link) in an existing class requires that a new class (with the desired feature) be developed and introduced into the system (through an appropriate generalisation relationship) (see also Section 8.3). An old class may be removed from the system as a result of the addition of a new class, or co-exist with the new class, depending on each particular situation (see also Section 8.3). The following sub-sections give explanations specific to the classes in *LifeWeb* about how each kind of change happens and how they are propagated.

8.2.1 Compositional change

Compositional changes are those that happen to objects that connect with each other through the *composes:hasa* links in the *LifeWeb* class system (see Figure 8-1). Typical changes of this kind are reordering, adding or removing *StructuralComponent* objects (or objects of a derived class thereof, as explained in Section 8.2.5) in the document. Such a change is usually the result of a customisation request direct from the user, or some system adaptation caused directly by the user's interaction, happening at the temporary sub-level of the concrete level.

This kind of change usually propagates up as high as the permanent sub-level (client or server) and does not result in a genetic (schematic) modification in the document instance. An exception to this is when the object to be removed is the last one in the document, or the first one of a newly introduced class, that is a *hybrid* class (see Sections 6.4.1.2 and 6.4.2.2.2. See also Sections 8.2.5, 8.2.6 and 8.5). In these cases the document instance will either not exhibit properties and behaviour belonging to the removed object, or exhibit the added ones belonging to the newly acquired object, respectively. (This is effectively a genetic or schematic change at the concrete level through the composition link.)

The frequency of the disappearance of an existing class, or appearance of a new (hybrid) class in client *Document* objects (object genome or XML source code) (of a given server *Document* object), is recorded, and may trigger a similar change in the corresponding server *Document* object if it reaches the object evolutionary threshold (see Section 8.1.1). If the same change is found in many other server *Document* objects of a given document type (DTD, class genome or class system), a new document type may be formed when the frequency of this change reaches the genetic evolutionary threshold (see Sections 8.1.2). If the change is the disappearance of an existing class, it may be removed from the relevant document type if and only if it is free of all dependency (see Section 6.4.2.4.3), otherwise no change will be made. If the change is the appearance of a new (hybrid) class, it will be added to the relevant document type, with appropriate relationships, forming a new document type.

This is the way that unused classes may be removed and new classes introduced to the system automatically from the concrete level as a result of users' interaction. Sections 8.3 and 8.5 give examples of the formation of document types and species.

8.2.2 Presentational change

Presentational changes are, like compositional changes, primarily local to a Web document or (client) *Document* object (object genome or XML source code), but may also propagate to higher levels. These are changes that happen to objects participating in the *displays:asn* link in the *LifeWeb* class system (see Figure 8-1). Typically a presentational change happens as a result of a customisation request directly from the user, or of some internal system adaptation. Such a request (or adaptation) essentially alters the presentation directives of the associating object, causing the object to be displayed in the new format. A presentational change propagates in a manner similar to a compositional change. This means it does not result in a genetic (schematic) change, unless the involving *Presentation* object introduces a new (hybrid) class (see Sections 6.4.1.2 and 6.4.2.2.2), or removes the final occurrence of its class in the document instance, and this change is found in a sufficiently large number of document instances of a given document type (see also Sections 8.2.5, 8.2.6, 8.3 and 8.5). In this way new presentation classes may be added to, and unused ones removed, from the system.

8.2.3 Referential change

Referential changes happen and propagate in a manner similar to compositional or presentational change. Typically a referential change is a system adaptation or user customisation request, which suppresses or shows *Hyperlink* objects (thus creates a different navigational path). Referential changes are also mainly local, and are carried to the genetic (schematic) level only if they introduce a new (hybrid) *Hyperlink* class (see Sections 6.4.1.2 and 6.4.2.2.2) or remove the last instance of a *Hyperlink* class in the involving document (see also Sections 8.2.5, 8.2.6, 8.3 and 8.5). This is how new derivatives of the *Hyperlink* class may be added and unused ones removed from the system.

8.2.4 Change of external service

The concept of external service is borrowed from a concept in biochemistry where functional groups are attached to molecules to modify the properties of these molecules (see Section 6.2.5).[23] External services are complete, encapsulated, functional units that can be freely mounted or unmounted onto a document at any time to extend the document functionalities. External services play an important role in the *LifeWeb* evolutionary process in that they allow for the incorporation of third-party software, dynamic customisation and automatic adaptation and evolution (see also Sections 9.4.2 and 9.4.3).

The use of external services is also significant because it allows the system to accommodate complex customisation and adaptation processing. A *Service* object can invoke the primitive methods (for instance, those implementing the genetic functions) defined in *LifeWeb* classes to perform more sophisticated tasks. For example, based on user requests or profile, a *Service* object can select structural components that satisfy some specified selection criteria and perform complex format processing, or it can sort structural components according to different orders based on different criteria.

Another useful application of external services is to confine customisation information. For example, the information regarding a customisation request (for instance, formatting a structural component in a client document as a table) can be fully encapsulated in a *Service* element (for instance, *TableFormatter*). Thus, there is generally no change made to other elements in the document, and customisation information is kept separate from the original document state. In this way the original state can be easily restored when required. Besides, this separation makes it simpler to keep the client document state up-to-date with the server document state, while customisation changes are still maintained. The implementation and mechanism of *Service* are explained in Sections 9.4.2 and 9.4.3.

8.2.5 Addition of a new class

A new class can be added at either the concrete level or genetic (schematic) level. It must be derived from an existing class in the document type being modified. The newly introduced class may have relationships with other classes, both new and existing, provided that the integrity of the class system

[23] A somewhat similar idea is also found in ordinary life. F›r instance, auxiliary equipment (for example, a diving suite) is used to extend functionalities of the human body (for example, diving).

is maintained.[24] (That is, the class system before or after change must form a DAG with self-loops.) In this respect, the addition of a class *C* in the inheritance tree may lead to the addition of, for instance, another class *C'* that is linked to it. *C'* may be connected with *C* by generalisation, aggregation or association, and the addition of *C'* must be done together with the addition of *C*. (This is effectively an addition of a class subtree or subsystem.)

At the concrete level

The presence of a new class in a document instance without that class being defined in the corresponding class system (main class genome or main DTD) is the result of *hybridisation* (see Sections 6.4.1.2 and 6.4.2.2.2). In this case, the new class must be defined in another class system (that is, a hybrid class genome or hybrid DTD). (In XML specifications, this DTD is accessible to the document either via the XML *namespace* mechanism, or as an *internal DTD* that is inserted into the document, as described in Section 3.2.2.) In this way, the new class (which should be initially supplied by a schema designer) is first introduced into the system as a hybrid class through a document instance (possibly by the system or user selection) via the flexible links. This change, which starts at the concrete level, can subsequently lead to a genetic (schematic) change and the creation of a new document type, following the same change propagation process as explained in Section 8.2.1.

It is worth noting that the introduction of a new (hybrid) class into a document instance effectively changes the document type (class genome or class system) of that particular instance (regardless of whether it will lead to a genetic change or not). This is the way that *LifeWeb* supports genetic (schematic) change from the concrete level on a component (object and class) basis through the flexible links.

At the genetic level

Classes added directly at the genetic (schematic) level are most likely done manually by Web administrators or developers in the top-down evolutionary process (see Section 8.1). In this case document types are created manually. This process is not automated and therefore not supported by the system. However, the existence of a meta-class system (meta-class genome or XML specifications) makes it possible to validate the new document type against the specifications defined for it at the meta-genetic level. This allows for meta-level compatibility and interoperability across document species of a given document genus (grammar).

[24] As a new class must have been designed and implemented by a human programmer, system integrity is maintained by the development environment (for example, the compiler).

8.2.6 Removal of an existing class

A class can be removed starting at either the concrete or genetic (schematic) level. In any case, a class can be removed if and only if it is free of all dependency (see Section 6.4.2.4.3). Any class that becomes disconnected from the class system, as a result of this removal, will also be removed, and may become a "free class" (free form) or destroyed (see Sections 6.4.2.1.2 and 6.4.2.4.3).

At the concrete level

A class will be depreciated, that is, disappear in the server *Document* object, if it is not used in a sufficiently large number of client *Document* objects. It might eventually be removed from the DTD (class genome or class system) if its degree of depreciation has reached the genetic evolutionary threshold and it is free of all dependency. This process is described in Sections 8.2.1 and 8.3.

At the genetic level

A class that is not used in any *Document* objects can be removed manually by Web administrators or developers provided that it is free of all dependency. As this requires a thorough check at the instance level, it might not be practical to do so unless supported by some automatic checking mechanism.

It can be seen that a class can seldom be removed permanently from the system due to the strict constraint of dependency clearance (see Section 6.4.2.4.3). It is more often however, that a class (or a branch of classes) may be removed from a document type or document species if it is a leaf node in the class inheritance tree. This is actually the start of a speciation process, which will be explained in detail in Section 8.3 below.

8.2.7 Changes at the meta-genetic and higher level

As classes (class genes or element types) that follow *Life Design* are also life forms, meta-classes (meta-class genes) and higher meta-level classes (higher meta-level class genes) can also change and evolve in the same fashion as life forms, or in this case, *Document* objects do. It is, however, not within the scope of this thesis to design the specifications and detail the evolutionary process of meta-classes and higher meta-level classes.

8.3 Formation of new document types and document species

The formation of a new document type or document species starts with the introduction of a new class to the class system. The following scenario illustrates the speciation process in a *Life Design* system, in particular, *LifeWeb*.

Suppose, at time T_0, the class inheritance tree of the document type D_0^A of document species A consists of only the four core classes of the model, which are C_1^A to C_4^A, connected with each other as shown in Figure 8-3 (a).

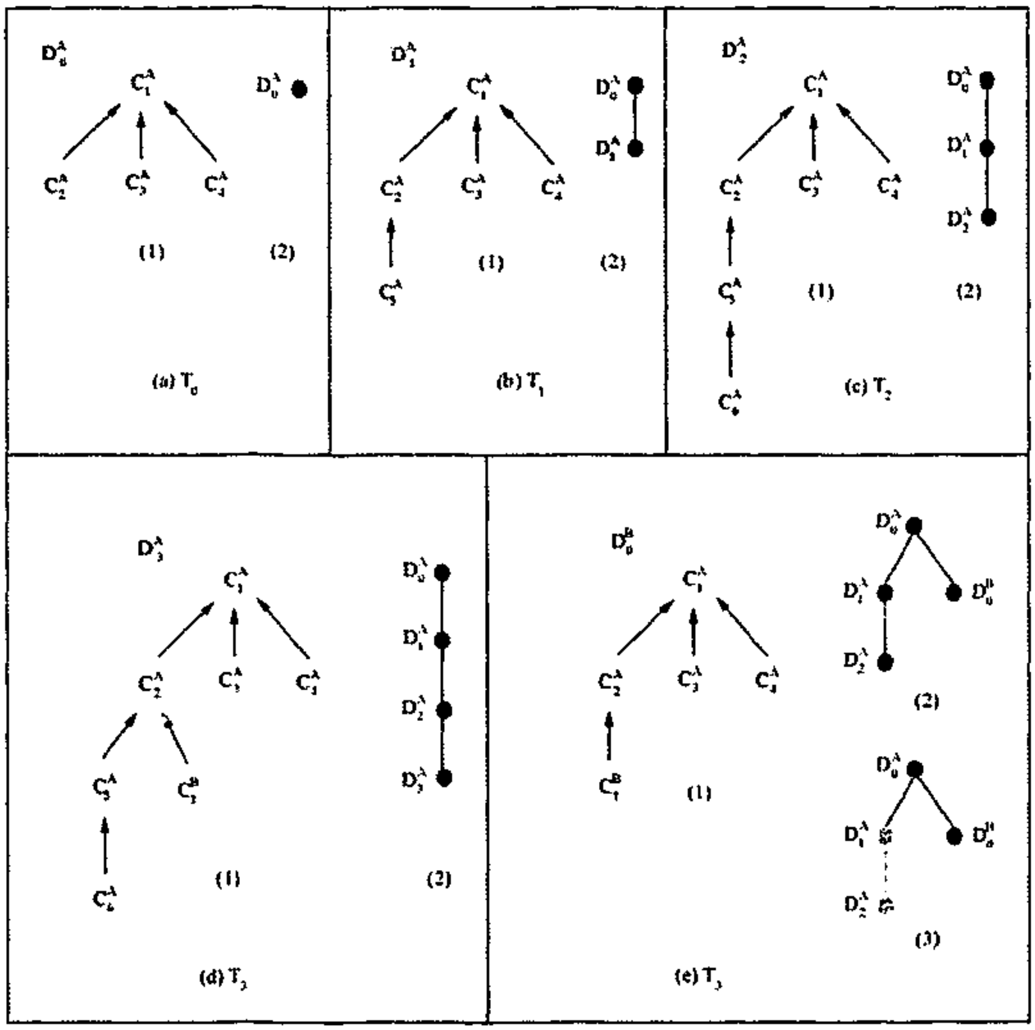


Figure 8-3: Speciation and the evolutionary tree

At time T_1 a new class C_5^A, which is derived from one of the four initial classes, for instance C_2^A, and has been successfully developed from the concrete level, is registered to D_0^A, forming a new document type D_1^A (see Figure 8-3(b)(1)). As D_1^A has all the classes in D_0^A and the extra class

C_5^A of D_1^A is derived from C_2^A in D_0^A, it conforms to D_0^A, and also belongs to document species A. The evolution tree has grown from D_0^A to D_1^A as shown in Figure 8-3 (b)(2).

Suppose C_5^A is specialised further so that at time T_2, a new class C_6^A, derived from C_5^A, is registered to D_1^A, forming a new document type D_2^A (see Figure 8-3 (c)(1)). The evolution tree also grows from D_1^A to D_2^A as shown in Figure 8-3 (c)(2).

Suppose further that it is necessary to remove some features in class C_5^A. As this cannot be achieved by any derivation from C_5^A, a new class C_1^B is derived from the core class C_2^A. There are two possible cases: in the instances where C_1^B is introduced, C_1^B can either co-exist with C_5^A or substitute C_5^A.

1 C_1^B co-exists with C_5^A: C_1^B and C_5^A are simply complimentary to each other, and neither shall be the cause to remove the other at any level of meta-modelling. The addition of a new media type (as a derivative of the *Material* class in the *LifeWeb* data model, as depicted in Figure 7-1) exemplifies this complimentary relationship (among different media types). In this case, if C_1^B is found sufficiently fit at the concrete level to be propagated to the genetic level, a new document type D_3^A will be formed and the evolution tree will also grow as shown in Figure 8-3 (d).

2 C_1^B substitutes C_5^A: C_1^B and C_5^A compete with each other starting in the instances that contain them. An example of this competition relationship is the addition of a modified version of an existing class in which both new and old classes have the same semantics but different features and one is meant to replace the other. Figure 8-3 (e)(1) shows the class inheritance tree of one particular instance in which C_1^B is introduced for this case. (In this instance C_1^B is still a *hybrid* class.) There are two cases in the competition relationship.

2.1 If the new class C_1^B does not find enough favour to survive the environment, that is, it does not multiply enough in instances, it will never propagate to higher level and may eventually disappear. In this case the old class (gene) C_5^A has defeated the new class (gene) C_1^B.

2.2 If on the other hand, C_1^B is found sufficiently fit to the environment to eventually be propagated up to the genetic level, it will be registered to D_0^A, forming a new document type. As the current document type (D_2^A) does not have all its elements in the new document type

(C_5^A and C_6^A are missing), that is, the new document type does not conform to the current document type, a new branch of document type is formed. This new branch starts from the last existing document type that the new one conforms with, which is D_0^A in this example (see Figure 8-3 (e)(1)). This essentially starts a *new document species* B with the first document type D_0^B (see Figure 8-3 (e)(2)). The removal of class C_5^A at the genetic level is considered in two cases:

2.2.1 Class C_1^B completely substitutes class C_5^A in all instances, which means C_5^A is now clear of instance dependency. There are two sub-cases for this:

2.2.1.1.1 Class C_6^A is still used in some instances – In this case C_5^A cannot be removed due to the schematic dependency constraint. The two document species A and B co-exist (realised in two document types D_2^A and D_0^B respectively), having a common ancestor D_0^A. This is the case where traces of ancestor's genes can still be found in the genes of current instances, but not realised. The evolution tree grows as shown in Figure 8-3 (e)(2).

2.2.1.1.2 Class C_6^A is not used in any instances – In this case both C_5^A and C_6^A are free of all dependency and will be removed permanently from the system. Consequently types D_1^A and D_2^A will also become *extinct*. The evolution tree in Figure 8-3 (e)(3) depicts this process, with the blurred nodes and line indicating a branch that has been terminated. This however can scarcely happen.

2.2.2 Class C_1^B partially substitute class C_5^A in some instances – In this case C_5^A cannot be removed due to the instance dependency constraint. The two document species A and B co-exist, having a common ancestor D_0^A (see Figure 8-3 (e)(2)).

This example shows how the speciation process in *LifeWeb* forms the *evolution tree*, which is very similar to the *tree of life* in nature as generally accepted by biologists [DAV98b] (see Figure 8-4). In the example above, for simplicity, the speciation event happens at a fairly early stage in the process, so that the branching occurs immediately at the root node of the evolutionary tree. In reality this event may be delayed much further and the evolution tree would picture the same as the tree of life in nature.

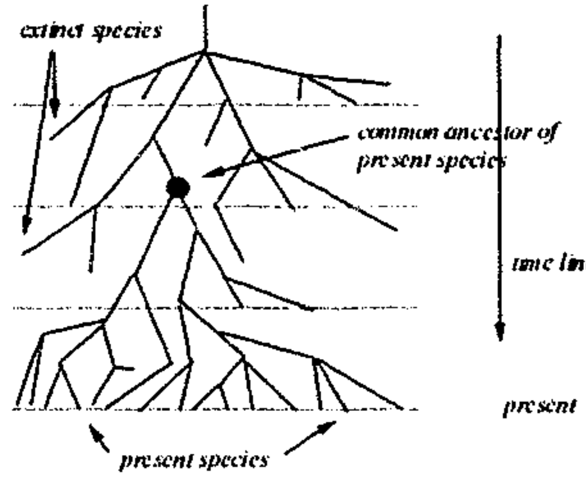


Figure 8-4 : Tree of Life

8.4 Multiplication *and* extinction *of document types and document species*

In nature a species spreads (multiplies) or becomes extinct through the reproduction cycle of an organism. It is a process of two opposite events through which the birth and death of an organism allow the fitter genes to be selected, improve and multiply, and the less adaptable genes dismissed, depreciated and become extinct. In other words, genes become multiplied or extinct by means of their instances through the reproduction cycle. In normal cases, the factor that controls this cycle is the life expectancy. By inducing death, this factor allows for new organisms to be born, while still maintaining some equilibrium state of the system.

A similar mechanism is designed in *LifeWeb* to control the multiplication and extinction of document types and document species, essentially maintaining scalability in the system. The following subsections explain this mechanism in terms of *life expectancy*, *reincarnation* and the *multiplication/extinction* of document types and document species.

8.4.1 *Life expectancy* and *reincarnation*

At birth all *LifeWeb* documents are assigned a *life expectancy*, which is held in the property *expiration*, to control their life span. There are two types of expiration:

- *relativeExpiration* specifies the amount of time that the object can live from the time it is last used (accessed)
- *absoluteExpiration* specifies the maximum amount of time that the object can live

If an object has not been accessed (used) for a *relativeExpiration* period of time, or if it is still used but has reached the *absoluteExpiration* time, it will be destroyed. In the first case, the object is destroyed forever. In the second case, the object will be immediately *reincarnated* without changing its identification but possibly with some updates to reflect the most recent evolutionary change in the system. In software engineering terms, this is simply a "restart" operation with updates.

8.4.2 *Multiplication* and *extinction* of document types and *document species*

The spread of a new document type or document species (class genome) has to be done to both newborn instances and existing ones.

With newborn instances, when a new document is created (most likely manually), it is possible to have it fashioned after the fittest, presumably the most recent document type that exists for a chosen document species (existing or new one).[25] This forms a *new generation* for that document species (or the first generation of a new species).

With existing instances, the most recent document type will also be assigned to them on reincarnation, which at the same time may release their original document type (if the original one is different from the new one). This assignment is doable because new types always conform to old ones of the same species, and the reincarnated instance can be validated under the new type. This process, which is repeated many times over the entire population of the species, eventually replaces all occurrences of old document types with new ones. That is, the new types are *multiplied* while the old types become *extinct*. A document species becomes extinct when the last document type defined for it is removed. This should happen only after the last instance defined by the last document type of that species is destroyed forever.

This mechanism (of using life expectancy to control the spread and extinction of document types and document species) allows for asynchronous and smooth migration of existing document instances to new types without creating a sudden heavy load on the system by the time a new type is formed. As

[25] Similar to the results of evolution in nature where existing species are proven to sustain better, new document species or types have also undergone a selection process to prove themselves more favourable. It is therefore possible to differentiate the fittest document type among many others of a given document species, based on their time stamp.

document instances can reproduce themselves, reincarnation can be done by themselves in a decentralised manner.

It is noted that in bottom-up evolution (which is assumed to be the normal system operation), it is generally not necessary to migrate individual elements (objects) in a given document instance to a new element type (class) that substitutes an existing type. This is because in this case, the new element type can be included in the main document type (main class genome) of the document instance only after it, as a hybrid type, has successfully competed with an existing type (in the main document type) from the concrete level. That is, in most cases (depending on the value of the genetic evolutionary threshold), the element (object) must have been defined in the new (hybrid) element type before this new type is (permanently) registered in the main document type (main class genome) of the given document instance.

It is also noted that in the literature of evolutionary computing, evolution is usually considered in two aspects: *local* and *global* [KUK00]. We are dealing only with global evolution in the sense that the evolutionary process described here is considered to happen over the entire Web, rather than at a specific Web site. Local evoluion is not within the scope of this thesis and will not be discussed here.

8.5 Some scenarios

This section gives some concrete scenarios that illustrate the evolutionary process described in the previous sections. In these scenarios (as well as in our *LifeWeb* implementation), the evolutionary threshold consists of two factors: the *minSize* factor, which specifies the minimum population size required before the other factor, *minRate*, takes effect. The *minRate* factor dictates the minimum rate of occurrence (or absence) of a gene for it to be added to (or removed from) the next higher level. (It is necessary to have the *minSize* factor to ensure that a reasonably large population is present before the evolutionary process is applied, since the size of this population is not known from the beginning). For example an object evolutionary threshold with *minSize* = 500 and *minRate* = 0.8 means that there should be at least 500 client documents for a given server document, and at least 400 of them (0.8*500) should bear (or not bear) gene *g*, for *g* to be added to (or removed from) the higher level (the server document).

8.5.1 Scenario 1

This scenario illustrates the competition of three object genes at the concrete level.[26] In this competition, these genes exist in a population of client documents, but only one of them can be present at a given server document.

Suppose that a server document *D* has a structural component *S* that can be displayed in the format of a table (*T*), list (*L*) or frame (*F*), subject to user customisation. Each of these formats represents an object gene that is being considered. Suppose further that in the object evolutionary threshold, *minSize* is equal to 5 (a small number for simplicity), and *minRate* 0.8. Suppose also that the structural component *S* in the server document *D* is originally displayed as a table (*T*), and there is no client document to start with. The scenario happens as follows. One day after the server document *D* is published, it is visited by several users and three customised client documents are created, all using *L* to display the structural component *S*. The situation is as follows:

Day 1		
Gene X	Number of occurrences n_X	Rate of occurrence (fitness) $f(X)=\frac{n_X}{\sum n_X}$
T	0	.0
L	3	1.0
F	0	.0
Total	3	

Table 8-1 : Observed fitness values after one day

As shown in Table 8-1, gene *L* has achieved a fitness factor of 1.0, which is above the *minRate* of 0.8, but it has occurred in only 3 individuals, which is below the *minSize* of 5. Consequently, no change is made to the server document at this stage. After two days, 10 client objects are created, therefore the *minSize* requirement is satisfied. However, no gene has achieved the *minRate* value, thus the server object is still unchanged (see Table 8-2).

[26] This is similar to the competition of class genes (at the genetic level) described in Section 8.3, but happens at the concrete level and does not make any changes to the evolution tree. It is also noted that although there is complementary relationship between class genes, no such relationship exists between object genes. (Because a class gene may define many instances but an object gene defines only one).

Day 2		
Gene X	Number of occurrences n_X	Rate of occurrence (fitness) $f(X)=\frac{n_X}{\sum n_X}$
T	5	.5
L	4	.4
F	1	.1
Total	10	

Table 8-2 : Observed fitness values after two days

After a month, 100 client objects are created, and 80 of them use L to display the structural component S (see Table 8-3). That means, gene L has achieved a fitness factor equal to the *minRate* threshold. As the *minSize* requirement is also satisfied, gene L will be propagated up to the server document D, changing the presentation of the structural component S in the server document D from T (table) to L (list).

In this process, it is remarked that for a given feature (the presentation of the structural component S in this example), the more variety of genes (formats) there is, the more difficult it is for a change to propagate up to higher level. This is because the competition among the genes is higher. The *minSize* factor plays an important role to ensure a sample statistically representative (large) enough for the whole population, which is still growing. The greater value of *minSize* also delays and reduces the possibility of change propagation. This example illustrates how the *LifeWeb* evolutionary model can be deployed to achieve optimal setting (in terms of user preferences) for a particular feature in a server document.

Day 30		
Gene X	Number of occurrences n_X	Rate of occurrence (fitness) $f(X)=\frac{n_X}{\sum n_X}$
T	15	.15
L	80	.80
F	5	.05
Total	100	

Table 8-3: Observed fitness values after one month

8.5.2 Scenario 2

This scenario illustrates the propagation of a complementary class gene from the concrete to the genetic level.

In continuation of the scenario above, suppose that the classes Table (*CT*) and List (*CL*) are defined in the document type *DTD1*, but the class Frame (*CF*) is defined in another document type, *DTD2*. We will consider in this example, the registration of *CF* in *DTD1*, that is, its propagation from document instances (defined in *DTD1*) to *DTD1*. Suppose that initially there are three server documents defined in *DTD1*, including the server document in the previous scenario. All of these server documents have a structural component, which may be displayed in the format of a table (*T*), list (*L*) or frame (*F*), and which they all initially display in *T*. We now call these server documents *D1, D2* and *D3*. The *minSize* and *minRate* required for *CF* to be added to *DTD1* is 5 and 0.6, respectively. (Note that these factors comprise the *genetic* – not *object* – evolutionary threshold applied on the population of the *server* documents, not the *client* ones). The object genetic threshold for all three client document populations derived from *D1, D2* and *D3* is still 5 and 0.8 for *minSize* and *minRate*, respectively. The scenario happens as follows.

Day 1							
	Gene X	Number of occurrences n_X			Rate of occurrence (fitness) $f(X)=\frac{n_X}{\sum n_X}$		
Client		D1	D2	D3	D1	D2	D3
	T	0	2	0	.0	.4	.0
	L	3	1	3	1.0	.2	1.0
	F	0	2	0	.0	.4	.0
	Total	3	5	3			
Server	*CT*	(*D1*, *D2*, *D3*) 3			1.0		
	CL	0			.0		
	CF	0			.0		
	Total	3					

Table 8-4: Observed fitness values after one day

One day after each of the server documents is published, *D1* generates 3 client documents, all using gene *L*. The client document population of *D2* consists of 5 client documents, 2 of those using *T*, 1 using *L* and 2 using *F*. The *D3* population is like the *D1* one, with 3 client documents, all using *L*.

Since none of the client document populations of *D1, D2* or *D3* have any gene that passes the object evolutionary threshold, *D1, D2* and *D3* remain unchanged. The situation after one day is shown in Table 8-4.

After one week, object gene *F* in the population of client documents of *D2* has passed the object evolutionary threshold (*minSize* = 5 and *minRate* = 0.8). The class gene *CT* in *D2* is thus changed to *CF*, increasing the rate of occurrence of *CF* from zero to 0.35. However, *CF* is not yet registered in *DTD1*, since both the *minSize* (5) and *minRate* (0.6) requirements for *CF* are still not satisfied (see Table 8-5).

Day 7							
	Gene X	Number of occurrences n_X			Rate of occurrence (fitness) $f(X)=\frac{n_X}{\sum n_X}$		
Client		D1	D2	D3	D1	D2	D3
	T	1	1	5	.125	.1	.5
	L	5	1	3	.625	.1	.3
	F	2	8	2	.25	.8	.2
	Total	8	10	10			
Server	*CT*	(*D1, D3*) 2			.65		
	CL	0			.0		
	CF	(*D2*) 1			.35		
	Total	3					

Table 8-5: Observed fitness values after one week

After one month, two more server documents (*D4* and *D5*), which also use only *CT*, are added to the population of server documents, and the situation has been changed as shown in Table 8-6.

At this stage, the two newly added server documents *D4* and *D5* quickly generate two populations of client documents that highly favour the use of object gene *F*. Since the object evolutionary threshold for object gene *F* is satisfied in both of these populations, the class gene *CT* in both *D4* and *D5* are changed to *CF*. This increases the rate of occurrence of *CF* from 0.35 to 0.6 over the entire population of server documents defined in *DTD1*. Since 0.6 is also the *minRate* required in the genetic evolutionary threshold for *CF*, and the respective *minSize* requirement (5) is also satisfied, *CF* is registered to *DTD1*.

Day 30											
	Gene X	Number of occurrences n_X					Rate of occurrence (fitness) $f(X)=\frac{n_X}{\sum n_X}$				
Client		D1	D2	D3	D4	D5	D1	D2	D3	D4	D5
	T	10	30	50	1	1	.125	.1	.5	.1	.1
	L	50	10	30	0	1	.625	.1	.3	.0	.1
	F	20	60	20	9	8	.25	.8	.2	.9	.8
	Total	80	100	100	10	8					
Server	CT	2					.4				
	CL	0					.0				
	CF	3					.6				
	Total	5									

Table 8-6: Observed fitness values after one month

This scenario shows that besides the popularity of a gene, the addition or removal of client and server documents to or from their respective populations can largely affect the chance of a gene being propagated (its survival).

8.5.3 Scenario 3

This scenario illustrates the competition of combinations of object genes (instead of only one gene as in Scenario 1) at the concrete level.[27] In this competition, combinations of genes exist in a population of client documents, but only one of them can be present at a given server document.

Suppose now that the structural component S (in Scenario 1) has two subcomponents containing an image and an audio file respectively, to illustrate its contents. These subcomponents can be added or removed in client objects, subject to user customisation. As each subcomponent can be either present or absent in the document (D), there are $2^2 = 4$ different possibilities, and the genetic patterns are represented in Table 8-7 below:

[27] This example is particularly given to compare evolution in *Life Design* and Genetic Algorithm (see Section 8.6).

Possibility number	Image	Audio	Binary representation
1	Absent	Absent	00
2	Absent	Present	01
3	Present	Absent	10
4	Present	Present	11

Table 8-7: Possibilities of customisation

Suppose that initially both the image and the audio files are present in the server document *D* (genetic pattern "11"), and the values for *minSize* and *minRate* are still the same as in Scenario 1 (5 and 0.8, respectively). The scenario happens as follows.

After one day, 5 client documents are created from the server document *D*, but none of the genetic patterns (X_i) has reached the *minRate* (Table 8-8). Consequently, no change is made to the server document at this stage.

Day 1			
Possibility number i	Genes X_i	Number of occurrences n_X	Rate of occurrence (fitness) $f(X) = \frac{n_X}{\sum n_X}$
1	00	1	0.20
2	01	1	0.20
3	10	2	0.40
4	11	1	0.20
	Total	5	

Table 8-8 : Observed fitness values after one day

After 30 days, there are 100 client documents created. The pattern "01" scores highest in the population (0.72), but is still under the required *minRate* (0.8) (see Table 8-9). The server object therefore still keeps its pattern of "11", even though at this stage the rate of occurrence for that pattern is almost zero.

This scenario illustrates how features may be combined to evolve together in *LifeWeb*. Usually features are combined when they are related. (In this example, the above features are related in the sense that they are all used to illustrate the textual contents of the structural component *S*). The more features are combined together, the lower the possibility for any combination to propagate to higher level (hence, the lower the possibility for changes at higher level). Some combination may also disappear at one or both levels (but this does not lessen the genetic variety as in the case of the Genetic Algorithm, which is explained in Section 8.6). It is noted that combinatorial explosion (that is, too many genetic patterns exist so that their fitness evaluation becomes computationally impossible) is not likely a problem here. This is because what can be combined to evolve together are only the object genes within one *single* (server) document (*not among* documents). (In the above example, they are the object genes of the image and audio elements). Furthermore, the number of object genes in one document that can be meaningfully combined (and propagate together to higher meta-levels) is quite limited.

Day 30			
Number i	Genes X_i	Number of occurrences n_X	Rate of occurrence (fitness) $f(X) = \frac{n_X}{\sum_i n_X}$
1	00	10	0.10
2	01	72	0.72
3	10	15	0.15
4	11	03	0.03
	Total	100	

Table 8-9: Observed fitness values after one month

8.5.4 Conclusion

These scenarios show how *LifeWeb* can be employed in a simple way to build adaptive Web documents. System evolution to higher genetic level also happens in a similar fashion. This simplicity is achieved due to the fact that user inputs have been used not only to drive the selection of the fittest genes but also to generate genetic variety, including introducing new genes. This is not the case with the Genetic Algorithm, explained previously in Section 4.3, and in Section 8.6. The evolutionary threshold, which in *LifeWeb*, consists of the two factors *minSize* and *minRate*, plays a critical role in controlling system evolution. Working out appropriate values for these factors can perhaps form an independent area of study.

8.6 LifeWeb evolution and the Genetic Algorithm

LifeWeb consists of populations of self-reproducing entities with various genetic traits, which can be altered to improve their fitness to user and developer requirements. These have made it feasible to apply the Genetic Algorithm (GA) (see Section 4.3) to *LifeWeb.*

Evolution in the GA, however, differs from that in *LifeWeb* in several aspects.[28] In the GA, new genes are internally generated mainly by crossover (with random recombination) and random mutation (secondary). In *LifeWeb,* they are externally produced by human users, authors or developers, and purposefully introduced into the system through user customisation. For instance, users may create their own presentational objects in their client documents to suit their needs; authors may create a pool of "ready-made" presentational objects in their server documents for users to choose from; and developers may create a pool of classes (element types) for authors to try on their (server) documents. This is a major difference between the two systems because they evolve in manners that are opposite with each other: one is random (thus is less efficient in terms of environmental fitness)[29] and the other controlled (thus is more efficient in the same terms).

Another difference is that the GA may terminate if a termination condition is reached or an optimum is found.[30] In *LifeWeb,* evolution is a process that lasts as long as the Web system does, unless the whole process is abolished (for instance, by its creator). There are also some other minor differences, for instance, in the GA, features (parameters) are typically combined together to evolve, whereas in *LifeWeb,* they often evolve singly.

The application of the GA to *LifeWeb* requires some change to the *LifeWeb* evolutionary model described in previous sections. As has been explained, in the GA, genetic strings are generated internally by the system through crossover or mutation, not from user requests for customisation as in

[28] Note that here we are comparing only *LifeWeb evolution,* not *evolvability,* with evolution in the GA. As has been mentioned in Section 4.3 and throughout the previous chapters and sections, evolvability has a larger problem domain than evolution.

[29] The effect of randomness in GA evolution may be reduced, but not eliminated, by the application of some prior knowledge [JON88].

[30] In a changing environment, the fitness function may change and a GA may be left continuously running so that individuals can continuously adapt themselves to the new environment. Typically however, a GA is made to terminate after a certain number of execution cycles. This is because of time and resource constraint and also, as has been mentioned before (Section 4.3), the focus of GAs is optimisation, not evolvability.

LifeWeb. Because these strings must be exposed to all users (the environment) to have their fitness evaluated, they cannot be instantiated in (personal) client documents, but in a number of server documents, accessible to all users. (For instance, in the example in Scenario 3 above we can create a population of 2 server documents for the 4 different genetic patterns to compete with one another.) As such, gene fitness can no longer be measured by the rate of occurrence of a genetic pattern (because there are not enough individuals carrying that pattern to make a statistically significant figure), but by some other measurement, for instance, its utilisation. The application of the GA on *LifeWeb* is probably most suitable to produce a (server) document instance with a set of optimal genes. The fitness function for this application is yet a question to be investigated.[31]

8.7 LifeWeb versus meta-modelling

As has been mentioned earlier (Section 4.2.2) and can be seen from the explanations of *Life Design* (Sections 6.4.1.2 and 6.4.2.2.2) and *LifeWeb* (Section 7.2), meta-modelling concepts have been applied in *Life Design* and *LifeWeb* to construct the recursive gene architecture in which individual entities systematically evolve in a bottom-up fashion (see Section 8.1). This architecture generally follows the same principles as standard meta-modelling frameworks such as the ISO/IEC IRDS [IRD90] (see Section 4.2). In addition, it extends the IRDS architecture with the concepts of the union level, and hybridisation, as explained previously in Section 6.4.1.2. Similarities and differences are also found between the *Life Design* recursive gene architecture and the UML Four-Layer Metamodel Architecture [UML98] (which can be considered an implementation of the IRDS). These are explained before in Section 6.4.1.2.

In brief, existing meta-modelling concepts and techniques have been largely applied in *Life Design* and *LifeWeb*, and extended to introduce new ones useful for system evolution. These concepts and techniques, combined with object-oriented technology, enable the design of a self-evolving system.

8.8 LifeWeb versus Database Schema Evolution

Life Design (hence *LifeWeb*) and Database Schema Evolution (DBSE) share the same concerns about enabling dynamic changes to the description (system schema) that models a particular system instance, and the consistent management of these changes (see Section 4.1). In this section, *LifeWeb*

[31] One possible fitness function is measured by the ratio between the number of accesses (hits) to an individual (document instance) over the number of accesses to all individuals of the population. Such a function however, is questionable whether the number of user accesses is sufficiently representative of user preferences.

is evaluated against DBSE in terms of its (*LifeWeb's*) capacity to (i) observe DBSE invariants and rules (see Sections 4.1.1.1 and 4.1.1.2), (ii) support DBSE evolution operations (see Section 4.1.1.3), and (iii) propagate changes (see Section 4.1.2).[32]

8.8.1 Observation of Database Schema Evolution invariants and rules

As explained earlier in various places (Sections 6.4.2.1.3, 6.4.2.4.4, 8.2, 8.2.5, 8.2.6, 8.3, 8.5), *LifeWeb* does not allow any change to the internal definition of an existing class,[33] and a schematic change in *LifeWeb* (or a *Life Design* system) happens only on a component (class) basis. In other words, a schematic change is carried out only through the addition of a new class or removal of an existing class (possibly with its "subordinate" classes – see Section 6.4.1.1.4 about the "subordinate" relationship), via the generalisation link. Because a new class must have been derived from an existing class to be eligible for addition (see Sections 6.4.2.1.3, 8.2.5, 8.3 and 8.5), and an existing class must have been free of all dependency to be eligible for removal (see Sections 6.4.2.4.3, 8.2.6, 8.3 and 8.5), both these operations do not introduce any conflicts to the evolving system. This means, DBSE invariants and rules (as listed in Sections 4.1.1.1 and 4.1.1.2) are either not relevant or naturally observed in *LifeWeb*.[34]

8.8.2 Support for Database Schema Evolution evolution operations

According to Banerjee et al. [BAN87], DBSE evolution operations are categorised into three groups: (i) Changes to the internal class definition, such as adding, removing or modifying attributes or methods; (ii) Changes to the generalisation/specialisation relationships between classes; and (iii) Changes to the class as a whole, such as adding or removing an entire class (see Section 4.1.1.3).

As explained earlier in several places (Sections 6.4.2.1.3, 6.4.2.4.4, 8.2, 8.2.5, 8.2.6, 8.3, and 8.5), only changes in the third category may happen in *LifeWeb*. Such changes are also the means to allow for changes in the first and second categories to happen. That is, the addition, removal or

[32] This evaluation is applicable also to *Life Design*.

[33] The internal definition of a class here is understood in the entirety of the class, which consists of its attributes, methods, including inherited ones, and its "subordinate" classes (see Section 6.4.1.1.4 about the "subordinate" relationship).

[34] To be more precise, DBSE invariants and rules are ensured (where relevant) by the development environment in which *LifeWeb* is implemented.

modification of a feature (attribute, method or link) in an existing class means that a new class (with the desired feature) must be developed and introduced into the system (through an appropriate generalisation relationship). For instance, in the example described in Section 8.3, the new class C_1^B is developed in order to remove some features in C_5^A (Figure 8-3 (d)). C_1^B may also be an implementation to remove the superclass C_5^A from C_6^A. The restriction that all schematic changes must happen through those in the third category allows *LifeWeb* to ensure systematic evolution on a component basis (see Sections 6.4.2.1.3, 6.4.2.4.4, and 8.8.1).

DBSE also defines some evolution operations that do not change the definition of a class or a class system, but only an attribute value that is common in the whole class (that is, global to all objects of that class). For instance, operations 1.1.6 and 1.1.7 (see Section 4.1.1.3) allow for changes to a default and shared attribute value, respectively.[35] Since these are changes to pure data values only, they are not required in *LifeWeb* to be implemented by a first-category operation (an introduction of a new class in this case). However, as they are global changes (relative to the objects defined by the given class), they must start locally first at individual objects at the concrete level (document instances), and propagate up to the genetic (schematic) level, in the same way any other evolutionary change is developed in *LifeWeb* (see Section 6.4.2.4.4).

8.8.3 Change propagation

LifeWeb and DBSE propagate changes in opposite directions. On the one hand, *LifeWeb* evolution starts from individual instances at the concrete level, and propagates up to higher meta-level of genes (bottom-up evolution). On the other hand, evolution in a database system starts at the schematic (genetic) level and propagate down to instances (top-down evolution) (see Section 4.1.2). This difference is instrumental in providing *LifeWeb* with a significant advantage over DBSE. *LifeWeb* evolution is a gradual, incremental and recursive process, where changes happen and propagate from individuals to groups, instances to schemata, and local to global. In this way, local and incremental changes automatically, gradually and recursively reshape the system to represent the new state. Bottom-up evolution frees *LifeWeb* from the complication and overhead that incurs in DBSE due to the need to propagate schematic and global changes down to affected instances (see Section 4.1). In the long-terms, it allows for systematic evolution, and prevents adhoc, "quick and dirty" changes to existing features that have caused many serious problems for the Web (see Sections 2.2.1.2, 2.2.1.3, 2.2.1.4, 2.2.1.5).

[35] A shared attribute is also commonly referred to as a *class variable* in the programming environment.

In brief, *LifeWeb* can support all DBSE evolution operations, with constraints that allow *LifeWeb* to evolve systematically and strictly on a component basis. Changes are propagated in a bottom-up fashion across levels of meta-modelling, which enables a smooth evolution of the whole system and eliminates overhead costs for maintaining data and schema consistency that is required in DBSE top-down evolution.

8.9 Conclusion

Life Design and the evolutionary model in *LifeWeb* propose an evolvable Web system in which new document types (DTDs) are derived from existing ones in an object-oriented manner. This means existing DTDs and their related software and tools can be reused, and the system is maintainable and manageable both statically and dynamically in its course of evolution. (See also Sections 9.4.2 and 9.4.3 for implementation of automatic evolution.) Evolution is enabled by changes introduced to individual *Document* objects (by Web developers and authors), driven by user needs, and incrementally accumulated and propagated through successive levels of meta-genes, resulting in changes at the schema or higher meta-modelling levels. System interoperability is maintained through the architecture of recursive levels of meta-genes (or meta-modelling). System scalability is generally maintained through the two processes addition and removal, which have cancelling effect to each other. This evolutionary model makes *LifeWeb* an *open* system in the sense that its evolution accepts (and calls for) collaborative work among public Web developers/authors and users worldwide (the external environment) to introduce changes and derive new DTDs. (This is made possible by the use of only open standards and technologies in the system implementation as will be described in Chapter 9.) As an analogy to the Darwinian evolutionary theory, where:

> The evolutionary process, taking place in each individual living organism, is enabled by its internal gene mutation and modification, driven by external forces of natural selection, and realised by the population's adaptation capabilities (Section 6.2.6),

it can be similarly said for *LifeWeb* that:

> The evolutionary process in *LifeWeb*, taking place in each individual *Document*, is enabled by its internal specifications modification, driven by external forces of user requirements, and realised by *LifeWeb* adaptation capabilities.

Chapter 9 LifeWeb Implementation

This chapter describes the implementation of *LifeWeb*. Two major decisions have to be made: the programming environment in which *LifeWeb* is implemented, and the notation in which the *LifeWeb* data model is expressed. These are explained in Sections 9.1 and 9.2. Section 9.3 shows the class mapping between the *LifeWeb* data model (see Section 7.1) and the *LifeWeb* implementation. The system architecture is described in Section 9.4 in three levels: basic, customisable and evolvable. Sections 9.5 and 9.6 explain how *LifeWeb* can enhance referential integrity and help in resource discovery. Finally, Section 9.7 describes *LifeWebManager*, a *LifeWeb* application for creating, managing, and maintaining *LifeWeb* documents.

9.1 Programming environment

We have chosen to implement *LifeWeb* as an extension to a Web server. There are several alternative technologies to implement such an extension, such as JavaServlet^{tm} [HUN98], and Common Gateway Interface (CGI) Script [MCC95]. We have chosen JavaServlet^{tm}, as it is known to outweigh CGI Script in terms of performance and flexibility, and is supported by the richly featured Java^{tm} technology. The *LifeWeb* data model must also be expressed in a well-defined language that should be well integrated with the Web, and the implementation must include a parser for the chosen language. For this purpose, the Extensible Markup Language (XML) (see Section 3.2.2) has been chosen, and an XML parser incorporated in our implementation. The system implementation thus makes use of the following development tools:

- Java Development Kit (JDK) 1.2: this JDK is distributed free of charge from Sun Pty. Ltd.. *LifeWeb* is implemented completely in Java$^{\text{tm}}$.
- HotServ: a Web server written by Scott Milton, a PhD candidate at the School of Computer Science and Software Engineering, Monash University. HotServ is written in Java and supports JavaServlet$^{\text{tm}}$. *LifeWeb* servlets can thus be developed and interface between the HotServ Web server and the *LifeWeb* engine. HotServ is released free of charge by the courtesy of its author.
- XML4J: An XML (see Section 3.2.2) parser, written in Java, that implements the XML specifications. This parser is developed and distributed free of charge by IBM AlphaWorks Pty. Ltd. [IBM98]. XML4J also supports the Document Object Model (DOM) (see Section 3.2.3). Figure 9-1 shows the class hierarchy of the relevant XML4J classes. (In this figure, for brevity, all the classes shown without a package name are the XML4J classes in the package "com.ibm.xml.parser". For example, the full name of class "Child" is "com.ibm.xml.parser.Child".) The *LifeWeb* engine is built as an extension of XML4J.

The system is developed and tested on a Pentium PC running Linux RedHat version 5.2.

```
•  java.lang.Object
   o  Child (implements org.w3c.Node)
      ▪  Parent
         •  DTD (implements org.w3c.DocumentType)
         •  TXAttribute (implements org.w3c.Attr)
         •  TXDocument (implements org.w3c.Document)
         •  TXElement (implements org.w3c.Element)
      ▪  TXCharacterData
         •  TXText (implements org.w3c.Text)
   o  Parser
```

Figure 9-1: Class hierarchy for XML4J (simplified)

9.2 Notation - LifeWeb XML

LifeWeb is a data model with multiple levels of meta-modelling, and each level has to be expressed in a well-defined notation (see Sections 6.4.1.2 and 6.4.2.2). The first two levels, concrete and genetic, are defined in Sections 7.1 and 7.2. In this section the notation for these first two levels is presented. (Detailed design and notation for higher meta-levels are not within the scope of this thesis).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE LifeWeb System="http://lifeweb.org/lifeweb.dtd">
<LifeWeb "id=http://lifeweb.org/lifewebbook.xml"
behaviour="LifeWeb">

        <table id="Chap1Pres" row="1" col="2" behaviour="LWTable"/>
        <book id="MyBook" heading="LifeWeb and Life Design"
        behaviour="LWBook">
                <struct id="Chap1" heading="INTRODUCTION"
                present="Chap1Pres" behaviour="LWStruct">
                        <text id="Chapter1Txt"
                        src ="../mybook/data.htm#intro"
                        behaviour="LWText"/>
                </struct>
                <struct id="Chap2" heading="LIFE DESIGN"
                behaviour="LWStruct>
                        <text id="Chap2Txt"
                        src="../mybook/data.htm#lifedesign"
                        behaviour="LWStruct"/>
                </struct>
        </book>

</LifeWeb>
```

Figure 9-2 : A simple LifeWeb document represented in XML

At the concrete level, a *LifeWeb* document essentially holds information describing its own structure, that is, meta-data. It can thus be expressed in a knowledge description language that supports meta-data and nesting structure (to satisfy *Life Design*'s structural constraint), such as Telos [MYL90, LOU92] or XML (Section 3.2.2). XML is chosen because it is an emerging Web standard that shows large potentials and is increasingly accepted [BOS97, CON97, TRE98, USD98].

At the genetic level, since XML has been chosen to represent document instances at the concrete level, it must also be used to describe document schemata, according to the XML specifications.

Thus, at the concrete level, a *LifeWeb* object is generally represented as an XML element, and a LifeWeb document an XML document. At the genetic level, a *LifeWeb* class generally corresponds to an element type, and the *LifeWeb* schema (LWS) a Document Type Definition (DTD). As can be seen in earlier examples (Section 7.2.2), *LifeWeb* maps fairly well into XML's syntactic model. (For convenience, the first example in Section 7.2.2 is reproduced in Figure 9-2). There are three exceptions to this mapping, and a special support for inheritance:

The *Behaviour* class

A *Behaviour* object has no corresponding XML element and the *Behaviour* class no corresponding element type. For instance, in Figure 9-2, the element <book id="MyBook"> has an association with the *Behaviour* object *LWBook* (represented by the attribute *behaviour="LWBook")*, but *LWBook* is not defined as an element in the given XML document. This is because *LWBook* really represents the

API implementation of the class *Book* (see Section 7.1), which cannot be represented in a (declarative) document (such as an XML document in this case). It is noted that objects of other classes, except for *Schema* (see below), are all represented as XML elements in the given document. For instance, the object *Chap1Pres:Table* (Figure 7-3) is represented as the XML element <table id="Chap1Pres"> (Figure 9-2).

The *Schema* class

An individual XML element often does not contain an explicit association to its *Schema* object as prescribed in the *LifeWeb* data model (Figure 7-1). In addition, a *Schema* object does not have a corresponding XML element, nor does the *Schema* class have a corresponding element type, and a *Schema* object is usually not defined within the XML document where it (the *Schema* object) is used.

```
<e xmlns:chem='http://chemistry.org/schema.dtd'>
      <chem:molecule>C6H6</chem:molecule>
</e>
```

Figure 9-3: XML namespace

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE greeting [
  <!ELEMENT greeting (#PCDATA)>
]>
<greeting>Hello, world!</greeting>
```

Figure 9-4: Internal DTD

The first exception exists because an XML document contains the <!DOCTYPE> construct that holds a reference to its main DTD, that is, its main system schema, or main class genome.[36] This gives every element in the document access to the document's main DTD (system schema or class genome), as well as its own element type definition (*Schema* object or class gene), without the need to explicitly associate each element to its own element type definition. For instance, in Figure 9-2, the element <book id="MyBook"> can access the document's DTD via the reference `http://lifeweb.org/lifeweb.dtd`, and its own element type definition via the reference `http://lifeweb.org/lifeweb.dtd#book`. In the case of a hybrid XML element, however, the element is explicitly associated with (qualified by) the DTD that defines its type via the *namespace* mechanism (Section 3.2.2). For instance, the element <molecule> in Figure 9-3 is associated with the DTD `http://chemistry.org/schema.dtd` via the prefix *chem*.

[36] XML also permits a document without a DTD, called a *well-formed* document. In this thesis we discuss only *valid* XML documents, that is, documents *with* their own DTDs.

The second exception exists because in XML, a *Schema* object (class gene) is an element type (and the *Schema* class is really a meta-class, which cannot be represented as an element type), which is usually declared in a DTD external to the XML document where it (the element type) is used (see Sections 7.1 and 7.2.2). Besides, even when a *Schema* object is declared within the given XML document, as in the case of an internal DTD (Section 3.2.2), it must be represented as an element type definition, not an element (Figure 9-4).

The <LifeWeb> element

The <LifeWeb> element corresponds to the document's *root element* defined in XML, which indicates the type of the document being defined and encloses every other element in the document (see Section 3.2.2).[37] This root element does not correspond to any *LifeWeb* object defined in the *LifeWeb* data model (Figure 7-1), but may be considered semantically equivalent to the document's object genome, because it represents the set of all elements in the document.

It is noted that a *LifeWeb* object does not directly contain its material or presentational components; instead it references these. This enforces the complete separation of the structural aspect from other aspects of the document.

Superclass

It is noted that XML does not support inheritance. We provide for this by the use of a special attribute *superclass* known to the *LifeWeb* system. The value of this attribute is the name of the superclass. In the current implementation, this *superclass* attribute is simply used as a shorthand notation to include inherited attributes in the subclass instead of an explicit listing of them. Similar to conventional object-orientation, an inherited attribute may be re-defined at the subclass. For instance, in Figure 7-6, element type *struct* inherits all attributes of element type *doc*, which are *id*, *heading* and *behaviour*. The value of attribute *behaviour*, however, is re-defined in *struct*.

9.3 Class mapping

Except for *Behaviour*, which represents a class API implementation (and thus is not implemented as a class), each class comprising the *LifeWeb* core model (Figure 7-1) is implemented as an equivalent Java class in the *LifeWeb* prototype. Some classes in the extended model (Figure 7-2), and some extra

[37] Note that the root element in an XML document is not the root form in *Life Design*. In Figure 9-2, the root form is the element <book id="MyBook">.

classes required in our particular implementation are also implemented. There are five packages in the system: *model, service, processor, dbm* (database management) and *misc* (miscellaneous).

9.3.1 The *model* package

Branch	*LifeWeb* class	Java class
	LifeWebObject	LWObject
	N/A	LifeWeb
Document	Document	LWDoc
	Publication	LWPub
	StructuralComponent	LWStruct
	N/A	LWPageBreak
Material	Material	LWMaterial
	Text	LWText
	Image	LWImg
Presentation	Presentation	LWPresent
	Table	LWTable
	List	LWList
Hyperlink	Hyperlink	LWHLink
	Hypertext	LWHText
Evolution	Schema	LWS
	Evolutor	LWEvolutor
Service	Service	LWService
	N/A	LWParameter

Table 9-1 : Implemented LifeWeb classes

The *model* package contains all implemented classes in the *LifeWeb* core and extended models, plus three extra classes: *LifeWeb*, *LWPagebreak* and *LWParameter*. The package can be subdivided into four branches: *Document, Material, Presentation, Hyperlink, Evolution,* and *Service* (see Table 9-1).

All classes in this package, except for *LWS* (*Schema*), are subclasses of the *LWObject* (*LifeWebObject*) class, which, in turn, extends the XML4J *TXElement* class that implements the Document Object Model (DOM) *Element* interface (see Figure 9-1, Figure 9-5, and also Section 3.2.3). *LifeWeb Schema* is a DTD (see Section 9.2), therefore it is implemented as a subclass of the XML4J *DTD* class (which is an implementation of the DOM *DocumentType* interface).

```
•  java.lang.Object
   o  com.ibm.xml.parser.Child
      ▪  com.ibm.xml.parser.Parent
         •  com.ibm.xml.parser.DTD
            o  au.edu.monash.sd.lifeweb.model.LWS
         •  com.ibm.xml.parser.TXDocument
            o  au.edu.monash.sd.lifeweb.model.LifeWeb
         •  com.ibm.xml.parser.TXElement
            o  au.edu.monash.sd.lifeweb.model.LWObject
               ▪  au.edu.monash.sd.lifeweb.model.LWDoc
                  •  au.edu.monash.sd.lifeweb.model.LWPub
                  •  au.edu.monash.sd.lifeweb.model.LWStruct
               ▪  au.edu.monash.sd.lifeweb.model.LWMaterial
                  •  au.edu.monash.sd.lifeweb.model.LWText
                  •  au.edu.monash.sd.lifeweb.model.LWImg
               ▪  au.edu.monash.sd.lifeweb.model.LWPresent
                  •  au.edu.monash.sd.lifeweb.model.LWTable
                  •  au.edu.monash.sd.lifeweb.model.LWList
               ▪  au.edu.monash.sd.lifeweb.model.LWHLink
                  •  au.edu.monash.sd.lifeweb.model.LWHText
               ▪  au.edu.monash.sd.lifeweb.model.LWEvolutor
               ▪  au.edu.monash.sd.lifeweb.model.LWService
               ▪  au.edu.monash.sd.lifeweb.model.LWPageBreak
               ▪  au.edu.monash.sd.lifeweb.model.LWParameter
   o  com.ibm.xml.parser.Parser
      ▪  au.edu.monash.sd.lifeweb.processor.LWParser
   o  au.com.live.httpd.HttpServlet
      ▪  au.edu.monash.sd.lifeweb.processor.LifeWebServlet
         •  au.edu.monash.sd.lifeweb.processor.CustomServlet
   o  au.edu.monash.sd.lifeweb.dbm.DBLink
   o  java.util.Hashtable
         •  au.edu.monash.sd.lifeweb.dbm.DBLinkTable
   o  au.edu.monash.sd.lifeweb.service.Service
      ▪  au.edu.monash.sd.lifeweb.service.ComponentSelector
      ▪  au.edu.monash.sd.lifeweb.service.TableFormater.
```

Figure 9-5: Class hierarchy of LifeWeb (simplified)

The *LifeWeb* class is derived from the XML4J *TXDocument* class (Figure 9-5), which is an implementation of the DOM *Document* interface (see Section 3.2.3). As explained earlier (Section 9.2), this class is required to represent the whole of a *LifeWeb* document.

LWPageBreak is needed to provide control over the sequential (page continuation) links. Because an object gene (*Document* object) is expected to be compact, it is possible for it to capture a large structure, potentially the whole Web site. The page break element <pb> gives the author the means to define physical partitions over the entire *Document* object, so that data can be rendered in a reasonable amount at a time.

LWParameter is needed to handle the <parameter> elements, which may be required by external services (<service> elements) (see Section 9.4.2).

9.3.2 The *service* package

The *service* package contains an abstract class, *Service*, from which external services must be derived. For testing purposes, two such external services are implemented and included in this package. They are: *ComponentSelector*, to select or deselect a structural component; and *TableFormatter*, to put a structural component into a table format.

9.3.3 The *processor* package

The processor package contains three classes: *LWParser*, *LifeWebServlet* and *CustomServlet*. *LWParser* is derived from XML4J *Parser* (Figure 9-5), and is needed to parse an XML document into a *LifeWeb* document of *LifeWeb* elements (see Section 9.4.1 below). *LifeWebServlet* and *CustomServlet* are derived from *HttpServlet* (defined by the JavaServlet^{tm} technology), and are needed to communicate between the Web server and the *LifeWeb* engine (see Section 9.4.1 below).

9.3.4 The *dbm* package

The *dbm* package, which contains a class *DBLinkTable* and an interface *DBLink*, provides database support to implement a *linkbase* for maintaining referential integrity (see also Section 9.5.2).

9.3.5 The *misc* package

The *misc* package contains other classes that provide some utilities or implement some design patterns. These classes are considered implementation details and not listed in this thesis.

Figure 9-5 shows the class hierarchy of the relevant classes in the *LifeWeb* implementation.

9.4 System architecture

This section describes the architecture of the system in three levels: basic, customisable, and evolvable.

9.4.1 Basic level

The *LifeWeb* system is implemented as a set of Java classes and servlets. The *LifeWebServlet* connects between the Web server and the *LifeWeb* engine. A request from the client for a *LifeWeb* document received by the Web server is sent to the *LifeWebServlet*, which loads the appropriate document, and passes it on to the *LWParser* for necessary processing.

Conceptually the processing of *LifeWeb* elements is done in a manner similar to the well-established theory of translation scheme known from compiler construction [AHO88]. As has been explained before (Section 5.2.2), the document component hierarchy is very similar to a parse tree where the materials associated to a document component are the terminals (leaves) of such a tree. A document node itself manifests a production rule expressing the sequential composition of the document from its next lower-level components. As the parser visits a node while traversing the document tree, it fires off the functions that process that node. In this sense, these functions implement the "semantic rules" associated with the production being used at that node. In the context of document processing, these "semantic rules" are the "processing rules" for that document level.

In the current prototype these functions are implemented as a set of methods with the same name, *handleElement()*, defined in XMLAJ *ElementHandler* interface, for each *LWObject* class in the *LifeWeb* engine. All *LifeWeb* elements are processed by *ElementHandlers*. Additional *ElementHandlers* can also be registered to the *LWParser* before the document is read. Our system has three groups of *ElementHandlers*. The *ContentHandlers* are responsible for loading the contents and inserting hyperlinks into the document, the *StructureHandlers* for layout and ordering structural components; and the *PresentationHandlers*, which are registered to the *StructureHandlers*, for formatting structural components before they are returned to the *LifeWebServlet*. *LifeWeb* also automatically generates structural and navigational links (such as table of contents or index, forward and backward) based on the structural and page break elements in the document. The fully filled-in and formatted document is finally returned back to the servlet to be written back to the client. Figure 9-6 depicts the basic system architecture.

It is noted that as elements are processed in a certain order while the document is being parsed, it is necessary that associated elements be defined before they are being referenced. For example, *Presentation* elements must be defined before the associating *StructuralComponent* elements. For instance, in Figure 9-2, the <table id="Chap1Pres"> element is declared before the associating <struct id="Chap1"> element.

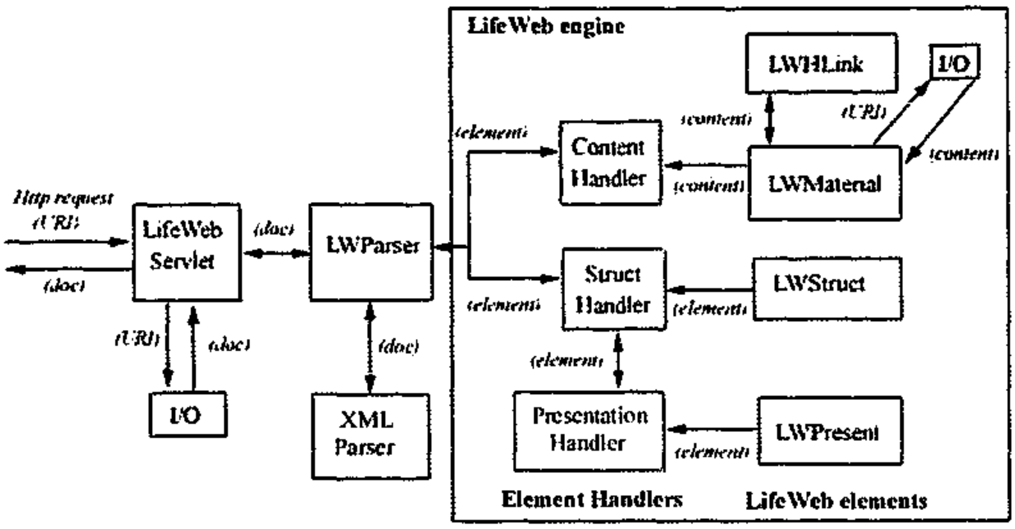


Figure 9-6 : LifeWeb system architecture (basic level)

9.4.2 Customisable level

Customisation has two flavours in *LifeWeb*, stateful and stateless.[38] Stateful customisations are saved in a client *Document* object (object gene) and will remain in successive invocations of the same document by the same client. Stateless customisations are not saved and will be destroyed when the HTTP connection to that client is closed. As write access is generally not allowed to client machines, in the current implementation, a directory in the file system of the server machine is reserved for storing client sites, subdivided into subdirectories, one for each client.

Customisation is handled on the basis of structural units (*Document* or *StructuralComponent* objects) in *LifeWeb*. According to Figure 7-1, such a unit can be associated with zero, one or several *Service* objects. These *Service* objects can be deployed to provide customisation. Customisation is handled in a way similar to the processing of a *LifeWeb* document at the basic level, where *Service* objects perform the role of *ElementHandlers*. Different from an *ElementHandler*, however, which defines a static method internal to a *LifeWeb* class (*handleElement()*), a *Service* object is an external and complete functional unit, dynamically created and bound to a *LifeWeb* object. This dynamic creation and binding is done via an intermediate <service> element internal to the *LifeWeb* engine (shown as the box with the label "LWService" in Figure 9-8). A <service> element is the XML representation of the (executable) corresponding *Service* object (shown as the box with the label "Service" in Figure 9-8). Such a <service> element and can be directly (hard) coded into the *LifeWeb Document* object or dynamically created by the *CustomServlet*. In the latter case, the document can be dynamically customised. The explanations below concern dynamic customisation, which also cover the static one.

CustomServlet

CustomServlet is a class derived from *LifeWebServlet* (Figure 9-6). It has extra features of being able to receive specific customisation requests from the user, create and save the corresponding <service> and other elements as appropriate (see below). These changes can be saved either to a specific client *Document* object (stateful customisation) or a temporary *Document* object (stateless customisation), as requested by the user. When a request for the document is received, the *CustomServlet* will pass the modified *Document* object (with the added <service> and other required elements) to the *LWParser* to be processed in the same fashion as described at the basic level (see Section 9.4.1). This time, however, all *LifeWeb* elements (*LWObject* objects), except for the <service> elements themselves, will be "served" by the <service> elements (representations of *Service* objects) first before they are handled by the *ElementHandlers*.

Service

```
<service id="TableFormatter93856721023"
       serviceClassName="TableFormatter" appliedClassName="struct">
      <parameter name="idrefs" value="table28909823412"/>
      <parameter name="view" value="user"/>
</service>
```

Figure 9-7 : A <service> element

Figure 9-7 shows the example of a <service> element. It encapsulates a reference to the implementation of the (external) service, that is, the full class name of the external service (*serviceClassName* attribute), the name of the element type onto which the service is applied (*appliedClassName* attribute), and a list of parameters required by each particular service (<parameter> elements). A <service> element is usually generated by the *CustomServlet* (in dynamic customisation), or may be manually created (in static customisation). In this particular example, this service will put in a table format all structural components (*appliedClassName="struct"*) whose attribute *view* is equal to *user*. The specifications for such a format is defined in the <table id="table28909823412"> element. This <table> element (not shown in Figure 9-7) is also dynamically generated by the *CustomServlet* (in dynamic customisation), or supplied (hard-coded) by the author (in static customisation).

[38] Also commonly referred to as *persistent* and *non-persistent*.

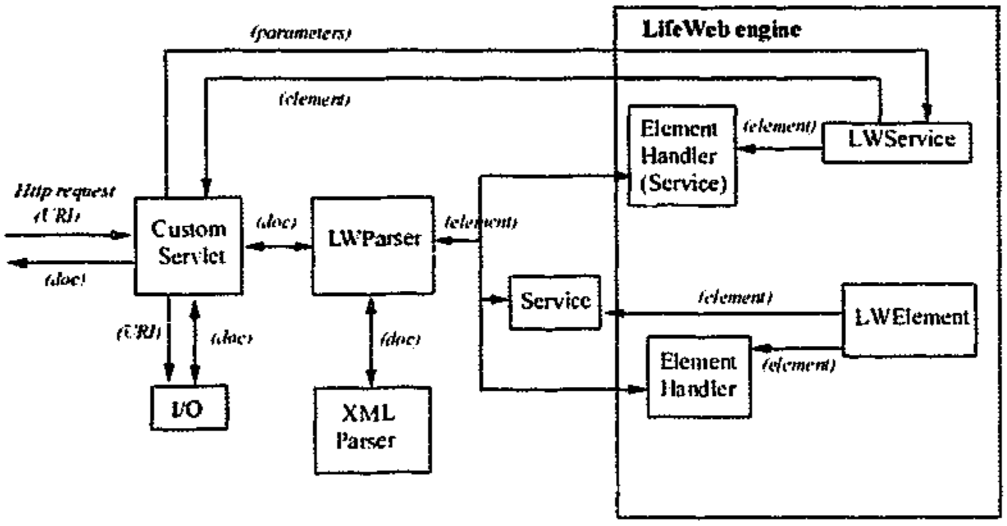


Figure 9-8: LifeWeb system architecture (customisable level)

During parsing, the *ElementHandler* for <service> elements dynamically instantiates *Service* objects (with the specified *serviceClassName*), creates the corresponding parameter lists for them, and registers them to the *LifeWeb Document* object. The *Service* object will then be invoked to serve the elements whose type matches the one captured in the *appliedClassName* attribute, when these elements are encountered while the document is being parsed. As *Service* objects must be created before they are used to serve other objects, their corresponding <service> elements must also be declared upfront in the *LifeWeb* document. Figure 9-8 depicts the architecture for the customisable system. In this figure all the *ElementHandlers* and *LifeWeb* elements, except for the one concerning the *LWService* element, have been simplified (shrunk down to two blocks), and the *LifeWebServlet* is not shown.

9.4.3 Evolvable level

Implementation of evolvability requires that the system has the capacity (i) to modify an evolving entity, that is, in the case of *LifeWeb*, a document instance (at the concrete level) or a LWS (at the genetic level); and (ii) to propagate changes from lower (concrete) to higher (genetic) meta-levels. This section explains the system's capacity in these aspects, and also how the evolutionary process can be automated.

9.4.3.1 Modifications to an evolving entity

To a document instance (concrete level)

Changes to a document, both client and server, is only required to be done declaratively to the document's object genome, that is, its XML source code. Such changes can be performed by the self-modification genetic functions (see Section 6.4.2.4.3), and written directly onto the XML code. These are simple changes, either to attribute values or links between objects.

To a LWS (genetic level)

A change to a LWS must be done to both the declarative object (that is, the LWS's object genome or DTD source code) and the *LifeWeb* runtime system. This means that (i) such a change must be appropriately declared in the LWS's source code; and (ii) the relevant class(es) must be integrated into the runtime system, preferably dynamically (without bringing the system down).

In our prototype system, the first requirement is carried out by the self-modification genetic functions (applicable to the *LWS* object) in the same way as it is done at the concrete level (see above). The second requirement is supported by the Java dynamic class instantiation [DEI98, JDK00] and *LifeWeb* dynamic class registration features. Using these features, information about *LWObject* classes is retrieved from the *LWS* object, which includes the full class names (captured in the *behaviour* attribute), and the corresponding element type names. This information is then used to dynamically instantiate the respective classes and register them to the *LWParser*. Objects of registered classes can then be processed by the *LWParser*, *ElementHandler*, and *Service* objects as appropriate. Thus, if the *LWS* has been declaratively changed (for instance, a new element type has been added), the change will be automatically accommodated into the runtime system (the corresponding class is instantiated and registered). Both of these changes can be done dynamically in our prototype system.

9.4.3.2 Change propagation

In a *LifeWeb* document, each element is a *life form* (Section 7.2), which can be made evolvable by associating it with an <evolutor> element. Changes are propagated across meta-levels by these <evolutor> elements, instances of the *LWEvolutor* class that implements an evolutionary engine (see Sections 6.4.2.4.4 and 7.1). Figure 9-9 shows the XML declaration of a *LifeWeb* element (<table id="table2342198745">) and its associated evolutor. In this example, the evolutor has two parts: the outer one (id="objectEvolutor") carries the *minSize* and *minRate* factors of the object evolutionary threshold (concrete level), and the inner one (id="geneticEvolutor") carries the factors of the genetic evolutionary threshold (Sections 8.1 and 8.5.1). In each of the thresholds, the positive values denote the thresholds for adding a link to, and the negative ones for removing a link from, another element (subform), respectively. This means, in Figure 9-9, if the element <table id="table2342198745"> is found *custom associated* with, for instance, a given structural component *S* in a server document *D* in 90 percent or more of the total client documents of *D*, and the total number of client documents of *D* is greater than or equal 200, then *S* will be permanently associated with the element <table

id="table2342198745"> in the server document *D*. Conversely, if <table id="table2342198745"> is found *custom de-associated* with *S* in 80 percent or more of the total client documents of *D*, and the total number of client documents of *D* is greater than or equal to 200, then *S* will be permanently *de-associated* with element <table id="table2342198745"> in the server document *D*.

```
<evolutor id="objectEvolutor"
          minSize="200 -200" minRate="0.9 -0.8">
      <evolutor id="geneticEvolutor"
                minSize="500 -500" minRate="1.0 -1.0"
      </evolutor>
</evolutor>

<table id="table2342198745" heading="Table 1"
      row="2" column="2" delimitor="struct"
      evolutor="objectEvolutor"
/>
```

Figure 9-9 : Evolutor element (represented in XML)

Since evolution in *LifeWeb* happens as a result of accumulated customisation changes, it always occurs in conjunction with customisation, that is, when a customisation operation is encountered, or a *Service* object is invoked. In the current prototype, therefore, a *Service* object (after making changes to the relevant components) directly invokes the *LWEvolutor* object (if present) of the added or removed component. The *LWEvolutor* object then inspects and detects if the same change has happened at other client *Document* objects for the same server *Document* object (or other server *Document* objects for the same *LWS*[39]) over the entire Web site. If an evolutionary threshold has been reached, the *LWEvolutor* object will register the change at the appropriate level (using the self-modification genetic functions described in Section 9.4.3.1). The architecture for the evolvable system is shown in Figure 9-10.

[39] The propagation of changes from the concrete to the genetic levels is not yet implemented in the current prototype.

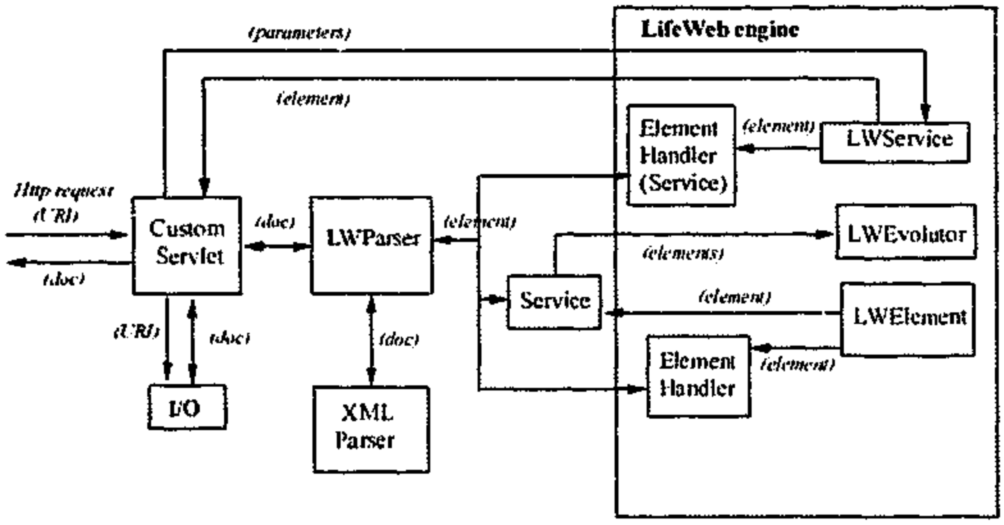


Figure 9-10 : LifeWeb system architecture (evolvable level)

9.5 Referential integrity

Referential integrity in the Web is a large issue and an independent field for study, and it is not within the scope of this thesis to carry out a comprehensive research in this area. *LifeWeb* thus does not aim to provide a complete solution to this problem, but seeks to alleviate the problem by its specific data model design. In particular, we are primarily concerned with referential integrity over a *local* system[40] only, that is, referential integrity in a system over which a *LifeWeb* server has control (at least read and write permissions). Referential integrity across *LifeWeb* servers is only dealt with in as far as data modelling is concerned. This section describes the design and implementation of the *LifeWeb Hyperlink* class, and explains how referential integrity can be improved in the system.

9.5.1 LifeWeb *Hyperlink* class

According to the *LifeWeb* data model (Figure 7-1) and its XML representation (Section 9.2), *LifeWeb* hyperlinks are stored within a document as separate XML elements, identified locally by a document-wide unique *id*, and globally by this *id* prefixed by the document's *id* (currently URL). *Hyperlink* is a *LifeWeb* class encapsulating the *source* from which the link eminates, a *destination* to which the link terminates, and a *scope* to specify how wide and/or how deep the link is defined.[41] Figure 9-11 shows

[40] The notion of *local system* here may not necessarily be restricted to one file system, but such a system must be fully under control of a given *LifeWeb* server.

[41] For simplicity, we include only attributes directly relevant to referential integrity.

an example of a *Hypertext* element <htext id="oo">, which is a child node of the *StructuralComponent* element <struct id="chapter1">.

The *source* attribute

The *source* attribute specifies the "hotspot" within the document upon which a user can "click and jump", for example a text span or an image (button). In the terms used by Hugh Davis [DAV98a], the *source* specifies the *content reference* part of the link. The *source*, together with the *scope* attribute, permits the use of *generic* hyperlink, which is explained below.[42]

```
<struct id="chapter1" heading="CHAPTER 1">
      <htext id="oo"
            source="object-oriented"
            destination="http://www.oo.org"
            scope="global"
      />
      <struct id="section1.1"
      ...
      </struct>
</struct>
```

Figure 9-11: A *Hypertext* <htext> element

The *destination* attribute

The *destination* attribute specifies the URL of the node to which the *source* is linked. In the terms used by Hugh Davis [DAV98a], the *destination* specifies the *node* part of the link.

The *scope* attribute

Scope allows hyperlinks to be *generic*, where one declaration of a *Hyperlink* element can be applied at one or more places, or recursively in the document tree. It specifies the exact place or places where the hotspots are in the document contents. In the current implementation the set of valid values for *scope* are:

- *"global"*: starting from the parent node of the (current) *Hyperlink* element and recursively down to all child nodes of that node, all occurrences of the *content reference* specified in the *source* will appear as hyperlinks. In the example shown in Figure 9-11, all occurrences of the text "object-oriented" in "chapter1", including all its subsections, shall be linked to the node specified in the *destination* attribute.

[42] It may be noted that theoretically the *source* can also be specified in terms of a region of an image (the equivalence of IMAGEMAP in HTML), possibly using some offsetting technique. For simplicity, this is currently not implemented in our prototype.

- *"local"*: similar to *global*, but without the recursion. A *local* hyperlink has effect only if the parent node of the link has its own contents, that is, it has at least one material node as its immediate child.

- *"self"*: the hyperlink is self-contained. The system will create a new hyperlink with the *content reference* specified in the *source* and link it to the node specified in the *destination*. By default the hyperlink will appear at the bottom of the contents of link's the parent node, although this may be customised by a presentation element. A *self* hyperlink is similar to a HTML hyperlink in that both the *content reference* and the *node* are tied together in one place (i.e. in the *Hyperlink* element), so that all the information needed to render the link is "right there" and there is no need to resolve the *content reference*. The difference, however, is that a *self* hyperlink is a globally identifiable object, and can be stored in a *link database* to support the maintenance of referential integrity (see Section 9.5.2 below).

- A specific number or a range of numbers: these numbers represent some offsetting method to specify the position(s) of the hotspot(s) within the whole (recursive) contents of the parent node of the link. In Figure 9-11 for example, if the *scope* is equal to "5", a hyperlink will be inserted at the fifth occurrence of the text "object-oriented" in "chapter1". If instead it is equal to "3-5", three hyperlinks will be inserted at the third, fourth and fifth occurrences of the text "object-oriented" in "chapter1".

In summary, the object-oriented design of *LifeWeb* allows hyperlinks to be encapsulated as separate objects, external to content data, and to make them bi-directional and generic. This approach improves upon the link integrity and embedded link problems in HTML (see Sections 2.2.1.2 and 2.2.2.5), although at the cost of added complexity due to the need to maintain content references (see Section 9.5.2.1.2), and to resolve and generate *Hyperlink* elements (see also Section 9.5.2.1.1). Another advantage of this approach is that authoring and maintenance costs can also be considerably reduced with the use of generic links.

9.5.2 Enhancing referential integrity

Referential integrity *within* a *LifeWeb* server can be ensured using the technique of *local link management*. Referential integrity *across LifeWeb* servers may be enhanced by the *dynamic link binding* technique. Both of these techniques are explained below.

9.5.2.1 Local link management

This technique operates only within a given *LifeWeb* server (local system). As pointed out by [DAV98a], there are two potential problems with hyperlinks:

- Dangling link problem: when the system fails to resolve the destination to a valid node. This problem occurs when the destination node can no longer be accessed (for example, because it has been moved)
- Content reference problem: when the system fails to resolve the source to the correct hotspot(s). This problem can happen only when the position of the hotspot is specified external to the data content (which is the case of LifeWeb and the systems mentioned above). When the data content is edited the reference may no longer be correct. In LifeWeb, this may happen if the *scope* attribute holds an offset value (see Section 9.5.1).

9.5.2.1.1 The dangling link problem

Because hyperlinks are encapsulated objects, separated from content data, it is possible to maintain them in a central link database (called *linkbase*). In the *LifeWeb* prototype, a link table is implemented to hold the identifiers (currently URLs) of the source and destination objects. This is supported by a built-in lightweight database. All insertions, deletions and movements of *LifeWeb* objects done under the control of the system (for instance, via a tool such as the *LifeWebManager* described in Section 9.7[43]) can be checked and synchronised to maintain link integrity.

By taking such an approach (external link), *LifeWeb* offers similar link service to some Open Hypermedia Systems (OHSs) such as HyperWave [HYP00], HyperDisco [WII96], Microcosm [DAV93], or by Ashman and Verbyla [ASH93]. It is perhaps potentially more efficient than these systems in this regard. This is because it stores hyperlink elements *within* the document, and the resolution and generation of these elements are done generally only once[44] when the document (or a page) is loaded, by methods *internal* to the *LifeWeb* document system itself (see Section 9.4.1). On the contrary, the other systems store hyperlinks in *external* linkbases only, and each link traversal

[43] This feature is not yet implemented in the current version of *LifeWebManager*, but already supported by the *LifeWeb* engine.

[44] This excepts when some change breaks a link (already resolved and generated) while the document is being viewed at the client site. If such a broken link is invoked, it will be necessary to regenerate the link from the original *LifeWeb* XML document. (It is noted that link integrity in the original document can be always maintained like records in a database).

incurs database transaction overheads [DAV98a]. In addition, the *LifeWeb* linkbase can be implemented as a simple lookup table (the actual objects are stored in the document itself), and there is no need for a commercial database support as in the other systems.

9.5.2.1.2 The content reference problem

Since hyperlink resolution is *natively* provided by *LifeWeb*, it is comprehensibly relatively simple to design a "link-aware tool" that is discussed by Hugh David [DAV98a] or implemented in HyperG [AND95]. Generally, this is an editing tool (such as the material processing application or text editor mentioned at the end of Section 9.7), which understands the *LifeWeb* syntax. *Hyperlink* elements can therefore be resolved, displayed and freely edited with the data contents while content references are maintained. For instance, if the data contents are changed in such a way that the change affects an offset value used in the *scope* attribute (see Section 9.5.1), the value can be updated automatically, or at least the author informed of the possible impact.[45] It is not within the scope of this thesis to design or implement this tool.

9.5.2.2 Dynamic link binding

The dangling link problem is most difficult to resolve across systems or servers. This is because updating all linked resources when a document is moved is notoriously expensive. As has been mentioned before (Section 9.5), it is not within the scope of this project to find a comprehensive solution to this problem. We do, however, provide *dynamic link binding* in our implementation as an optional way to alleviate it. Using this technique, hyperlinks can be optionally checked for validity before being inserted into the document contents and rendered at the client agent. This technique is possible and simple to implement in *LifeWeb* due to its specific object-oriented design, where hyperlinks are encapsulated objects (elements), which "know" how to resolve (and validate) themselves.

Dynamic link validation is a trade-off between link integrity and performance. In the current *LifeWeb* implementation, while a valid link generally does not take a noticeable amount of time to validate, an invalid one can cause considerable delay. The likelihood of invalid links during runtime, however, can be reduced by using a tool, such as the *LifeWebManager* described in Section 9.7, to check link integrity during design time. Another limitation is that if a Web resource is moved between two

[45] If a document is being edited by multiple authors, such a tool should also implement necessary features such as concurrency control. It is not within the scope of this thesis however, to discuss these features.

dynamic link validation operations, there will be a time interval in which the link is not valid. The likelihood of this happening however is comprehensibly very small.

9.6 Resource discovery

A *LifeWeb* document, or the document's object genome (see Sections 7.2 and 9.2), essentially contains only information describing itself, or "meta-data". It can thus be deployed by a user agent to facilitate resource discovery, similar to other meta-data schemes, such as Dublin Core (DC) or Resource Description Framework (RDF) (see Section 3.3). For instance, a *StructuralComponent* object has the attribute *heading*, which can be used in a search by keyword, or a *Publication* object has the attribute *author*, which can be used in a search by author. If necessary, a searchable entity, for instance, *author*, can be described in more details as shown in Figure 9-12 below:

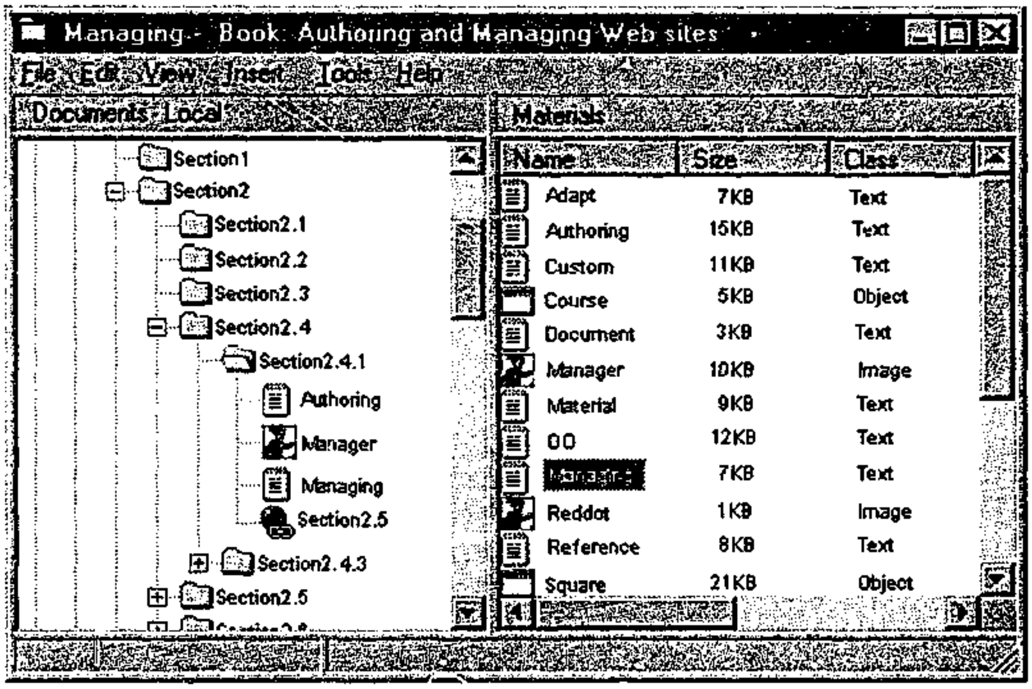
```
<pub   ID="ooBook"   title="Object-Oriented   Software   Design"
author="Rumbaugh, Blaha">
      .....
</pub>
<author ID="Rumbaugh" email="rumbaugh@omg.com"/>
<author ID="Blaha" email="blaha@omg.com"/>
```

Figure 9-12: Using attributes of LifeWeb elements as meta-data

The design of the *LifeWeb* DTD (data model) however, does not have a primary focus on facilitating resource discovery. For instance, the element type <author> in the above example can be defined another DTD, possibly one specialised in facilitating resource discovery, and imported into a *LifeWeb* document via the XML namespace mechanism (see Section 3.2.2). In addition, extensive work in this area (resource discovery) has been done (see Section 3.3), and most of it can be integrated into *LifeWeb* (for instance, RDF and DC).

9.7 LifeWebManager: A LifeWeb application

The design of *LifeWeb* allows for *LifeWeb* objects to be created and managed in a simple "drag-and-drop" paradigm. Based on this idea, we have designed a *LifeWeb* application called *LifeWebManager*, and with permission of the school, a prototype for it was developed at Monash University by a group of third-year computing students, as an Industrial Experience project. *LifeWebManager* allows the management and maintenance of a *LifeWeb* document on (only) three aspects: structure, content (material), and presentation.[46] Figure 9-13 shows a user interface of the application, which is built on top of the *LifeWeb* engine. It is designed to be similar to the Windows File Manager. The *LifeWebManager* has two views, *Material* and *Presentation*. In each view, the window always has two parts, one containing objects of the *Document* branch, and the other objects of the *Material* branch or *Presentation* branch (see Figure 5-1).



Name	Size	Class
Adapt	7KB	Text
Authoring	15KB	Text
Custom	11KB	Text
Course	5KB	Object
Document	3KB	Text
Manager	10KB	Image
Material	9KB	Text
OO	12KB	Text
Managing	7KB	Text
Reddot	1KB	Image
Reference	8KB	Text
Square	21KB	Object

Figure 9-13: ***LifeWebManager* - *Material*** **View**

In the *Material* View (Figure 9-13) for instance, the *Document* part shows the structure of (part of) a *Document* object, which is very similar to that of a DOS or UNIX file system. The *Material* part shows the "raw materials" (actual files) as they are physically located in the authoring file system. In the *Presentation* View (not shown), the *Presentation* part show the available *Presentation* objects, which can be associated with objects in the *Document* part. Generally, an object may be dragged and dropped onto another one to create an aggregation or association link (including hyperlink) between them. For instance, dragging and dropping a *Material* object onto a *Document* object creates an aggregation link; and dragging and dropping a *Document* object onto another one creates a hyperlink between the two. Object properties can be displayed and edited in a popup window, and hyperlinks can be validated by the *LifeWeb* engine. *LifeWebManager* generates an XML file based on the information visually presented.

[46] For simplicity, the management and maintenance of objects of other types such as *Service* and *Evolutor* are not covered in *LifeWebManager*.

We note that *LifeWebManager* supports the management and maintenance of *LifeWeb* documents on a component (object) basis only. In particular, the lowest level of granularity consists of objects whose classes are atomic *life forms* (see Sections 6.4.2.1.2 and 7.2) such as *Material* or *Presentation* (see Section 5.2.2).[47] It is unaware of, for instance, the "raw materials" contained inside a *Material* object, and editing to these "raw materials" must be passed to an appropriate processing application, such as a text editor. The design and development of such processing application and the processing of such "raw materials" are outside the scope of this thesis.

[47] By software engineering principles, modularisation and/or component-based processing are actually advantages of *LifeWeb*. That is, the "limitation" mentioned here is not considered a disadvantage of the system.

Chapter 10 Conclusion

The phenomenal growth of the Web in a very short time has resulted in many deployment problems that stem from inadequate system design (see Section 2.2). Such growth requires that there should be a data model for the Web that can capture its dynamic characteristics and cope with future changes. In Chapter 3, our review shows that the existing Web data models do not satisfy this requirement; in particular, the problems of adaptability and evolvability of the Web are not yet addressed. For this reason, we have constructed and implemented *LifeWeb*, a data model for the Web document system, which addresses a range of deployment issues of the Web, including adaptability and evolvability. *LifeWeb* does this by tackling these issues directly at the Web's fundamental design level. The deployment problems that *LifeWeb* has addressed are: data management and maintenance, document design and authoring, customisability, adaptability and evolvability, referential integrity and resource discovery. The associated design problems that have been undertaken are: data structure, document properties, lack of functional behaviour, statelessness, extensibility, evolvability, meta-data, and embedded hyperlinks.

Our approach to deal with the adaptability and evolvability issues is based on our review of system evolution (see Chapter 4), and our observations about biological life and its use in computer science as an analogy (see Chapter 6). We have chosen to use biological metaphors to establish ***Life Design***, an object-oriented design methodology for constructing evolvable systems. *LifeWeb* obtains its capacity to solve the adaptability and evolvability problems by implementing this *Life Design*. The wide range and the nature of the issues addressed by *LifeWeb* show the strength of the approach it has taken, which includes data modelling, object-orientation, and biological metaphors. These three technologies form the underlying philosophy for solving the research questions proposed in this thesis, and are what make *LifeWeb* achieve what it has.

10.1 Achievements of thesis

The achievements of this thesis are included in the successful application of data modelling, object-orientation and biological metaphors to establish *Life Design* and construct *LifeWeb*, a system that has the potential to solve or alleviate the Web's problems mentioned in the previous section, especially adaptability and evolvability, which have not been addressed.

In terms of data management and maintenance, and document design and authoring, a *LifeWeb* document can be designed, managed and maintained on the basis of any class that is associated with, or a part of, the *Document* class. In the current *LifeWeb* data model, these classes are: *StructuralComponent, Material, Presentation, Hyperlink, Schema, Behaviour, Service,* and *Evolutor* (including their subclasses).[48] In addition, the system can automatically generate navigational and structural links during runtime, effectively eliminating large maintenance costs (see Sections 7.1, 7.2.5.1, 9.4.1 and 9.7). The particular design features that have brought *LifeWeb* these advantages are data structure and document properties. Such features allow *LifeWeb* to capture the document structure and separate the different document properties found in traditional document systems (see Sections 5.2.1, 5.2.2 and 7.1).

In terms of customisability, a *LifeWeb* document can be customised (statically or dynamically) on the basis of any class that is associated with, or a part of, the *Document* class, where such a relationship (association and composition) defines a flexible link at the lower (concrete) level. In the current *LifeWeb* data model, these classes are: *StructuralComponent, Material, Presentation, Hyperlink,* and *Service* (including their subclasses) (see Sections 7.1, 7.2.5.1, 7.2.5.2, 8.2 and 9.4.2). A document element whose type is one of these classes can be added to or removed from its respective document. Such a change essentially alters the document's structure, content, presentation, navigational path, or the way it is customised. Because the separation of document properties makes a *LifeWeb* document

[48] The first four classes (structure, content, presentation and hyperlinks) represent the most common aspects (from an author's perspective) for document design, maintenance and management. Only these are implemented in our *LifeWeb* application, *LifeWebManager* (see Section 9.7).

compact, customised documents, which are client documents, can be wholly distributed at client sites. This allows for customisation information to be maintained (persistent) across Web sessions for individual users, and statelessness, one of the design issues of the Web, overcome (see Sections 5.2.2 and 7.2.3).

In terms of adaptability and evolvability, *LifeWeb* is the only Web data model we know that can adapt and evolve (including extend) itself systematically. *LifeWeb* evolution, designed after Darwinian evolution, occurs over populations of (customised) client documents mentioned above. Such evolution makes it possible for *LifeWeb* to adapt itself to user needs in a systematic, gradual and incremental manner through successive (low to high) meta-modelling levels. By applying object-orientation, such evolution means that in *LifeWeb*, new document types (DTDs) as well as their related tools and software can be derived from existing ones, which increases reusability, manageability, maintainability and extensibility. The Java implementation of *LifeWeb* permits automatic evolution of the system, where classes and DTDs can be dynamically added to or removed from the runtime system (see Section 9.4.3). The *LifeWeb* evolutionary model gives rise to a system which is scalable, interoperable and open (to accept public contributions for its development) (see Section 8.9).

In terms of referential integrity, our project only attempts to alleviate this problem and does not aim to offer a comprehensive solution. Within this scope, by eliminating embedded links (a directly related design problem) and encapsulating them in objects, separate from the document contents, a *LifeWeb* server can maintain link integrity within itself. Links across servers can be optionally validated and broken ones eliminated before document rendering (see Section 9.5).

In terms of resource discovery, a *LifeWeb* document contains information describing itself, which is essentially meta-data that can be deployed by other systems such as search engines or robots to help users locate the desired resources (see Section 9.6).

The system closest to *LifeWeb* that we are aware of is HyperWave, described in Section 3.2.1. The most important difference between the two systems is that HyperWave does not deal with adaptability and evolvability. Some characteristics of the Document Class introduced in the latest release of HyperWave[49] are similar to those of a class gene in *Life Design* (see Section 6.4.2.2). Here, the Document Class and a class gene both generate persistent instances, and can both serve as bases upon which new classes or class genes can be derived. The notion of class gene however, is fully developed into a system design methodology in our research, whilst the HyperWave's Document

[49] This release was in November 1999, which post-dated our publication about *LifeWeb* (1998) [NGU98b].

Class is simply a way to accommodate some special document types. *LifeWeb* manageability, maintainability and customisability, inherent from evolvability, are also carried out in the context of a richer and meaningfully evolving model. (For example, presentational management or customisation in HyperWave is fairly limited and cannot automatically evolve with changes in the data model.) Customisation in *LifeWeb* is fully supported in the sense that all server documents can be customised for each user by the distribution of object genes. (In contrast, HyperWave permits only one Home Collection per user.) (See Sections 3.2.1 and 7.2.3.)

In comparison to other Web data models such as the service-centred ones reviewed in Section 3.1, *LifeWeb* also has extra advantages. The most important one is inherent in its document-centred approach, in which *LifeWeb* can map its data model to that implied in the Web document system. Thus it is less affected when the Web infrastructure is changed, and can even directly address the design issues of the Web.

We claim that our technology improves on the Web manageability, maintainability and customisability over existing Web data models, and lays a foundation for Web adaptability and evolvability.

10.2 Limitations of thesis

A *LifeWeb* document can be manipulated (designed, managed, maintained and changed) on a component basis only. For example, the system is not aware of (and cannot manipulate) the character-level formatting of the "raw material" of a document, such as the font setting of one or a set of characters or words that does not by itself form a *LifeWeb* element. Although this places a restriction on the level of granularity that the system can handle, in terms of software engineering principles, component-based processing (or modularisation) is both an advantage and a required feature of quality software.

There are some implementation limitations. An important one is that in *LifeWeb* evolution, the propagation of change from the instance to the schema levels requires that an instance (for example, a document) be able to trigger a write operation to its schema (for example, the document's DTD). In addition, a schema must "know" about the population of its instances in order to assess whether an evolutionary threshold has been reached (for a particular evolving feature). With regard to the first issue, for simplicity, our current system can support change propagation from an instance to its schema if they both reside on the same machine, but not otherwise. One possible way to overcome this limitation is to use distributed object technology such as CORBA [BEN95, MOW97] or DCOM [RED97] to provide support for communication between objects in a distributed network. The second issue may be resolved by having all documents register themselves to their main DTDs, or by having a DTD log all documents that use it. This may involve drafting a policy effective in the whole Web and thus requires a global effort and a separate project.

Other implementation limitations include the direct (hard) coding of some features such as the numbering schemes or layout of structural components. These limitations however, are for simplicity, and can be easily overcome if required. For example, it is possible to develop a set of classes to handle different numbering schemes or layout. (These new classes can then also be added dynamically as part of the system evolutionary process.)

10.3 Future work

There is an abundant amount of work yet to be done for the future of our project. The most immediate need is perhaps to carry out an empirical study for *LifeWeb*, especially to test its evolvability in the real environment. Such a study requires first of all that the implementation limitations above be overcome. Some industrial and practical requirements, such as security and performance, may need to be considered and supported in the system prototype. Secondly, a *LifeWeb* core data model must be accepted by the Web community, or a group of pioneer Web developers, who will also contribute new element types, document types, or service types to its evolution. Experiments can then be carried out at a number of sample Web sites. This is a large project which requires not only technical soundness but also skills in organising, administrating, co-ordinating, and fund-raising.

Local evolution (that is, evolution in the domain of a specific Web site), which has not been considered in our thesis, is a meaningful topic for further investigation in *LifeWeb* (see Section 8.4.2). Because Web sites may serve different communities with differing needs, their evolutions may diverge towards different directions, and the interactions between them can be quite complicated When local evolution is considered, a document type or species may even reverse back to its original form. For example, a locally favoured element type that leads to the formation of a new document type in one community may be disgraced in another, so that the new document type, when imported to be used in the other community, will eventually regress. Local and global evolution has been an active research topic, especially in the field of artificial intelligence. Interested readers are directed to this literature, in particular, evolutionary computing and neural network [KUK00].

Another area that may eventually need to be considered is evolution at higher meta-modelling levels. If automatic evolution is required at these levels, we will need formal design and notations for the evolutionary entities (life forms, genes and genomes) there. Researchers in this area will find much input from, and possibly also interaction to, work done in the meta-modelling and meta-architecture literature [GIU96, JAR98, MAE88], especially UML [UML98] and XML-Schema [XSC00], the emerging standards in the object-orientation and Web fields.

The evolutionary threshold, which plays a critical role in the *LifeWeb* evolutionary process (see Section 8.5), is yet another interesting and necessary topic to be studied in the *LifeWeb* evolutionary model. Empirical researches may be done to work out appropriate threshold values for different evolving features, or categories of evolving features, and/or environments. Such values may even be adaptively computed during the evolutionary process, which perhaps necessitates an in-depth study in artificial intelligence, adaptive and evolutionary computing.

It is also highly desirable that work to be done in *LifeWeb* is targeted at satisfying the more sophisticated industrial requirements from the Web community, such as enriching the linking model, maintaining referential integrity across server boundaries, and improving the presentational model (which has been intentionally simplified in our project). Much work has been done in these areas in the Web and Open Hypermedia System (OHS) communities, especially in the XML suite. Interested readers are referred to the OHS literature [DAV93, DAV94, DAV98a, DEX94, ASH93, ASH94, VER94, NUN99] or the Web and XML literature [CSS98, XLK00, XPT00, XSL00]. Because XML is in the current development trend of the Web, and *LifeWeb* has used XML to represent its data model, integrating *LifeWeb* with the XML technology is both strongly desirable and highly feasible.

Bibliography

[AHO88] Aho, A.V., Sethi, R., and Ullman, J.D., 1988. *Compilers, principles, techniques, and tools.* Addison-Wesley Publishing Company.

[ALB83] Alberts, B., Bray, D., Lewis, J., Raff, M., Roberts, K., Watson, J.D., 1983. *Molecular biology of the cell.* New York & London: Garland Publishing, Inc.

[ALM95] Almasi, G. et al., 1995. Web* - A technology to make information available on the Web. *In: Proceedings of the IEEE Fourth Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, Berkeley Springs 20-22 April, 1995. Los Alamitos, CA.: IEEE Computer Society Press, 147-153.

[AND62] Andrews, D.H. and Kokes, R.J., 1962. *Fundamental Chemistry.* John Wiley & Sons Inc.

[AND95] Andrews, K., Kappe, F. and Maurer, H., 1995. Serving information to the Web with Hyper-G. *In: Proceedings of WWW5 International Conference*, Paris 6-10 May 1995. Elsevier Science. Also available at: http://www.igd.fhg.de/archive/1995_www95/proceedings/papers/105/hgw3.html [Accessed: 15 Oct 2000].

[APC00] The Apache Software Foundation, 2000. Apache Server [online]. Available at: http://www.apache.org/. [Accessed: 15 Oct 2000].

[ASH93] Ashman, H. and Verbyla, J., 1993. Externalizing hypermedia structures with the functional model of the link. *In: Proceedings of Seventeenth International Online Information Meeting*, London 7-9 Dec, 1993. Oxford: Learned Information, 291-301.

[ASH94] Ashman, H., Verbyla, J. and Cawley, T., 1994. Hypermedia management in large-scale information systems using the Functional Model of the link. *In: Proceedings of the 5th Australasian Database Conference*, New Zealand 17-18 Jan 1994. Singapore: Global Publications Services, 247-257.

[AYA79] Ayala, R.J. and Valentine, J.W., 1974. *Evolving.* Benjamin-Cummings.

[AYA84] Ayala, R.J., 1984. *Genetic variation and evolution.* Burlington, N.C.: Scientific Publications Department, Carolina Biological Supply Co.

[BÄC96] Bäck, T., 1996. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms.* University Press.

[BAN87] Banerjee, J., Kim, W., Kim, H-J., and Korth H.F., 1987. Semantics and implementation of schema evolution in object-oriented databases. *In: Proceedings of the ACM SIGMOD Annual Conference on Management of data*, San Francisco 27-29 May 1987. ACM Press, 311 – 322.

[BEN95] Ben-Natan, R., 1995. *Corba - A Guide to Common Object Request Broker Architecture.* McGraw-Hill.

[BEN97] Ben-Natan, R., 1997. *Objects on the Web: Designing, Building, and Deploying Object-Oriented Applications for the Web.* McGraw-Hill.

[BER89] Berners-Lee, T., 1989. *Information Management: A Proposal* [online]. Available at: http://w3.org/History/1989/proposal.html [Accessed: 15 Oct 2000].

[BER94] Berners-Lee, T., Cailliau, R., Nielsen, H.F. and Secret, A., August 1994. The World-Wide-Web. *Communication of the ACM,* 37 (8), ACM Press, 76-82.

[BER98] Berners-Lee, T. Evolvability [online]. *Keynote address in WWW7 International Conference,* Brisbane 15-17 Apr 1998. Available at: http://www.w3.org/Talks/1998/0415-Evolvability/slide1-1.htm. [Accessed: 15 Oct 2000].

[BOO94] Booch, G., 1994. *Object-oriented analysis and design with applications.* 2nd edition. Redwood City, CA.: Benjamin/Cummings Pub. Co.

[BOS97] Bosak, J., 1997. XML, Java and the Future of the Web. *World Wide Web Journal*, 2 (4), 219-229. Also available at http://sunsite.unc.edu/pub/sun-info/standards/xml/why/xmlapps.htm. [Accessed: 15 Oct 2000].

[BOU99] Bouvin, N.L., 1999. Unifying strategies for Web augmentation. *In: Proceedings of the 10th ACM Conference on Hypertext and hypermedia: returning to our diverse roots*, Darmstadt 21-25 Feb 1999. ACM Press, 91-100.

[BRA93] Brathwaite, K.S., 1993. *Object-oriented Database Design: Concepts and Application.* Academic Press, Inc.

[BRA99a] Bra, P.D., Houben, G-J., and Wu, H., 1999. AHAM: a Dexter-based reference model for adaptive hypermedia. *In: Proceedings of hypertext '99 on Hypertext and hypermedia*, Darmstadt 21-25 Feb 1999. ACM Press, 147-156.

[BRA99b] Bra, P.D., 1999. Design Issues in Adaptive Hypermedia Application Development. *In: Proceedings of Second Workshop on Adaptive Systems and User Modeling on the World Wide Web. In*: Brusilovsky, P., Bra. P.D. (eds.). *TUE Computer Science Report 99-07*, 20-39. Also available at: http://wwwis.win.tue.nl/asum99/debra/debra.html [Accessed: 15 Oct 2000].

[BUS45] Vannevar, B., Jul 1945. As We May Think. *The Atlantic Monthly*, 101-108.

[CLA98] Claypool, K., T., Jin, J., and Rundensteiner, E.A.,1998. SERF: schema evolution through an extensible, re-usable and flexible framework. *In: Proceedings of the 1998 ACM 7th International Conference on Information and Knowledge Management*, Washington 3-7 Nov, 1998. ACM Press, 314-321.

[COA91] Coad, P., and Yourdon, E., 1991. *Object-oriented Design*. London: Prentice-Hall International.

[COO00] Cookie Central, 2000 [online]. http://www.cookiecentral.com/. [Accessed: 15 Oct 2000].

[CON87] Conklin, J., 1987. Hypertext: an introduction and survey. *IEEE Computer* (Sep 1987), 17-41.

[CON93] Conradi, R., Fernström, C., Fuggetta, A., 1993. A conceptual framework for evolving software processes. *ACM SIGSOFT, Software Engineering Notes*, 18 (4) (Oct 1993), 26-35.

[CON97] Connolly, D., Khare, R., and Rifkin, A., 1997. The evolution of Web Documents: The Ascent of XML. *World Wide Web Journal Special Issue on XML*, 2 (4) (Fall 1997), 119-128.

[COR93] Corcos, A.F. and Monaghan, F.V., 1993. *Gregor Mendel's Experiments on plant hybrids : a guided study*. N.J. : Rutgers University Press.

[CSS98] W3C Recommendation, 1998. *Cascading Stylesheets, level 2 (CSS2)* [online]. Available at: http://www.w3.org/TR/REC-CSS2/. [Accessed: 15 Oct 2000].

[CUN93] Cunningham, D.J., Duffy, T. M., Knuth, R.A., 1993. The Textbook of The Future. *In:* McKnight, C., Dillon, A., Richardson, J., (eds.) *Hypertext: A Psychological Perspective*. New York : E. Horwood, 19-49

[DAR69] Darwin, C., 1859. *In:* Darwin, F. (ed.), 1969. *The Foundations of The Origin of Species: two essays written in 1942 and 1944*. Cambridge University Press.

[DAR77] Darwin, C., 1859. *In:* Burrow, J.W. (ed.), 1977. *The Origin Of Species By Means Of Natural Selection*. Peguin Books.

[DAV93] Davis, H.C., Hall, W., Pickering, A., and Wilkins, R., 1993. Microcosm: an open hypermedia system. *In: Proceedings of the conference on Human factors in computing systems*, Amsterdam 24-29 Apr 1993. ACM Press, 526.

[DAV94] Davis, H.C., Knight, S., and Hall, W., 1994. Light hypermedia link services: a study of third party application integration. *In: Proceedings of the 1994 ACM European conference on Hypermedia technology*, Edinburgh 19-23 Sep 1994, ACM Press, 41-50.

[DAV98a] Davis, H.C., 1998. Referential integrity of links in open hypermedia systems. *In: Proceedings of the ninth ACM conference on Hypertext and hypermedia: links, objects, time and space-structure in hypermedia systems*, Pittsburgh 20-24 Jun, ACM Press, 207 – 216.

[DAV98b] Paul, D., 1998. *The Fifth Miracle: The Search for the Origin of Life.* Allen Lane, The Peguin Press.

[DAW76] Dawkins, R., 1976. *The Selfish Gene.* 1989 ed. Oxford University Press.

[DAW95] Dawkins, R., 1995. *River out of Eden.* London: Weidenfeld & Nicolson.

[DEI98] Deitel & Deitel, 1998. *Java^{tm}: How to program.* 2nd edition. Prentice-Hall Incorp.

[DEN83] Denhardt, D.T., 1983. *Replication of DNA.* Burlington, N.C.: Scientific Publications Department, Carolina Biological Supply Co.

[DER99] DeRose, S.J. and Dam, A.V., 1999. Document Structure and Markup in the FRESS hypertext system. *Markup Languages,* 1(1), (Winter 1999), 7-46.

[DEX94] Halasz, F., Schwartz, M., 1994. The Dexter Hypertext Reference Model. Communication of the ACM, 37 (2) (Feb 1994), 30-39.

[DHT97] Microsoft. *Dynamic Hypertext Markup Language (DHTML)* [online]. Available at: http://msdn.microsoft.com/workshop/author/dhtml/dhtmlovw.asp [Accessed: 15 Oct 2000].

[DOM98] W3C Recommendation, 1998. *Document Object Model (DOM), level 1* [online]. Available at: http://www.w3.org/TR/REC-DOM-Level-1/. [Accessed: 15 Oct 2000].

[DUB00] Dublin Core Metadata, 2000 [online]. Available at: http://purl.oclc.org/metadata/dublin_core/. [Accessed: 15 Oct 2000].

[EKL98] Eklund, J., 1998. Adaptive Hypertext and Hypermedia Projects [online]. http://www.education.uts.edu.au/projects/ah/projects.html. [Accessed: 15 Oct 2000].

[ELM94] Elmasri, R., Navathe, S.B., 1994. *Fundamentals of Database Systems.* The Benjamin/Cummings Publishing Company, Inc.

[FTP85] RFC959, 1985. *File Transfer Protocol (FTP).*

[GEL97] Gellersen, H-W. et al., WebComposition: An Object-Oriented Support System for the Web Engineering LifeCycle. *In: WWW6 International Conference Proceeding,* Santa Clara 7-11 Apr 1997.

[GIU96] Giumale, C.A., and Kahn, H.J., 1996. A View of Information Modelling. In: Berge, J-M., Levia, O., Rouillard, J., (eds.). *Meta-Modeling: Performance and Information Modeling*. Kluwer Academic Publisher.

[GOP93] RFC1436, 1993. *The Internet Gopher Protocol (Gopher).*

[GVU95] Graphics, Visualization and Usability Center, 1995. *GVU's NSFNET Backbone Statistics* [online]. Available at: http://www.cc.gatech.edu/gvu/stats/NSF/merit.html. [Accessed: 15 Oct 2000].

[HAG94] Hagan, D.L., 1994. *An Object-Oriented Hypertext System Integrating Discrete-Event Simulation.* Master Thesis. Monash University.

[HAL95] Halpin, T., 1995. *Conceptual Schema & Relational Database Design.* 2nd ed. Australia: Prentice Hall.

[HEN97] Henderson-Sellers, B., 1997. Towards the formalization of relationships for object modelling. *In:* Mingins, C., Duke, R., Meyer, B. (eds). *Technology of Object-Oriented Languages and Systems: TOOLS 25*, Melbourne 24-28 Nov 1997. IEEE Computer Society, 267-284.

[HER94] Herwijnen E.V., 1994. *Practical SGML.* 2nd edition. Boston: Kluwer Academic Publishers.

[HNG00] W3C Working Draft. Hypertext Transfer Protocol – Next Generation (HTTP-NG) [online]. Available at: http://www.w3.org/Protocols/HTTP-NG/. [Accessed: 15 Oct 2000].

[HOL75] Holland, J.H., 1975. *Adaptation in natural and artificial systems.* The University of Michigan Press, Ann Arbor, MI.

[HTM92] W3C Recommendation, 1992. *HTML 1.0 Specification* [online]. Available at: http://www.w3.org/History/19921103-hypertext/hypertext/WWW/MarkUp/Tags.html. [Accessed: 15 Oct 2000].

[HTM98] W3C Recommendation, 1998. HTML 4.01 Specification [online]. Available at: http://www.w3.org/TR/REC-html40/. [Accessed: 15 Oct 2000].

[HTT97] Fielding, R. et al., Jan 1997. *RFC2616, Hypertext Transfer Protocol -- HTTP/1.1.*

[HUN98] Hunter, J., Crawford, W., 1998. *Java servlet programming.* 1st edition. CA.: O'Reilly, 1998.

[HYP00] HyperWave [online]. http://www.hyperwave.com. [Accessed: 15 Oct 2000].

[HYT97] ISO/IEC 10744:1997. *Information technology-Hypermedia/Time-based Structuring Language (HyTime)*, 2nd edition. Also available at: http://www.ornl.gov/sgml/wg8/docs/n1920/. [Accessed: 15 Oct 2000].

[IBM98] IBM AlphaWorks, 1998. *XML parser for Java* [online]. http://www.alphaworks.ibm.com. [Accessed: 15 Oct 2000].

[IDL97] Object Management Group, 1997. *OMG IDL Syntax and Semantics* [online]. Available at: http://www.omg.org. [Accessed: 15 Oct 2000].

[IKE99] Ikeji, A.C. and Fotouhi, F., 1999. An adaptive real-time Web search engine. *In: Proceedings of the second international workshop on Web information and data management.* ACM Press, 12 – 16.

[ILU91] Xerox Parc, 1991. *Inter-Language Unification (ILU)* [online]. Available at: ftp://ftp.parc.xerox.com/pub/ilu/ilu.html. [Accessed: 15 Oct 2000].

[ILU96] W3C Informational Draft 07-Mar-96. *The ILU Requester: Object Services in HTTP Servers* [online]. Available at: http://www.w3.org/TR/WD-ilu-requestor-960307. [Accessed: 15 Oct 2000].

[ING95] Ingham, D.B., Little, M.C., Caughey, S.J., Shrivastava, S.K., 1995. Bringing Object Oriented Technology to the Web. *World Wide Web Journal*, Issue 1, 89-105. *Also in: WWW4 International Conference Proceedings*, Boston 11-14 Dec 1995.

[ING96] Ingham, D.B., Little, M.C., Caughey, S.J., 1996. Fixing the "Broken-Link" Problem: The W3Objects Approach. *Computer Networks & ISDN Systems*, 28 (7-11), 1255-1268. *Also in: WWW5 International Conference Proceedings*, Paris 6-10 May 1996.

[ING97] Ingham, D.B., Little, M.C., Caughey, S.J., 1997. Supporting Highly Manageable Web Services. *Computer Networks and ISDN Systems*, 29 (8-13), 1405-1416. *Also in: WWW6 International Conference Proceedings*, Santa Clara, CA 7-11 Apr 1997.

[INT91] RFC1296, 1991. *Internet growth (1981-1991).*

[IRD90] ISO/IEC 10027:1990. *Information Technology – Information Resource Dictionary Systems (IRDS) Framework.* ISO/IEC International Standards.

[ISC00] Internet Software Consortium, 2000. *Internet Domain Survey* [online]. Available at: http://www.isc.org/ds/. [Accessed: 15 Oct 2000].

[JAC93] Jacobson, I. et al., 1993. *Object-oriented software engineering: a use case driven approach.* Rev. 4th print. Mass. : Addison-Wesley.

[JAR98] Jarke, M., Pohl, K., Weidenhaupt, K., Lyytinen, K., Marttiin, P., Tolvanen, J-P., and Papasoglou, M., 1998. Meta Modelling: A Formal Basis for Interoperability and Adaptability. *In:* Kramer, B., Papazoglou, M., Schmidt, H-W. (eds.). *Information System Interoperability.* Research Studies Press Ltd, John Wiley & Sons Co., 229-263.

[JDK00] Java Development Kit (JDK) 1.2 API documentation [online]. Sun Microsystems. Available at: http://java.sun.com/products/jdk/1.2/docs/api/index.html. [Accessed: 15 Oct 2000].

[JEF78] Jeffrey, C., 1978. *Biological Nomenclature.* 2nd ed. Crane: Russak.

[JON88] Jong, K., D., 1988. Learning with Genetic Algorithms: An Overview. Langley, P. et al. (eds), *Machine Learning.* 3 (1-4). The Netherlands: Kluwer Academic Publishers, 121-138.

[KAP94] Kappe, F., Andrews, K., Faschingbauer, J., Gaisbauer, M., Maurer, H., Pichler, M., and Schipflinger, J., 1994. Hyper-G: a new tool for distributed hypermedia. *In: Proceedings of the IASTED/ISMM International Conference. Distributed Multimedia Systems and Applications,* Anaheim 15-17 Aug 1994. IASTED/ISMM-ACTA Press, 209-214.

[KEM87] Kemper, A., Lockemann, P.C., and Wallrath, M., 1987. An object-oriented system for engineering applications. *In: Proceedings of the ACM SIGMOD Annual Conference on Management of data,* San Francisco 27-29 May 1987, 299-310.

[KHA98] Khare, R., Rifkin, A., 1998. The origin of (document) species. *Computer Networks and ISDN Systems,* 30 (1-7), 389-397.

[KOR80] Kornberg, A., 1980. *DNA Replication.* San Francisco: W.H. Freeman and Co.

[KOT84] Kotteman, J., Konsynsky, B., 1984. Information Systems Planning and Development: Strategic Postures and Methodologies. *Journal of Management Information Systems,* 1(2), 45-63.

[KOZ92] Koza, J. R., 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* The MIT Press.

[KUK00] Ku, K.W.C., Man Wai Mak; Wan-Chi Siu, 2000. A study of the Lamarckian evolution of recurrent neural networks. *IEEE Transactions on Evolutionary Computation.* 4 (1) (Apr 2000), 31-42.

[LAG96] Lagoze, C., Lynch, C.A. and Ron Daniel Jr., 1996. *The Warwick Framework: A Container Architecture for Aggregating Sets of Metadata* [online]. Available at: http://cs-tr.cs.cornell.edu:80/Dienst/UI/2.0/Describe/ncstrl.cornell/TR96-1593. [Accessed: 15 Oct 2000].

[LAM86] Lamport, L., 1986. *LATEX : a document preparation system.* Mass.: Addison-Wesley.

[LIN98] Lin, D., Loui, M., C., 1998. Taking the byte out of cookies: privacy, consent, and the Web. *In: Proceedings of the ethics and social impact component on Shaping policy in the information age,* Washington 10-12 May 1998. ACM Press, 39-51.

[LIU94] Liu, C-T., Chang, S-K., and Chrysanthis, P.K., 1994. Database schema evolution using EVER diagrams. *In: Proceedings of the workshop on Advanced Visual Interfaces AVI'94*, Bari 1-4 Jun 1994. ACM Press, 123-132.

[LOU92] Loucopoulos, P. and Zicari, R. (eds.), 1992. *Conceptual Modeling, Databases, and Case: An Integrated View of Information Systems Development*. John Wiley & Sons, Inc.

[MAE88] Maes P., Nardi D. (eds), 1988. *Meta-Level Architectures and Reflection*. Elsevier Science.

[MAN98] Manola, F., 1998. Towards a Richer Web Object Model [online]. *ACM SIGMOD Record*, 27 (1), Mar 1998. Also available at: http://www.acm.org/sigmod/record/issues/9803/index.html. [Accessed: 15 Oct 2000].

[MAN99] Manola, F., 1999. Technologies for a Web Object Model. *IEEE Internet Computing*, Jan/Feb 1999. Also available at: http://www.objs.com/survey/wom-ieee.htm. [Accessed: 15 Oct 2000].

[MCC95] McCool, R., 1995. *Common Gateway Interface* [online]. Available at: http://hoohoo.ncsa.uiuc.edu/cgi/. [Accessed: 15 Oct 2000].

[MER00] Merit Network, 2000 [online]. Available at: http://nic.merit.edu/. [Accessed: 15 Oct 2000].

[MER96] Merle, P., Gransart, C., Geib, J-M., 1996. CorbaWeb: A Generic Object Navigator [online]. *In: WWW5 International Conference Proceedings*, Paris 6-10 May 1996. Also available at: http://www5conf.inria.fr/fich_html/papers/P33/Overview.html [Accessed: 15 Oct 2000].

[MEY86] Meyrowitz, N., 1986. Intermedia: The architecture and construction of an object-oriented hypermedia system and applications framework. *In: Conference proceedings on Object-oriented programming systems, languages and applications*, Portland 29 Sep - 2 Oct 1986. ACM Press, 186-201.

[MOW97] Mowbray, T.J., Ruh, W. A., 1997. *Inside CORBA: Distributed object standards and application*. Addison-Wesley.

[MYL90] Mylopoulos, J., Borgida, A., Jarke, M. and Koubarakis, M., 1990. Telos: Representing Knowledge about Information Systems. *ACM Transactions on Information Systems*, 8 (4) (Oct 1990), 325-362.

[NAM99] W3C Recommendation, 14-Jan-1999 [online]. *Namespaces in XML*. Available at: http://www.w3.org/TR/1999/REC-xml-names-19990114/. [Accessed: 15 Oct 2000].

[NEL65a] Nelson, T. H., 1965. A File Structure for the Complex, the Changing and the Indeterminate. *In: Proceedings of the ACM 20th National Conference*, ACM Press.

[NEL65b] Nelscn, T.H., 1965. The Hypertext. *In: Proceedings of the World Documentation Federation.*

[NEL87] Nelson, T.H., 1987. *Computer Lib / Dream Machines.* Redmond, Wash. : Tempus Books of Microsoft Press.

[NEL90] Nelson, T.H., 1990. *Literary Machines 90.1: Report on project Xanadu.* Sausalito, California: Mindful Press.

[NEU58] von Neumann, J., 1958. *The computer and the brain.* New Haven: Yale University Press.

[NFS95] NFSNET Backbone data set, 1988-1995 [online]. Available at: ftp://nic.merit.edu/nsfnet/statistics/. [Accessed: 15 Oct 2000].

[NGU96] Nguyen, T-L., Newmarch, J., Baird, J., 1996. Network Performance – Impact on Networked Education Software. *In: Proceedings of Ausweb'96,* Brisbane 7-9 July, 1996. Southern Cross University, 281 – 289.

[NGU98a] Nguyen, T-L. Wu, X., Sayed, A.S.M., 1998. Object-Oriented Modeling of Multimedia Document. *Computer Networks and ISDN Systems: The International Journal of Computer and Telecommunications Networking,* 30 (1-7), 578-582. *Also in: WWW7 International Conference Proceedings,* Brisbane 14-18 Apr 1998.

[NGU98b] Nguyen, T-L., Wu, X., Sayed, A.S.M, 1998. LifeWeb: An Object-Oriented Model for the Web. *In:* Callaos, N., Yu, H. H., Garcia, A. (eds.). *Proceedings of SCI'98 & ISAS'98 Conference,* Orlando 12-16 July 1998. International Institute of Informatics and Systemics, 301-308.

[NGU99] Nguyen, T-L., Schmidt, H., 1999. Creating and Managing Documents with LifeWeb. *In: Proceedings of AusWeb99 Conference,* Southern Cross University, 230-241.

[NIE88] Nielson, J., 1988. Hypertext'87 Trip Report. *In: ACM SIGCHI Bulletin 19 (4) (April 1988),* 27-35.

[NNT86] RFC977, 1986. *Network News Transfer Protocol (NNTP).*

[NUN99] Nürnberg, P.J. and Ashman, H., 1999. What was the question? Reconciling open hypermedia and World Wide Web research. *In: Proceedings of the tenth ACM Conference on Hypertext and hypermedia: returning to our diverse roots.* ACM Press, 83-90.

[OMG00] Object Management Group, 2000 [online]. Available at: http://www.omg.org/. [Accessed: 15 Oct 2000].

[ORB00] IONA Technologies, 2000. *Orbix* [online]. Available at: http://www.orbix.com/. [Accessed: 15 Oct 2000].

[OUS94] Outsterhout, J., 1994. *Tcl and the Tk toolkit.* MA: Addison-Wesley.

[PER98] Perkowitz, M., Etzioni, O., 1998. Towards adaptive Web sites: Conceptual framework and case study. *In: Proceedings of WWW8 Conference*, Toronto 11-14 May 1999. Elsevier Science, 167-180.

[PET97] Peters, R.J. and TamerÖzsu, M., Mar 1997. An axiomatic model of dynamic schema evolution in objectbase systems. *ACM Trans. Database Syst.* 22 (1), 75-114.

[PIC97] W3C Recommendation, 1997. Platform for Internet Content Selection (PICS) [online]. Available at: http://www.w3.org/PICS/. [Accessed: 15 Oct 2000].

[RAG96] Raggett, D. (ed.), 1996. Inserting objects into HTML (W3C Report). *World-Wide-Web Journal*, 1 (3) (Summer 1996). Also available at: http://www.w3journal.com/3/s2.raggett.html. [Accessed: 15 Oct 2000].

[RAG97] Raggett, D., 1997. Client-Side scripting and HTML (W3C Working Draft). *World-Wide-Web Journal*, 2 (2) (Spring 1997). Also available at: http://www.w3journal.com/6/s2.raggett.html. [Accessed: 15 Oct 2000].

[RDF99] W3C Recommendation, 1999. *Resource Description and Framework (RDF), Model and Syntax* [online]. Available at: http://www.w3.org/TR/REC-rdf-syntax/. [Accessed: 15 Oct 2000].

[RED97] Redmond III, F. E., 1997. *DCOM: Microsoft Distributed Component Object Model.* IDG Books Worldwide, Inc.

[REE95] Rees, O., Edwards, N., Madsen, M., Beasley, M., McClenaghan, A., 1995. A Web of Distributed Objects. *In: WWW4 International Conference Proceedings*, Boston 11-14 Dec 1995. Also available at: http://www.w3.org/pub/Conferences/WWW4/Papers/85. [Accessed: 15 Oct 2000].

[REL94] Relihan, L., Cahill, T., Hinchey, M.G., 1994. Untangling the World-Wide-Web [online]. *In: Proceedings ACM SIGDOC'94*, Banff. New York: ACM Press, 17-24.

[RPC95] RFC1832, 1995. *Remote Procedural Call Protocol.*

[RUM91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy. F., and Orensen W., 1991. *Object-oriented Modeling And Design.* Prentice Hall.

[SCH91] Schmidt, H., and Omohundro, S., 1991. Clos, Eiffel and Sather: A Comparison. Technical Report. TR-91-047. ICSI, Berkeley.

[SCH95] Schrage, M., 1995. Revolutionary Evolutionist [online]. *In: Wired.* Archive 3.07 (Jul 1995). Available at: http://www.wired.com/wired/archive/3.07/dawkins_pr.htm. [Accessed: 15 Oct 2000].

[SGM86] ISO8879:1986. *Information processing – Text and office system – Standard Generalization Markup Language (SGML).* ISO International Standards.

[SGM00] Abortext, 2000. *An Abortext SGML Whitepaper* [online]. Available at: http://www.arbortext.com/Think_Tank/SGML_Resources/Getting_Started_with_SGML/getting_started_with_sgml.html. [Accessed: 15 Oct 2000].

[SMA96] Small, P., 1996. *Lingo Socery - The Magic of Lists, Objects and Intelligent Agents.* West Sussex: John Wiley & Sons.

[SMA98] Small, P., 1998. Magical A-Life Avatars - a new paradigm for the Internet. Connecticut: Manning Publications.

[SMI93] Smith, J.M., 1993. *The Theory of Evolution.* 3rd ed. Great Britain: University Cambridge Press.

[SMI82] Smith, A., 1982. The wealth of nations: Books 1-3. Harmondsworth: Penguin.

[SMI99] Smith, A., 1999. The wealth of nations: Books 4-5. London: Penguin.

[STO86] Stonebraker, M., and Arowe, L., 1986. The design of POSTGRESS. *In: Proceedings of the conference on Management of data,* Washington May 28-30 1986. ACM Press, 340-355.

[STO96] Stoutamire, D., and Omohundro, S., 1996. *Sather 1.1 Specification* [online]. Availble at: http://www.icsi.berkeley.edu/~sather/ [Accessed: 15 Oct 2000].

[TEI94] Sperberg-McQueen, C.M. and Burnard, L. (eds.), 1994. *Guidelines for Electronic Text Encoding and Interchange; TEI P3, Text Coding Initiative, Chicago, Oxford.* Association for Computers and the Humanities (ACH), Association for Computational Linguistics (ACL), and Association for Literary and Linguistic Computing (ALLC).

[TRE98] Treese, W., 1998. Putting it together: what's all the noise about XML? *In: netWorker,* 2 (5) (Dec 1998), ACM Press 27 – 29.

[UML98] UML Reference v1.3 [online]. Available at: http://www.rational.com/uml/. [Accessed: 15 Oct 2000].

[URI98] Internet Engineering Task Force (IETF) URI Working Group [online]. Available at: http://www.ics.uci.edu/pub/ietf/uri/. [Accessed: 15 Oct 2000].

[URL94] RFC1738, 1994. *Uniform Resource Locator (URL).*

[URN00] Internet Engineering Task Force (IETF) URN Working Group [online]. Available at: http://www.ietf.org/html.charters/urn-charter.html. [Accessed: 15 Oct 2000].

[USD98] Usdin, T. and Graham, T., 1998. *XML: not a silver bullet, but a great pipe wrench. In: StandardView*, 6 (3) (Sep 1998), ACM Press 125 – 132.

[VER94] Verbyla, J. and Ashman, H., 1994. A user-configurable hypermedia-based interface via the functional model of the link. *Hypermedia*, 6 (3) ,193-208.

[W3C] World-Wide-Web Consortium [online]. Available at: http://w3.org. [Accessed: 15 Oct 2000].

[WAI94] RFC1625, 1994. *Wide Area Information Servers (WAIS) over Z30.50-1988.*

[WAT83] Watson, J.D., Tooze, J., 1983. *The DNA story.* Freeman.

[WII96] Wiil, U.K., Leggett, J.J., 1996. The HyperDisco approach to open hypermedia systems. *In: Proceedings of the seventh ACM conference on HYPERTEXT '96* Bethesda March 16-20 1996, ACM Press, 140-148.

[WOB00] WebObjects [online]. Available at: http://www.apple.com/webobjects/whitepaper/index.html. [Accessed: 15 Oct 2000].

[XHT00] W3C Working Draft, 2000. Extensible Hypertext Markup Language [online]. Available at: http://www.w3.org/TR/xhtml1/. [Accessed: 15 Oct 2000].

[XLK00] W3C Working Draft, 2000. XML Linking Language [online]. Available at: http://www.w3.org/TR/xlink/. [Accessed: 15 Oct 2000].

[XML98] W3C Recommendation, 1998. Extensible Markup Language [online]. Available at: http://www.w3.org/TR/1998/REC-xml-19980210. [Accessed: 15 Oct 2000].

[XPA00] W3C Recommendation, 2000. XML Path Language [online]. Available at: http://www.w3.org/TR/xpath. [Accessed: 15 Oct 2000].

[XPT00] W3C Working Draft, 2000. XML Pointer Language [online]. Available at: http://www.w3.org/TR/xptr. [Accessed: 15 Oct 2000].

[XSC00] W3C Project, 2000. XML Schema [online]. Available at: http://www.oasis-open.org/cover/schemas.html. [Accessed: 15 Oct 2000].

[XSL00] W3C Working Draft, 2000. Extensible Stylesheet Language [online]. Available at: http://www.w3.org/TR/xsl/. [Accessed: 15 Oct 2000].

[XSLT00] W3C Recommendation, 2000. XSL Transformation [online]. Available at: http://www.w3.org/TR/xslt. [Accessed: 15 Oct 2000].

[YAN88] Yankelovich, N., Haan, B.J., Meyrowitz, N.K. and Drucker, S.M., 1988. Intermedia: The Concept and the Construction of a Seamless Information Environment. *IEEE Computer* (Jan 1988), 81-96.

[ZAK96] Zakon, R. H., 1996. *Internet Timeline v2.3a* (Feb 1996) [online]. Available at: http://info.isoc.org/guest/zakon/Internet. [Accessed: 15 Oct 2000].

Vita

Publications arising from this thesis include:

[NGU98a] Nguyen, T-L. Wu, X., Sayed, A.S.M., 1998. Object-Oriented Modeling of Multimedia Document. *Computer Networks and ISDN Systems: The International Journal of Computer and Telecommunications Networking*, 30 (1-7), 578-582. *Also in: WWW7 International Conference Proceedings*, Brisbane 14-18 Apr 1998.

[NGU98b] Nguyen, T-L., Wu, X., Sayed, A.S.M, 1998. LifeWeb: An Object-Oriented Model for the Web. *In:* Callaos, N., Yu, H. H., Garcia, A. (eds.). *Proceedings of SCI'98 & ISAS'98 Conference*, Orlando 12-16 July 1998. International Institute of Informatics and Systemics, 301-308.

[NGU99] Nguyen, T-L., Schmidt, H., 1999. Creating and Managing Documents with LifeWeb. *In: Proceedings of AusWeb99 Conference*, Southern Cross University, 230-241.

[NGU00a] Nguyen, T-L., 2000. LifeWeb and the Evolution of Document Types. *In: Proceedings of XML Asia Pacific Conference 2000*, Allette Systems Pty Ltd., to appear

[NGU00a] Nguyen, T-L., 2000. LifeWeb: An Evolvable Web. *Technical Report No. 2000/63*, School of Computer Science and Software Engineering, Monash University.

[NGU00b] Nguyen, T-L., 2000. LifeWeb: A solution to the broken-link problem. *Technical Report No. 2000/65*, School of Computer Science and Software Engineering, Monash University.

Software arising from this thesis include:

LifeWeb – A prototype system for the *LifeWeb* data model.

LifeWebManager – A tool for creating and managing *LifeWeb* documents.

Permanent address: School of Computer Science and Software Engineering
Monash University
Australia

This dissertation was typeset with Microsoft Word by the author.