A Views-Based Design Framework for Web Applications

by

David Bruce Brown

A thesis

presented to the University of Waterloo

in fulfilment of the

thesis requirement for the degree of

Master of Mathematics

in

Computer Science

Waterloo, Ontario, Canada, 2002

Abstract

Web application design is a broad term that refers to any aspect of designing a Web application, including designing Web interfaces to data. There are a number of commercial software tools available that employ various techniques for implementing Web access to databases. However, these techniques apply only to portions of Web application implementation and lack a common design paradigm. We believe that defining an approach to designing and implementing Web applications based on views, frameworks, and patterns allows us to extend an object-oriented design representation and develop reusable and extensible design solutions for a class of Web applications in which the concerns are separated.

The major contribution of this thesis is the development of a new approach to designing Web applications, producing the following visible contributions: a model for Web applications based upon a separation of concerns using views; a framework for system elements so that object-oriented design patterns can be used to build the application; a reusable design approach so that many Web applications can be built around the same framework; a coherent and organized framework representation using extended UML; and an implementation approach that can be implemented on a number of different platforms using a range of software and tools. In support of this solution, this thesis shows the design and implementation of a proof-of-concept Web application using this design technique.

We believe that the approach to Web application design promoted in this thesis has proven itself useful in a practical way for the case studies discussed herein and points the way to a wider range of design and implementation possibilities.

Acknowledgments

To Dr. Donald Cowan for his guidance and expertise, Dr. Paulo Alencar for his help, Dr. Grant Weddell for his patience, my parents Bruce and Claudia Brown for their support, my wife Annette for putting up with me, and Tasmin just for being there.

> "Writing a book is an adventure. To begin with it is a toy and an amusement. Then it becomes a mistress, then it becomes a master, then it becomes a tyrant. The last phase is just as you are about to be reconciled to your servitude, you kill the monster and fling him to the public."
>
> Sir Winston Churchill: on writing a book (or a thesis).

Table of Contents

Chapter 1: Introduction 1

1.1 Motivation 1

1.2 Problem Statement 2

1.3 Proposed Solution 2

1.3.1 Views 2

1.3.2 Frameworks and Design Patterns 5

1.3.3 Representation 6

1.4 Contributions 7

1.5 Outline of the Case Study 7

1.6 Related Work 10

1.7 Thesis Overview 13

Chapter 2: A Views-Based Framework 14

2.1 The Framework 14

2.2 The Repository 16

2.3 The Middle 19

2.4 The Clients 24

2.5 A Simple Framework Example 29

Chapter 3: A Sample Web Application Framework Design 32

3.1 Application Overview 32

3.2 The DCRIS Applications Design 33

3.3 The DCRIS Repository Package Design 35

3.4 The Middle Package Design 38

3.5 The Clients Package Design 40

3.6 Discussion 44

Chapter 4: Extending the Framework 46

4.1 'Other Expertise 46

4.2 Other Language Support 48

4.3 Discussion 50

Chapter 5: Reusing the Framework 52

5.1 The DCRIS Administrative Web Application 52

5.2 Further Framework Reuse 54

5.3 Discussion 57

Chapter 6: Implementing the Framework 58

6.1 The Repository Package 58

6.2 The Middle Package 59

6.3 The Clients Package 61

6.4 Further Implementation Notes 65

Chapter 7: Conclusions and Future Work 66

7.1 Discussion 66

7.2 Future Work 67

Appendix I: The Directory of Canadian Researchers in International Security (DCRIS) 69

I.1 DCRIS Description 69

I.2 DCRIS Entries 70

I.3 DCRIS Use Cases 72

Appendix II: Using Unified Modeling Language (UML) 75

II.1 Basic UML Elements 75

II.2 The WAE Model 76

II.3 Extending the WAE Model 79

The DCRIS Repository 82

III.1 DCRIS Tables 83

III.2 DCRIS Table Relations 90

Bibliography 92

List of Illustrations

Figure 1 A Simple View 4

Figure 2 Multiple Views 4

Figure 3 The Framework Packages 15

Figure 4 A Generic Repository Package 17

Figure 5 A Generic Middle Package 20

Figure 6 Middle - Repository Associations 21

Figure 7 Nested Middle Package Views 22

Figure 8 Middle Package Services 23

Figure 9 A Generic Clients Package 25

Figure 10 Clients - Middle Associations 26

Figure 11 Nested Clients 27

Figure 12 Invoice Example 29

Figure 13 DCRIS Design Overview 33

Figure 14 The DCRIS Repository Package 35

Figure 15 Repository Table Relationships 36

Figure 16 The DCRIS Middle Package 38

Figure 17 Middle to Repository Views 39

Figure 18 View Making up an Directory Entry 39

Figure 19 The DCRIS Clients Package 40

Figure 20 Views Making up an Entry List 41

Figure 21 ‘Keyword’ Example 41

Figure 22 ‘Keyword’ Display Example 43

Figure 23 Addition of Other Expertise 47

Figure 24 Addition of Language 49

Figure 25 Class Reuse Between DCRIS Applications 53

Figure 26 Framework Superclass Example 55

Figure 27 HTML Forms Templates 56

Figure 28 Sample *keyword* Table Contents 58

Figure 29 SQL Script for the *v_keyword* View 59

Figure 30 XML Representation of *v_keyword* 60

Figure 31 XML Representation of *v_keyword_list* 60

Figure 32 XSLT Template for a List of Keywords 62

Figure 33 HTML Code for a List of Keywords 62

Figure 34 Output of a List of Keywords 62

Figure 35 XSLT Template for a Keyword Form 63

Figure 36 HTML Code for a Keyword Form 63

Figure 37 Keyword Form Output 63

Figure 38 Web Page Request Processing 64

Figure 39 Typical DCRIS Entry 71

Figure 40 DCRIS Use Cases 72

Figure 41 DCRIS Repository Overview 82

Figure 42 *Keyword* Table 83

Figure 43 *List* Table 84

Figure 44 *Member* Table 84

Figure 45 *Member_Keyword* Table 85

Figure 46 *Member_Supp_Key* Table 86

Figure 47 *Publication* Table 86

Figure 48 *Pub_Type* Table 87

Figure 49 *Recommendation* Table 88

Figure 50 *Rec_Type* Table 88

Figure 51 *Supp_Key* Table 89

Figure 52 Table Relationships 90

Chapter 1: Introduction

1.1 Motivation

Web application design is a broad term that refers to any aspect of designing a Web application, including referring to designing Web interfaces to data. With the proper back-end server, Web browsers can provide an instant user interface for a wide range of applications. Input and output approaches are, broadly speaking, standardized and familiar to users. They can provide both client and server data processing making them flexible and allowing performance balancing. With the spread of the Internet, Web applications are becoming more and more popular.

One of the spurs to the dramatic growth of the World Wide Web is the ease in which content can be placed in Web pages and posted. Creating Web interfaces to databases needs software more sophisticated than a simple HTML editor or word processor. There are a number of commercial software tools available for implementing Web access to databases. However, such tools lack a common design paradigm [41]. These include Janna System's LivePage, Sybase Inc. Net Dynamo, IBM's Net Data, etc.[1] Free software tools such as Perl or C-based (Common Gateway Interface) CGI scripting and dynamic PHP Hypertext Processor (PHP) [44] Web pages are also in widespread use. In a typical Web application each Web page stands on its own and is custom written to perform a particular job. Client and server-side scripting provide some small measure of reuse through the use of script libraries and Cascading Style Sheets (CSS) [42]. Extensible Style Language (XSL) [43] style sheets provide reusable formatting code. These techniques apply only to portions of Web application implementation. This thesis suggests a basis for organizing Web application design.

[1] All trademarks and copyrights are owned by their respective companies.

1.2 Problem Statement

A successful Web application design must bring together three major elements: structure, navigation, and presentation. To bring these elements together in a Web application requires a clear and systematic design approach that overcomes design problems related to separation of concerns, design frameworks and patterns, and framework representation.

- In Web applications the three main concerns of structure, navigation and presentation are often intermixed and it makes the design difficult to organize, reuse, and extend [46, 18].
- There is a need for a comprehensive approach that support the design of a class of Web applications and is documented by design patterns [41, 10, 11].
- There is a need to be able to represent such a framework with an object-oriented extensible notation [34, 6, 26].

1.3 Proposed Solution

This thesis examines Web application design, specifically focusing on the issues of separation of concerns, reuse, and design representation. It proposes the use of views to deal with separation of concerns, a framework to deal with reuse, and extensions to the Unified Modeling Language (UML) to deal with representation.

1.3.1 Views

Separation of concerns is the first of the three key elements in Web application design that this thesis emphasizes. Separation of concerns is a software engineering principle that hides complexity through abstraction. Software applications have two 'concerns': First is the basic concern that deals with the essential functionality of an application, i.e., what is the main purpose of a particular piece of software? Second are the special purpose concerns, such as the user interface, control, or timing.

Separating these concerns makes software easier to design, implement, reuse, and modify. The goal of the views model is to abstract the modeling of relationships between modules, and thus ease the job of the designer [2].

One approach to designing a software application that emphasizes separation of concerns is to design software using *views*. The views (or *views-a*) model is an extension to the software design primitives *is-a* (specialization), *has-a* (aggregation), and *uses-a* (association). The Abstract Data Type (ADT), with its emphasis on information hiding, is a very important concept in software engineering, but is incomplete. The views model extends the concept of the ADT by providing an interface that acts as a buffer between an ADT and the external world or other ADTs [3].

The views model has two basic constructs: objects and their interfaces (views). An object should be designed to be independent of the external world as it is the job of the view to connect the object to that world. The way in which the internal states of an object are presented to the external world is irrelevant to the object. For example, the value of an integer in a viewed object could be presented by a view as a number, a position on a dial, or a point on a slider. A user could change this integer value in a number of ways. Yet for all of these, the viewed object need not 'know' which representation was used. The view provides a clear separation of external representation from internal state. The implementation details of an object-view relationship are unimportant for design purposes. Indeed, objects and their views can be connected together at code-time, link-time, or run-time. During design, only their states and public interfaces are considered.

Whenever the internal state of an object changes, a view of that object must change its presentation of that state. A view may also change the state of an object through the object's public interfaces, or controls. In other words, a view is a representation of the state of the object it views.

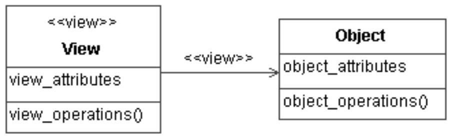


Figure 1 A Simple View

Figure 1 shows an example of a simple views relationship. Views and objects differ only in their the relationships they have with each other. A view is just another type of object that is called a view because of the duties it performs. Thus, a view may be viewed in the same way an object can be viewed. In this context, a 'view' becomes an 'object'.

A view puts a user at arms' length from the object it views. All interactions between the user and an object are encapsulated in the view. The only knowledge the user has of the object's internal state is provided by the view. Since a view is an object itself, it also has states and controls that can be accessed and manipulated by other views. Thus views may be nested, allowing the creation of complex user interfaces.

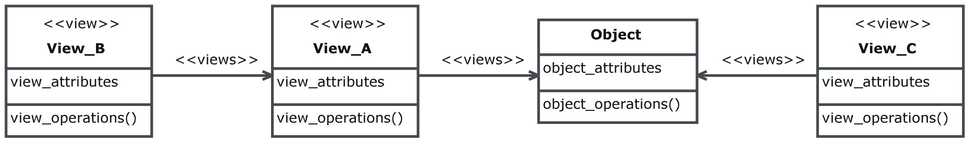


Figure 2 Multiple Views

Figure 2 shows a more complex views relationship consisting of four objects. *Object* is viewed directly by *View_A* and *View_C*. The one-way associations between the views and *Object* imply that *Object* is unaware of the existence of *View_A* and *View_C*. *View_A* and *View_C* can examine the state of *Object* at any time, but *Object* cannot view the states of *View_A* and *View_C*. Further, both *View_A* and *View_C* can alter *Object*'s state through *Object*'s public interfaces, whereas *Object* does not have access to the public interfaces of *View_A* and *View_C*. Both *View_A* and *View_C* must show *Object* in the same state. Since *Object* may be in only one state at any given time, *View_A* cannot show *Object* in one state, and *View_C* show *Object* in another. In other words, multiple views of a single object must be consistent. (Multiple views need not be the *same*, just consistent. For example, an analog display and a digital display can both be used to show the current time. The time is consistent; its views differ).

Views may be nested. *View_A* is viewed by *View_B*. To *View_B*, *View_A* is simply another object. *View_B* can examine and alter the state of *View_A*, whereas *View_A* is unaware of the existence of *View_B*. Since the state of *View_A* depends on the state of *Object*, and *View_A*'s has access to *Object*'s public operations, *View_B* can indirectly examine and alter the state of *Object* through *View_A*. *View_B* must display the state of *View_A* that is consistent with *View_A*'s display of the state of *Object*.

The advantages of using a views model are clear [3]:

- The views model emphasizes loose coupling between modules
- This separation of concerns improves reusability since modules are not closely linked together
- The higher level of abstraction makes a designer's visualization of a software system easier
- The designer need only formally specify the connections between modules and systems using the view model and leave the implementation details until later

1.3.2 Frameworks and Design Patterns

Reuse is key to keeping down the cost of developing large-scale software systems. The reuse of software design elements from one project to another allows rapid and efficient development of the new project through minimizing customization [12, 25]. One design pattern that emphasizes reuse is the *framework*.

A framework is a "reusable chunk of architecture" [16]. A framework describes the interaction between the various objects that make up a software system and provides common functionality for the applications that are built around that framework. This identification of common functionality is key to a framework's support for reuse - design components from one system can be used with little or no modification in another system where the systems are similar. A framework is generally useful only within a single problem domain and is therefore most useful when a domain contains many applications [18]. Well-designed frameworks evolve through use. As more applications are developed for a particular framework, features common to those applications can be built into the framework, thus improving the framework [16].

Modularity can also be emphasized within a framework by dividing the framework into several parts according to their concerns (interface, control, timing, etc). These parts can be individually modified, and reused. Further, the framework can be extended to add new functionality as it provides hooks for the addition of new design elements [5]. The framework presented in this thesis uses views to establish modularity and reuse as well as separation of concerns.

1.3.3 Representation

Representing a software design in a coherent, logical, and standard manner is key to the comprehension of that design [26]. Muddled representation leads to muddled implementation. A good modeling language is important to good design representation.

The Unified Modeling Language (UML) [32] is a widely accepted standard for modeling software systems. UML is a highly flexible, extendable, modeling language that can be used to represent a wide range of software systems and individual design elements [16]. This thesis uses an extended version of UML to model a framework for Web applications. The first extension involves the representation of views as particular types of UML classes. The second extension has been developed by Conallen [6], and is named Web Application Extension (WAE). *Appendix II* shows these extensions in more detail. This thesis demonstrates how the WAE model can be used as part of a views-based framework to improve the design of Web applications.

1.4 Contributions

The thesis of this work is that defining an approach to designing and implementing Web applications based on views, frameworks, and patterns allows us to extend an object-oriented design representation and develop reusable and extensible design solutions for a class of Web applications in which the concerns are separated.

The major contribution of this thesis is the development of a new approach to designing Web applications. The work involved in attaining this goal has produced the following visible contributions:

- a model for Web applications based upon a separation of concerns using views
- a framework for system elements so that design patterns can be used to build the application.
- a reusable design approach so that many Web applications can be built around the same framework.
- a coherent and organized framework representation using extended UML.
- an implementation approach that can be implemented on a number of different platforms using a range of software and tools.

In support of this solution, this thesis will show the design and implementation of a proof-of-concept Web application using this design technique.

1.5 Outline of the Case Study

The Directory of Canadian Researchers in International Security (DCRIS) was established at Wilfrid Laurier University in Waterloo, Ontario, to support research on all aspects of international security by identifying Canadian researchers and their specific areas of expertise, and by facilitating communication amongst the community of Canadian researchers in international security. (See *Appendix I* for a description of the Directory and its contents).

It was important that the Directory be made available to as many people as possible across Canada in order to facilitate communication amongst its members. Access to the Directory had to be simple, fast, and reliable. Using the World Wide Web was an obvious answer to these problems: Web pages are, for the most part, platform independent; Web browsers are relatively easy-to-use and have consistent interfaces; Web pages can easily mix text, graphics, video and sound; and the Web can be accessed world-wide. In 1996 the author was asked to design and implement two Web applications for the Directory - one for public access and one for administrative access. The author was chosen for this task because of his experience in Web application creation for the Wilfrid Laurier University's Department of Physics & Computing.

Since 1991, the Department of Physics & Computing at Wilfrid Laurier University has been making increasing use of electronic instructional tools in its courses. In the beginning, course assignments were made available on the university's mainframe computer, and electronic mail was used to enhance communication between students and course instructors. With the appearance and widespread adoption of the Internet, course material was made available through World Wide Web pages. This material included lectures, labs, assignments, and links to off-campus resources. Automated quizzes were created and used to a limited degree in some courses. Student marks were made available to students on the Web. Courses used electronic mailing lists to pass on course announcements, and provide a forum for discussion of course material and concerns. Much of the information available through the Department's Web applications was dynamic. Student grades, for example, could not be efficiently presented through static Web pages because of security concerns and the sheer volume of such data. The Department made extensive use of Web pages that were built on the fly from data extracted from a repository. Such data could also be searched and filtered dynamically. The DCRIS administrator was interested in tapping into this experience and making use of it in the creation of the DCRIS Web applications.

The first iterations of the DCRIS Web applications were based upon data stored in flat files accessed through Common Gateway Interface (CGI) scripts. These scripts, written in Perl for the most part, allowed users of the DCRIS public Web application to search the Directory. They also allowed the administrator to store and update the Directory contents. This method allowed some degree of software reuse as the same CGI scripts could be called from different Web pages as needed. This reuse was done strictly on an ad-hoc basis, however. As well, although the hypertext aspect of the Directory material was much enhanced, all links had to be designed, implemented, and maintained by hand. Consistency across Web pages also had to be maintained by hand.

Although this combination of ASCII flat files and CGI scripts worked, it was script and server intensive, and the performance suffered. As well, the data was not very portable and changes to the layout of the Directory contents were difficult to implement. When the Department acquired a Web server capable of running an SQL database the Directory data was moved to this database. This reduced the need for scripting, allowed the separation of the database server and the Web server, and significantly improved the speed of the delivery of the Directory content. Automated Web-based testing tools used by the Department went through a similar evolution.

In 1998 the Department acquired Janna System's LivePAGE software, which eased the management of Web applications, and, with its SQL database underpinnings, improved the ability of the designer to develop dynamic Web pages. Livepage allows the embedding of SQL code and scripts in various languages in Web pages, and is much easier to use and organize than CGI scripting. The Directory was reimplemented in LivePage in 1999 / 2000.

As useful as these evolving software tools have been in dealing with the implementation of the Directory Web application, none of them deal with the issue of how such a Web application should be designed. As well, as the software and interfaces improved, so did the demands placed on them. The Directory administrator requested more searching and filtering capabilities, easier updating, and statistical information on the Directory's use. The lack of an organized design for the Directory Web application made adding these capabilities difficult as there was no organized framework onto which these new features could be easily grafted. Further, the lack of an underlying framework and reusable modules made the application increasingly difficult to maintain because of the lack of organization amongst the application's modules.

1.6 Related Work

Many approaches concentrate on the application architecture. Ebner et. al. [27] suggest a five-layer architecture based upon an object-oriented framework. This architecture divides the application into five modules: presentation, user interface components, business logic, data management, and system infrastructure. The data model is explicitly separated from the data source, and uses a broker interface between the modules, allowing components to be added, deleted, and modified with minimum effort. This thesis suggests that business logic should not be treated as a single indivisible module, but rather that all elements of a Web application design have their own business logic.

One approach to improving Web application design is through the use of domain-specific languages. The <bigwig> language, as described Brabrand et. al. [28], is a high-level language specifically designed for the implementation of interactive, dynamic content Web sites. <bigwig> is designed to deal with what the authors define as the six major problems of Web site design and implementation: sessions, dynamic documents, concurrency control, form field validation, database integration, and security. This thesis suggests extending an existing design language (i.e. UML), rather than creating a custom language.

Applying an object-oriented framework to Web application design is the approach of a number of recent papers. Schwabe et. al. [29, 30] propose using the Object-Oriented Hypermedia Design Method (OOHDM) to design Web applications. OOHDM allows a designer to view a Web application design from three different perspectives: the conceptual model, which represents domain objects and their functionality; the navigational model, which defines navigation nodes as views of conceptual objects; and the interface model, which represents the visible manifestation of navigation nodes. Within each of these models design patterns such as wrappers, advisors, observers, and decorators are used to ease the burden of design. In [30] the authors advocate the use of views (using the views approach of [1]) to define navigation nodes. This thesis also suggests the use of views applied to three different perspectives.

The object oriented-hypermedia (OO-H) method outlined by Gomez et. al. [31] is another object-oriented approach to Web application design. Building upon Conallen's work [33], it uses the Unified Modeling Language (UML) to represent Web application architecture. In a way similar to that used in [29], OO-H divides the Web application design into three views: the conceptual view; the navigation view; and the presentation view. Each of these views has its own UML representation. The OO-H method further calls for the use of XML together with XSL stylesheets rather than HTML in order to provide device-independent flexibility in the Web application design and implementation. This thesis also examines the use of XML and XSL to implement Web application designs.

The UML is in common use in software design; indeed, it is claimed that "The Unified Modeling Language (UML) ... is a visual modeling language that is presently the de-facto standard for modeling software intensive systems" by Larsen and Conallen [36]. The UML, and extensions of it, are put forward as being an excellent method of representing web application designs by Conallen [6, 33], [31], Fuentes et. al. [35], and Larsen and Conallen [36]. Developing UML profiles, or collections of UML elements to represent a particular design approach is suggested in [35]. The authors point out that although application frameworks are an excellent design tool they often suffer from poor documentation. UML profiles are a way of providing a common graphical notation and vocabulary for a particular application domain. They offer a case study of a model named MultiTEL as an example of how UML profiles can be developed and applied. Although not called a ‘profile’, both [6] and [36] propose a similar approach through a set of UML extensions called the Web Application Extension (WAE). In addition, the authors propose a Reusable Asset Specification (RAS) for defining reusable bundles of design and code in a Web application. This thesis suggests the use of UML, but not the ‘profile’ concept.

Despite the popularity of UML, not all researchers in this field are enamored of it. Reinhartz-Berger et. al. [40] suggest that a UML description of a Web application is unnecessarily complex. They propose an approach called OPM/Web, an extension to the Object-Process Methodology (OPM). As an object-oriented approach OPM/Web emphasizes the separation of Web design into three main areas: structure, navigation, and presentation.

Another object oriented approach to Web application design is the Extensible Web Modeling Framework (XWMF), explained by Klapsing et. al. [39]. XWMF is an application of the Resource Description Framework (RDF), which is a World Wide Web Consortium (W3C) standard for processing metadata. The paper demonstrates how a Web application design formally described using XWMF separates content and layout and supports reuse. Further, the paper describes tools developed by the authors to create, process, query, and analyze a Web application’s RDF description. This thesis suggests that using metadata as a basis for Web application implementation is fertile ground for future work.

The use of XML for data communication between a database and a web application is discussed by Bertino and Catania [38]. The authors describe how XML is becoming a standard data interchange format for the Web and the problem of storing and accessing XML using relational databases. They discuss how XML data is homogenous and structured, heterogenous and unstructured, or a hybrid of these, and databases can either mimic an XML data structure or simply store raw XML data. Each approach has benefits and drawbacks in terms of design and implementation. Bertino and

Ferrari [37] stress the difficulties of integrating data models, schemas, and instances, and how XML can be used to ease some of these difficulties. This thesis shows how XML can be used to implement a views-based framework design.

1.7 Thesis Overview

This thesis is organized as follows. In *Chapter 2* we describe a generic views-based framework for Web application design, showing how it can be broken down into different design packages: the repository package, the middle package, and the clients package. Further, it demonstrates how views provide the connections between the different packages and isolate them from each other. In *Chapter 3* we give a specific example of a Web application design framework and discuss how the various elements of the DCRIS public Web application can be designed using this framework. In *Chapter 4* we demonstrate how the framework can be extended to add additional functionality to the DCRIS public application. In *Chapter 5* we discuss reuse of the framework with respect to the DCRIS administration Web application and another, significantly different Web application, and show how reuse can take place both at the design and code level. In *Chapter 6* we demonstrate how the framework design concepts can implemented and use the DCRIS Web applications as specific examples of proof of concept implementations. In *Chapter 7* we discuss the lessons drawn from the thesis' design approach and possible future work. In *Appendix I* we describe the DCRIS projects in some detail. In *Appendix II* we describe the UML extensions used in this thesis. In *Appendix III* we describe in detail the structure of the DCRIS relational database repository.

Chapter 2: A Views-Based Framework

In this Chapter we introduce a Web application design framework. This framework is divided into three separate packages, the Repository Package, the Middle Package, and the Clients package. Each of these packages uses views to promote separation of concerns and reuse. This framework is amenable to being modeled in UML. These UML elements are described in detail in *Appendix II.*

2.1 The Framework

The key to understanding this framework is to recognize that a set of data can be viewed as a collection of objects. Objects that share common attributes can be viewed as classes. Further, these objects can be viewed as existing at three different levels. On the first level an object exists in persistent storage. Such an object may be of limited use to a user because of its simplicity or lack of readability. These objects can be searched, sorted, and selected. The concern at this level is strictly permanence and access to these objects. At the second level further objects may be instantiated, updated, deleted, or combined by manipulating the objects in storage. More complex objects may be made up from simple objects or even simpler objects may created from extracting a subset of the attributes of a particular object. The concern at this level is the creation and updating of objects, some of which may have no permanent existence, but are made up of combinations of permanent objects. At the third level the objects from the second level may be formatted so as to be accessible to a user. In their raw form objects at the second level are generally not easily decipherable and the concern at this level is to make these objects accessible to a user, human or otherwise. At all three levels there are rules as to how these objects should be processed and views are used to describe this processing.

As a simple example, imagine a company that needs to produce invoices for its customers. At the first level the company has a database that stores customer and order information. The way in which these customer and order objects are stored follows the business rules (explicit or implicit) that this company has laid down. At the second level the company produces invoice objects based upon their customer and order information. The company has rules for what goes into an invoice, for example the customer's name and address may be part of the invoice, but the customer's phone number may not, despite the fact the company has every customer's phone number in its database. The company bills customers on a monthly basis and all orders for a particular calendar month become part of a customer's invoice. Further, information can be added to an invoice. The invoice must have a total based upon the orders it contains, though no such explicit total is stored in the company's database. At the third level, once the invoice object has be built from the appropriate customer and order objects and further processing applied, the invoice must be sent to the customer in a format the customer can read. The company gives customers a choice of how they would like to access an invoice: through a Web page, a letter, or by phone. The company must format each particular invoice according to the customer's desires.

This example illustrates how a particular problem (e.g. producing invoices) can be divided into the three elements that make up the proposed framework: storage, manipulation, and display.

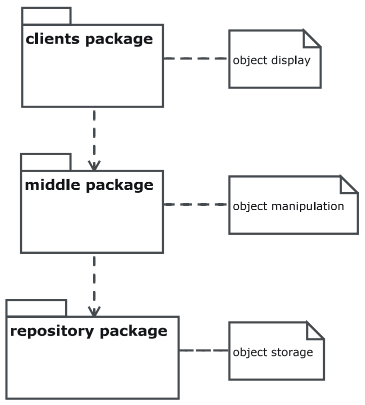


Figure 3 The Framework Packages

Figure 3 shows the elements that make up the framework. Each element is represented by a

UML *package*. A package is a high-level container for UML design elements and is useful for breaking a large model up into smaller, more manageable pieces [7]. Packages may be related in a number of ways. In **Figure 3** the dashed arrow between packages represents a dependency. In UML a dependency relationship specifies that a change in the source may affect the target that uses it, but not necessarily the inverse. The dependencies in **Figure 3** show that the *clients package* depends on the *middle package*, and the middle package depends on the *repository package*. Thus changes to the middle package do not affect the repository, but changes to the repository will likely affect the middle. Similarly changes to the clients package do not affect the middle package, but changes to the middle package will likely affect the clients package. This fits nicely into the views concept as a change in a viewed object affects a view of it, but a change in a view does not affect the object being viewed. The dashed lines in **Figure 3** between the packages and the notes to the right simply indicate the relationships between the UML elements and the corresponding notes.

The repository package represents the object store that a user wishes to access and the operations needed to access that object store. The repository architecture and design is a well-known software design pattern [11], as described in section 1.3.2. In implementation terms a repository can be a database, a set of text files, a set of XML data, or some other kind of data storage mechanism.

The middle package contains objects extracted from the repository. These objects can be a subset or a superset of all the objects available in the repository. In the middle package there are methods for searching, sorting, and selecting objects acquired from the repository. In implementation terms the middle package is composed of database scripts or other executable programs.

The clients package represents a user’s view of the repository contents as filtered through the middle package. Depending on the user’s needs, this view can take many different forms such as paper output, machine readable output, and video display. The clients package views assemble the objects provided by the middle package views into appropriate output formats.

2.2 The Repository Package

A repository design pattern provides a central data structure with an access interface for multiple clients; these clients are independent of one another; each client can access the repository simultaneously; and the repository controls client access through a transaction mechanism. The repository reacts only to requests from outside, and thus does not have a signaling mechanism to alert external clients of internal changes. Repositories in general may have signaling mechanisms, such as triggers; such mechanisms are not necessary for this particular design. Clients must query the repository to discover if any changes in its internal state have occurred [12].

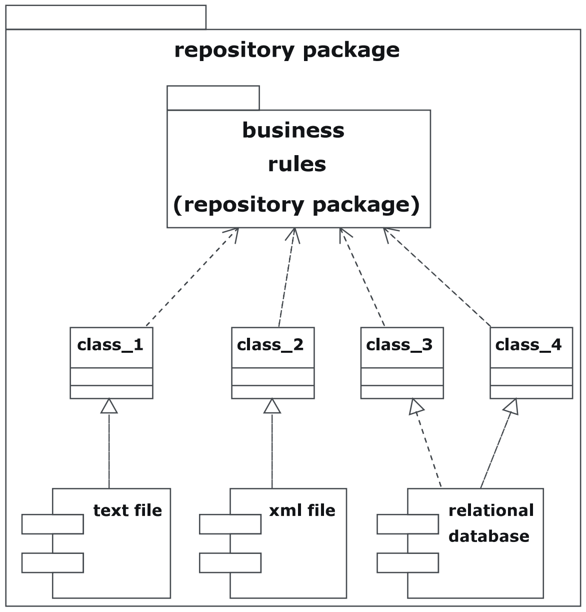


Figure 4 A Generic Repository Package

Figure 4 shows a generic repository. A repository may contain a number of classes (here labeled *class_1*, *class_2*, *class_3*, and *class_4*), each of which is realized by a *component*. In UML a component represents the implementation level of a software system. Components map to some real-world element, such as a text file, xml data, or a relational database. Components are important in the design of a software system since every conceptual element of a design must eventually be mapped to a real-world element. A system's components need only be added in the final phases of the system's development. The realization relationship is represented in UML by a dashed arrow between a module and the component that realizes it, as between the *class_1* class and the *text file* component. The realization arrow is similar to a generalization arrow, except that the shaft of the arrow is dashed, implying that realization is a form of inheritance [16].

Figure 4 shows that the repository package contains *business rules*. The various classes shown are all dependent upon the repository's business rules package. Business rules represent the conceptual definitions of objects within the repository, the classes are the design realizations of those business rules. As before the dependency relationship between the business rules and the classes that depend on them implies that as the rules changes, the classes must be altered to fit those rules. On the other hand changes to the classes (e.g. a change in language used to define the classes), do not imply a change to the rules governing those classes, merely that a different means of expressing a business rule has been chosen by the designer.

Using the example given in Section 2.1, a repository may contain a customer class based upon the company's definition of a customer. This class may contain customer information such as name, address, telephone number, and customer ID as required by the company's business rules regarding customers. If the company uses an SQL database, then the actual customer data would be stored in an SQL table. If the company changes their definition of a customer, then both the customer class and its implementation would have to be changed to match the new customer definition.

The key to understanding the usefulness of a repository is flexibility. A repository structures data, allowing it to be searched, sorted, and selected. The repository provides a clear and concise interface for clients to access this data. The repository does not impose any display formatting on its data, but it does impose a structure that governs how data may be accessed. The repository is concerned only with the data itself. More complex relationships and presentation must be handled by the other elements of the framework [14].

Separating content from the other elements of a software system not only makes the overall design easier, it also makes future expansion and uses of the system easier. A repository is completely unconcerned with the presentation of its data, and therefore any clients wishing to present this data must apply their own formatting. The corollary of this is that the repository does not impose any limitations on the formatting that can be applied to its data, and thus clients may apply any formatting they like to the repository's data so long as they comply with the repository's interfacing rules.

In keeping with a views approach to software design a repository is completely independent of the objects that view it. It merely delivers or updates data upon request. Any formatting that must be applied to the repository data must be applied by the objects that view the repository, not by the repository itself. The objects that view the repository must respect the repository's rules for accessing data, not the other way around. Thus separation of concerns is enforced.

Given its independence from the other elements of a particular software system, a repository can become part of many software systems. In one system a repository could be the source of data for a Web application. In another system that same repository could be the source of data for a database query tool. In yet another it could be the source of data for documentation. The applications may be isolated from each other as they have no common elements beyond the repository itself [12].

2.3 The Middle Package

A repository provides the means for the data it contains to be sorted, searched, and selected. However, it does not decide on the sorts, searches, and selections. Before data can be displayed by a client, it must be structured in some way. The internal structure imposed upon data by the repository may be of no use to a client. For example, if data is stored in an SQL database it is very likely normalized, i.e., redundancy is minimized and relations between objects implemented through some arbitrary foreign key values. The final output desired by a user may need that redundancy reimposed and may not need these foreign key values exposed. Thus the objects retrieved from a repository may need restructuring before the appropriate presentation formatting can be applied. A *middle view* therefore represents a view of an object before it is formatted for display purposes.

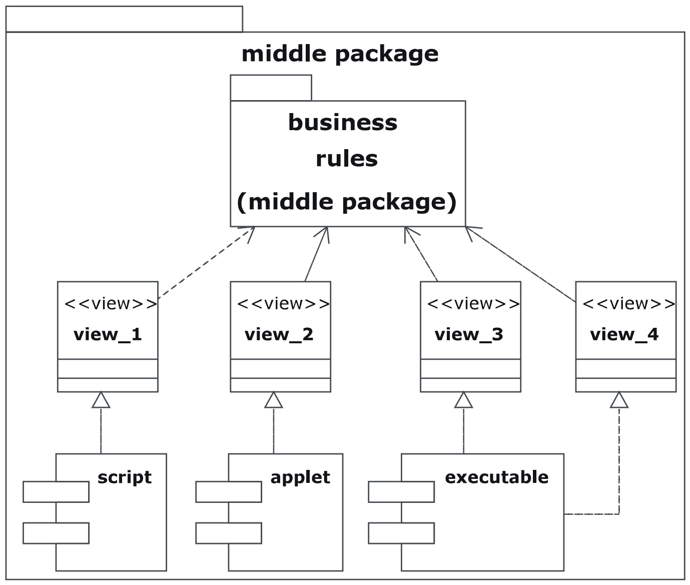


Figure 5 A Generic Middle Package

Figure 5 shows a sample generic middle package. A middle package may contain a number of views (here labeled *view_1*, *view_2*, *view_3*, and *view_4*), each of which is realized by a component. These components represent real-world resources: a script (such as an SQL script); a java applet; or an executable file. As shown by relationships between the *executable* component and *view_3* and *view_4*, one component may support multiple views.

The middle package also contains a *business rules* package. The business rules package represents the sets of rules that must be followed when creating a view of data in the repository. These rules determine the selections and projections that are applied to the repository data when generating a view. These rules are determined by the real-world demands placed upon the software system. As these systems undergo change, the business rules may also be modified. Note that the repository structure need not necessarily change in response, merely how data is retrieved from that structure.

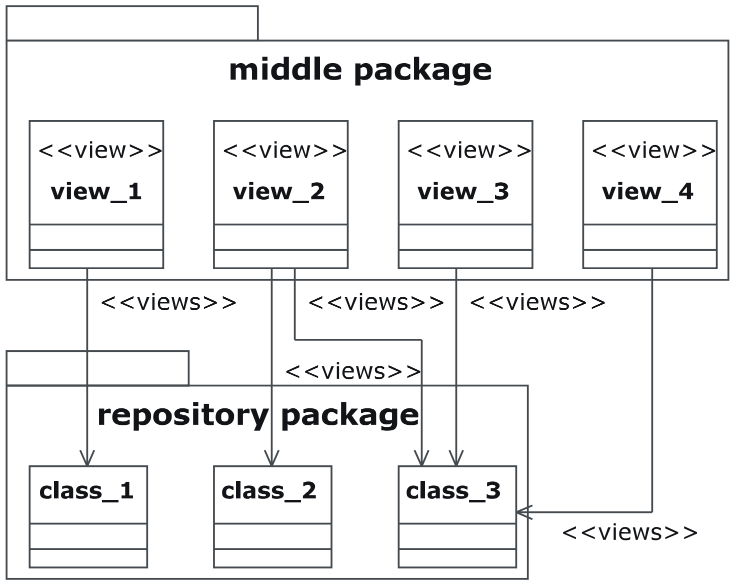


Figure 6 Middle - Repository Associations

The package notation is simply a short-hand way of showing the relationship between two high-level design components. **Figure 3** shows the dependencies between the highest level packages in the design framework. At a lower level, **Figure 6** shows how the individual views of the middle package relate directly to the individual classes in the repository package. One view may be restricted to a single class in the repository, as in the relationship between *view_1* and *class_1*. A view may reference more than one class in the repository package, as in the relationships between *view_2* and classes *class_2* and *class_3*. A single class may be viewed by multiple views, as in the relationships between *class_3* and views *view_2*, *view_3*, and *view_4*. Again, it is important to emphasize that in keeping with the definition of a view, the classes within the repository package are unaware of the existence of the views in the middle package. The only requirement of the repository package classes is that they provide an interface to allow themselves to be viewed. Views may not access repository package classes except through these interfaces.

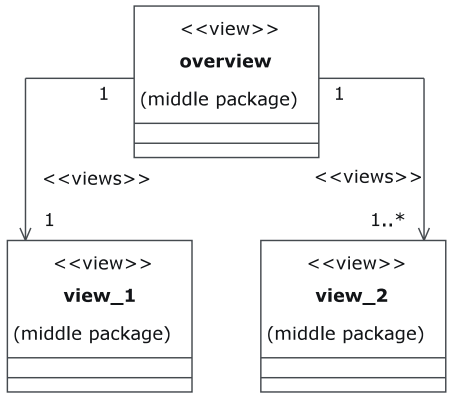


Figure 7 Nested Middle Package Views

Views in the middle package are not limited to viewing repository package classes. In general views may be made up of other views. **Figure 7** shows a sample *overview* view that references both *view_1* and *view_2*. The association multiplicities show that *overview* references only one occurrence of *view_1* but references one or more occurrences of *view_2*. Although *overview* does not access any repository package classes directly, it has access to the repository data through *view_1* and *view_2*. In real-world terms *overview* (as its name implies) may represent some sort of aggregate report. Data is extracted from the repository by *view_1* and *view_2* and *overview* bundles that data together, selecting and projecting elements of those views as necessary. Furthermore, *overview* could provide sums or other aggregate functions of the data in the two views it accesses.

The views *view_1* and *view_2* may or may not be able to stand alone in terms of being accessed for display purposes by elements of the clients package or they may exist only to serve the needs of *overview*. What is important is that either case may be true. Views in the middle package may be reused for different purposes. For example, *overview* may represent an invoice, *view_1* customer information, and *view_2* a list of items purchased by that customer. *view_1* could be reused as part of a list of customers and *view_2* as part of a list of all items sold in a month.

The middle package has two further functional aspects: it controls the flow of data between the repository package and itself, and between itself and the clients package. We call these specific functions *the network services* and *client services* respectively.

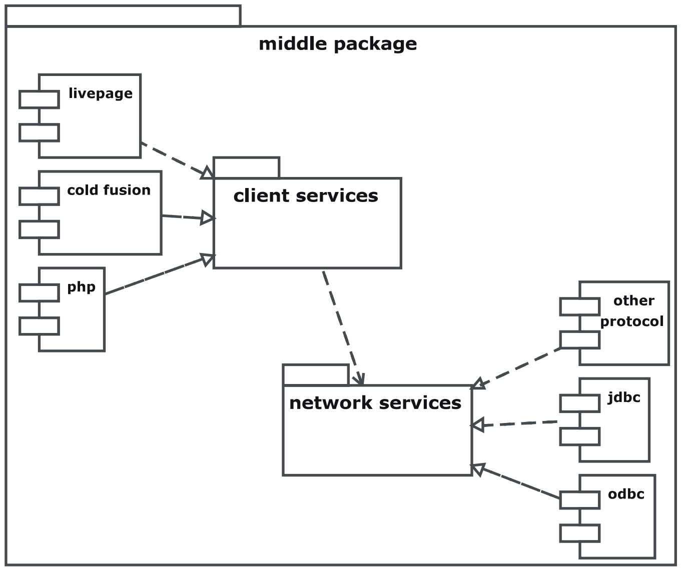


Figure 8 Middle Package Services

Figure 8 shows the components that make up the middle package services. The network services represents the distribution of data within the framework design, as implemented using protocols such as Open Database Connectivity (ODBC), Java Database Connectivity (JDBC), or other protocols. These other protocols need not be database protocols. Since the repository may consist of files rather than a relational database, these other protocols may simply be the operating system's file access protocols.

The client services represent the applications that take the data retrieved by the network services and process it before passing it onto the clients package. These applications are generally packages such as Cold Fusion, LivePage, or PHP Hypertext Processor (PHP). The client services implement the middle package views - the processing and formatting of repository data.

Access to the data in the repository package is granted by a discretionary security mechanism. In a discretionary security mechanism a user has access to an entire data structure or none of it [24]. If the repository is based upon a relational database such a structure could be a table or a set of attributes of a table, and access to these structures would be controlled by the database's internal security mechanism. If the repository is based upon a file system, then the data structure is a file and access to the file is controlled by the file system's user permissions mechanism. In order to access these data structures the middle package's network services are treated as users by the repository. A casual user, who is anyone who is not an administrator, has no way of accessing a Web application's data directly and must work through the middle package's services. If there are logins and passwords necessary to access persistent storage, these logins and passwords are hidden from the casual user within the middle package. The casual user does not even know how or where the data is stored. The Web application acts as a buffer between the causal user and the data the user ultimately wishes to access.

Mandatory security restricts access to data based upon the contents of the data rather than the data structure [24]. Mandatory security is applied by the middle package's client services. A client service applies data filtering to the objects the network service retrieves from the data source, allowing only data that the end user is authorized to see to get through to the Web server. As the client service processes requests it examines the credentials of the user requesting the data and fulfills or rejects requests accordingly.

2.4 The Clients Package

An object constructed by a view in the middle package is of little use to a user unless that object has been formatted in a way that makes it accessible to the user. Because the middle package elements concern themselves only with providing a structured view of the data in the repository, applying display formats to that data must be left to the views of the clients package. This distinction between object storage and manipulation and how objects are viewed is key to the concept of separation of concerns. The only concern of the clients package is to present data in some appropriate display format.

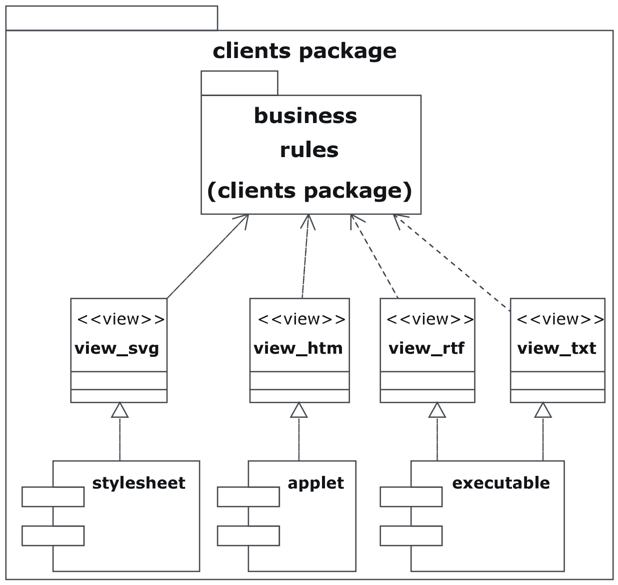


Figure 9 A Generic Clients Package

Much like the generic middle package of **Figure 5**, **Figure 9** shows a sample generic clients package. A clients may contain a number of views (here labeled *view_svg*, *view_htm*, *view_rtf*, and *view_txt*), each of which is realized by a component. These components represent real-world resources: an XSL stylesheet; a java applet; or an executable file. As with all components each of these has a interface defined: given the proper parameters and access to the repository package classes they will produce the desired view object. As shown by relationships between the *executable* component and *view_rtf* and *view_txt*, one component may realize many views.

Further, as with the views in the repository and middle packages, the views in the clients package depend on business rules. These rules define not what data is to be presented, but rather how it is to be presented. Referencing the example given in Section 2.1, imagine that an invoice object has been created by the appropriate view in the middle package. Because the company wishes to produce an HTML version of that invoice an appropriate view in the clients package is used to apply HTML code to the invoice object. The invoice view of the clients package determines which data elements have what codes applied (bold or italics, for example), and where on the page the data is placed.

Changes in the formatting and layout of the invoice do not affect the data that makes up the invoice as defined in the appropriate middle package view.

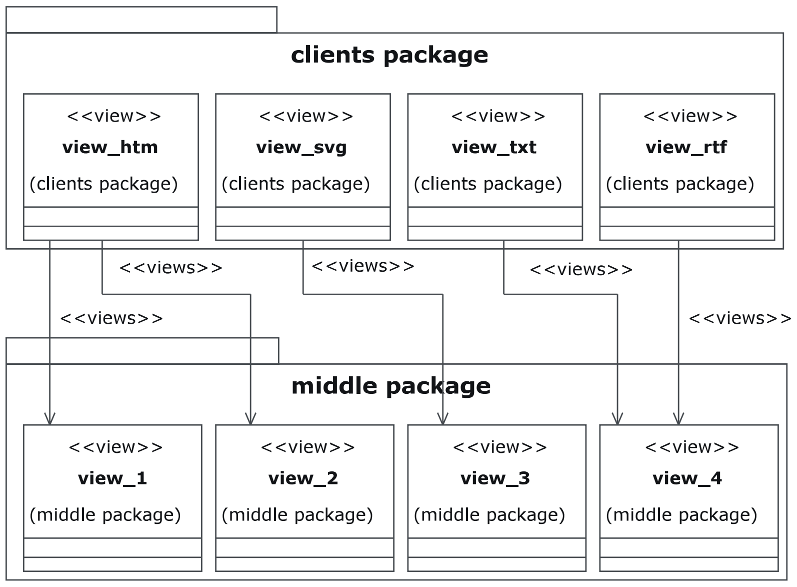


Figure 10 Clients - Middle Associations

Like **Figure 6**, **Figure 10** shows how the individual views of the clients package relate directly to individual views in the middle package. One clients package view may be restricted to a single middle package view, as in the relationship between *view_svg* and *view_2*. A clients package view may reference more than one middle package view, as in the relationships between *view_htm* and the middles *view_1* and *view_2*. A single middle package view may be viewed by multiple clients package view, as in the relationships between *view_4* and views *view_txt* and *view_rtf*. Again, it is important to emphasize that in keeping with the definition of a view, the views within the middle package are unaware of the existence of the views in the clients package. The only requirement of the middle package classes is that they provide an interface to allow themselves to be viewed. Clients package views may not access middle package views except through these interfaces.

The arbitrary names given to the views of this generic package are illustrative of their range and purpose. Thus, the name of the view *view_htm* implies that it produces an HTML formatted object from the data provided by the middle *view_1* and *view_2*. Similarly the name of the view *view_rtf* implies that it produces an rtf formatted object from the data provided by the middle *view_4*. This does not imply that these are the only display formats that can be applied to middle package objects; rather it simply indicates the possibilities available to the software designer.

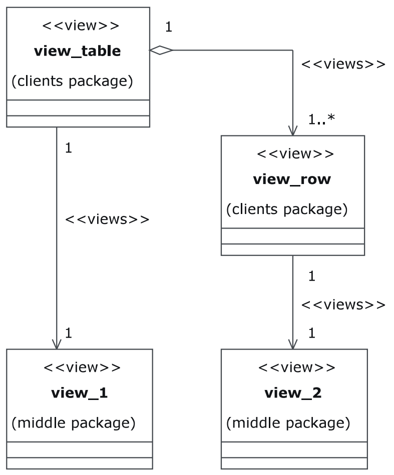


Figure 11 Nested Clients

Like the middle package, the clients package may contain nested views. **Figure 11** shows an example of such nested views. The view *view_table* references a single instance of the middle *view_1* directly, perhaps to get a title for the table. Each instance of the *view_row* view references a single instance of the middle *view_2*. However, *view_table* must contain one or more instances of *view_row*. Thus, as the names of the views imply, a table of data (its actual formatting is irrelevant) is made up of a set of rows of individual data elements. These data elements are extracted from the repository through the middle elements as the clients have no direct access to the repository itself.

Separation of concerns can be made an intrinsic part of software design using this views framework approach. Data storage and structure is separated from how the data is selected, projected, and joined. This process of selection and projection is further separated from how the resulting objects are displayed. Each level is isolated from the one above it. Changes to the clients layer have no effects on the middle layer, and changes to the middle layer have no effect on the repository layer. Further, this layered approach promotes reuse. Because an object in the middle layer is independent of how it is to be displayed, it can be extracted from the repository once and displayed in as many different ways as the designer desires by simply applying different views to it. The idea is to minimize the number of classes in both the repository and the middle layers. A properly designed view can be used in many places, minimizing design time and making global changes easier to implement. Reuse is key to the importance of views, since effective software reuse can improve software quality [5].

Discretionary security is also applied at the clients package level by restricting access to the application's Web pages. Each Web page in the Web application contains some top-level view of the data in persistent storage. In the same way that a database table or a file are structures, a Web page can be treated as a structure. Users are granted access to these pages through a Web server authorization mechanism. Thus an administrator has access to Web pages that allow updating the contents of the repository while a casual user has access only to Web pages that produce reports.

Together, security mechanisms in the repository, middle, and clients packages provide a Web application with the ability to provide different views of data based upon a user's credentials and authorization. For example, in a Web application used by a university department, a professor is allowed access to Web pages that list all students' marks for a course that professor teaches. Further, the professor is allowed to manipulate those marks through appropriate Web pages, and see aggregate information such as an average mark. A student is allowed access to Web pages that contain only information relating to that student, such as the student's grades or course schedule. The student is not allowed to see information belonging to other students or to change any of this information. The student may or may not be allowed to see aggregate course information such as the average course grade. A department head is allowed to set which Web pages can be seen by which professors based upon the courses that they teach.

2.5 A Simple Framework Example

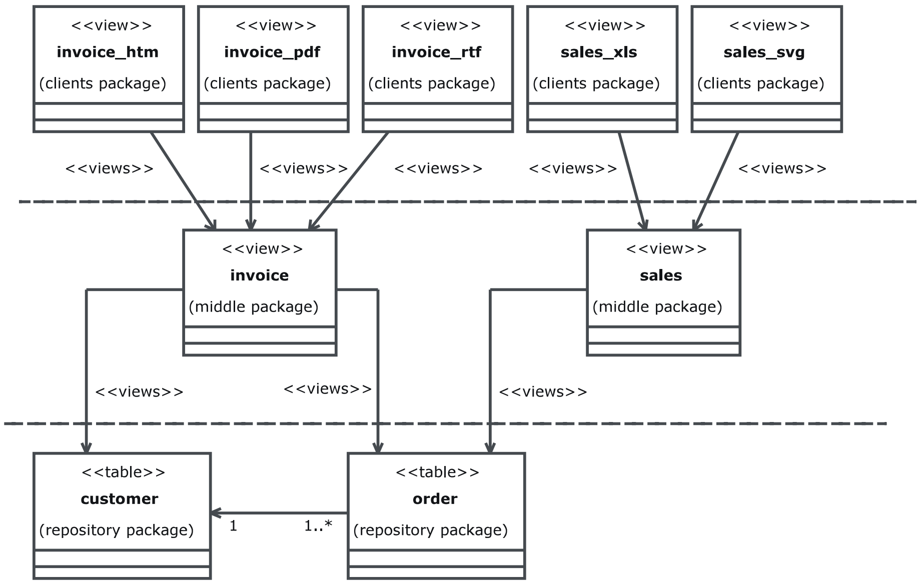


Figure 12 Invoice Example

Although Chapter 3 discusses in detail a project implemented using the views framework model it is instructive to provide a simple example of how this general approach to designing a software system using a views-based framework can be applied. **Figure 12** shows how the three packages of the framework can produce a set of invoices and sales reports in various formats. The repository package contains two data tables, *customer* and *order*. Both *customer* and *order* are tables of data as shown by their stereotypes. The multiplicity of the association between them shows that a customer may have many orders, but each order belongs to only one customer. According to the business rules governing this data, each time a customer places an order an invoice must be generated. Thus, in the middle package, an *invoice* view is defined according to this business rule. An invoice combines appropriate customer and order data. Note that not all of the data for a particular customer or order may appear on the invoice. Part of the duties of the *invoice* view is to select and project only the elements of the repository data that fit the business rule that defines it. Thus, an invoice may include a customer's shipping address but not their phone number. As well, the invoice may generate data that does not appear in the repository. The total value of an invoice may not be stored in the *order* table, but the *invoice* view could generate it on the fly. The important thing is that when the *invoice* view has been generated for a particular order and matching customer, that view contains all the data, and only the data, defined by the business rule for an invoice. The *invoice* view is not ordered or formatted for display purposes. The data it contains is structured only so that it can be retrieved by a clients package view that understands its structure. In other words, the *invoice* view has interface methods that allow a clients package view to access its data.

According to the business rules governing these invoices, each customer receives three copies of an invoice: a Web page generated when the customer places an order as represented by the view *invoice_htm*; a pdf version emailed to the customer as represented by the view *invoice_pdf*; and a paper document mailed to the customer as represented by the view *invoice_rtf*. These three views format the data contained in the *invoice* view according to their particular display languages, html, pdf, and rtf.

Using a similar approach to the invoicing example, generating sales reports is a two step process. The imaginary company's sales reports are based upon the customer orders generated over a period of time. Thus the middle package's *sales* view is based upon the data in the *order* table. The *sales* view contains only the data pertinent to a sales report. Thus it leaves out any customer information that is part of an order, selects only orders placed during a given time period, and based upon the company's business rules generates appropriate totals and subtotals for items sold and values invoiced.

Once the *sales* view has been generated it has to be formatted by the clients package into a form appropriate for display. In this example the sales data is formatted to fit a spreadsheet (represented by the view *sales_xls*), and for graphical display (represented by the view *sales_svg*).

This simple example shows how the layered views-based framework can support separation of concerns and reuse. In summary, each of the three layers has a specific concern. The repository package concerns itself with data storage and retrieval. The middle package concerns itself with creating objects by selecting and projecting data from the repository according to business logic. The middle package need not know how the repository is implemented, it needs to know only how to interface with the repository. The clients package concerns itself with formatting the objects created by the middle package. As with the relationship between the middle package and the repository, the clients package need not know how the middle package is implemented, it needs to know only how to interface with the middle package.

This example also shows how reuse can be achieved. The data in the *order* table is used by both the *invoice* and *sales* views for very different purposes. Although defined only once, both the *invoice* and *sales* views are referenced by multiple views in the clients package, also for very different purposes, and using different formats.

Chapter 3: A Sample Web Application Framework Design

This chapter demonstrates how the views-based framework was used to design an actual application. It shows how views are an integral part of this framework, and how each of the packages within the framework take on a different role in application design. Indeed, the separation of the design elements into different packages reinforces both reuse and the separation of concerns, both critical goals in software design. An overview of the framework is followed by individual descriptions of each package and how these different packages are integrated.

3.1 Application Overview

The applications designed using this framework are the Directory of Canadian Researchers in International Security public Web application and administrator Web applications. The public Web application is a Web application that allows anyone with an internet connection and Web browser to search for and examine information on members of the Directory, their publications, and expertise in international security topics. The administrator Web application provides a front end for the Directory administrators to modify information about Directory members. (See *Chapter 1* and *Appendix I* for details of the DCRIS project). This chapter concentrates on three specific elements of the public Web application - enough to illustrate the use of the concepts discussed in *Chapter 2*. *Chapter 4* shows how this framework can be used to design the administrator Web application, and how to increase functionality in both Web applications.

The three elements of the DCRIS project used to illustrate the views-based framework are the *'Member'*, *'Keyword'*, and *'Publication'* elements. A 'Member' is a researcher in international security who is part of the Directory. A 'Publication' is a book, conference paper, or journal article published by a Directory ‘Member’. A ‘Keyword’ is a broad area of study in which a ‘Member’ has expertise. Although it is not absolutely necessary for a ‘Member’ to have authored a ‘Publication’ in order for that ‘Member’ to be part of the Directory, in practice most ‘Members’ have authored at least one ‘Publication’. Similarly most ‘Members’ have at least one area of broad expertise. Part of the purpose of the Directory is to allow its users to see what ‘Members’ have published on the topic of international security, and their areas of expertise.

3.2 The DCRIS Applications Design Overview

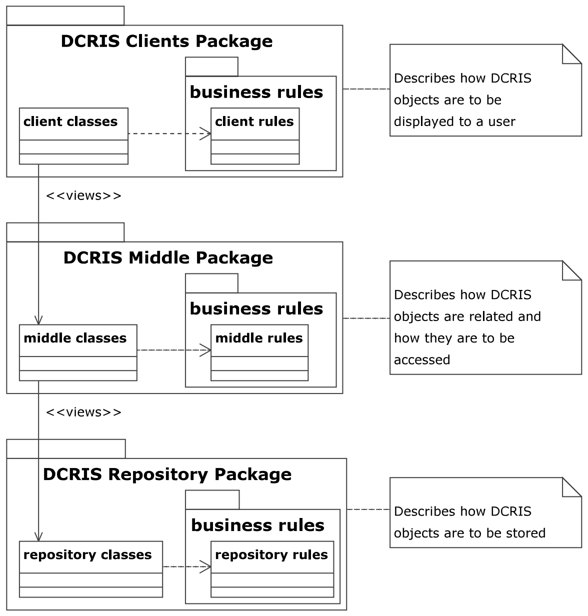


Figure 13 DCRIS Design Overview

Figure 13 shows the design overview for the DCRIS public Web application. Its structure is divided into the three levels discussed in Chapter 2, the repository package, the middle package, and the clients package. Each package contains a set of business rules and the classes that depend on those business rules.

The first step of the design process is to define the business rules classes. Business rules are derived from information gathered from a the different users involved in a project during the requirements phase. The designer must determine at what level to place each rule. In general, rules that determine how objects are to be displayed are placed at the clients package level. Rules that deal with how objects are to be stored are dealt with at the repository package level. All other rules fall into the middle package level by default. For example, a requirement that 'Publications' belonging to a Directory 'Member' must be output in a bulleted list below a 'Member''s name clearly belongs to the clients package business rules as it deals with how data is to be displayed. A requirement that 'Member' data consists of information such as surname, given name, mailing address, and email address clearly belongs to the repository package business rules as it deals with data that is to be stored. A requirement that 'Members' may be selected from the Directory on the basis of their expertise belongs to the middle package business rules by default: this requirement is about what is displayed, not how it is to be displayed, and therefore does not belong to the clients package; and it is about how 'Member' objects are to be selected, not stored, and therefore does not belong to the repository package.

Business rules are documented in English and their translation into code is done by hand by human programmers. As these rules change, the code representing them changes.

The next step in the design process is to define the package classes based upon the business rules in each package. As discussed in *Chapter 2*, each of these classes maps to one or more business rules within the same package. The classes in the middle package may view classes in the repository package and classes in the clients package may view classes in the middle package. Classes in the clients package may not view classes in the repository package directly. Further, any given class may be an aggregate of other classes in the same package, as will be demonstrated.

3.3 The DCRIS Repository Package Design

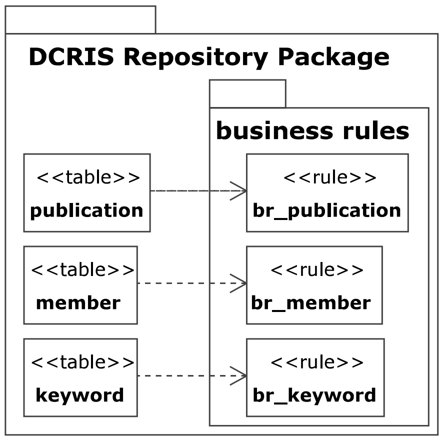


Figure 14 The DCRIS Repository Package

Figure 14 shows some of the classes that belong to the DCRIS repository package. The business rule classes here are the *br_publication*, *br_keyword*, and *br_member* classes. They represent the rules that define the permanent DCRIS project objects. Each of these classes is identified as a business rule class both by the fact that they are contained within a *business rules* package, and by the use of the `<<rule>>` stereotype. The classes *publication*, *keyword*, and *member* represent the permanent objects to which the previous rules refer. Each of these classes depends on these business rules. Should the rules change, the classes that represent them must change as well.

Generally, business rules define only the broad outlines of objects and the relationships between them and not the details on how these objects are to be realized. Thus, the business rules for a DCRIS 'Member' demand that the name, address, and telephone number of that 'Member' be recorded, but not how many characters should be set aside for each item. Another business rule states that a 'Member' must be uniquely identifiable without explaining how this identification is to be accomplished. A further business rule states that a 'Member' owns zero or more of the 'Publications' in the Directory, and has an expertise in at least one of the set of 'Keywords' in the Directory without describing how these relationships are to be expressed. This is not to say that business rules cannot go to this level of detail - a business rule could describe how a five digit customer number is used to identify a customer uniquely, for example - the point here is that business rules need not go to this level of detail. These things can be left up to the application designer.

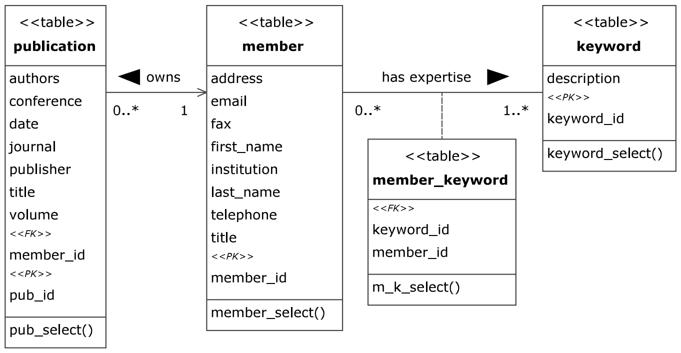


Figure 15 Repository Table Relationships

Figure 15 shows the classes that were designed as a result of the business rules referred to in **Figure 14**. Each class has its attributes and operations listed and the relationships between the classes are explicitly shown. Each class has been given a `<<table>>` stereotype to indicate that objects of this class are permanent. These tables happened to be implemented using SQL, although the choice of language can be deferred until the implementation stage of the project. The *publication*, *member*, and *keyword* classes contain attributes as explicitly defined by their associated business rules, such as the *publication authors*, *publisher*, and *title* attributes, and the *member last_name*, *first_name*, and *institution* attributes. These classes also contain attributes implicitly defined by their governing business rules. Each object in the *publication*, *member*, and *keyword* tables must be uniquely identifiable, so the attributes *pub_id*, *member_id*, and *keyword_id* were designed to do just that. Each of these attributes has a `<<PK>>` stereotype to indicate that they are primary keys, i.e., unique identifiers for the objects instantiated from each class. Because the business rules do not describe explicitly how these objects are to be identified uniquely, the designer has the leeway to make an appropriate choice.

The relationships between the *member*, *publication*, and *keyword* classes were also implicitly defined by their governing business rules. Because relationships between tables are typically defined by foreign key values, this technique was used here. Thus, the *publication* class contains a *member_id* attribute that is a foreign key value (as defined by the <<FK>> stereotype) relating each ‘Publication’ to a matching ‘Member’. (Although a many-to-many relationship between ‘Member’ and ‘Publication’ would better represent the possibility of more than one ‘Member’ contributing to a single ‘Publication’, the DCRIS administrators stipulated that because the ‘Publication’ list for each ‘Member’ was not exhaustive, they would select only representative publications that did not overlap between members.) This association between *member* and *publication* is named *owns*. The arrow beside the association name indicates how the association should be read: thus, a ‘Member’ owns ‘Publications’. The purpose of an association name is to clarify a relationship between classes [7]. In order to define the many-to-many relationship between the *member* and *keyword* classes a new table, *member_keyword*, was defined, containing foreign keys to the two related tables. The association between the tables is named “*has expertise*” with the arrow pointing towards the *keyword* table, thus reading: a ‘Member’ has expertise in many ‘Keywords’. Again, these design choices are arbitrary on the part of the designer - the important thing to recognize is that they fulfill the demands of the project’s business rules.

The operations defined for these repository classes are limited to selecting sets of objects from the repository (as in the operations *member_select*, *pub_select*, etc.). As discussed in *Chapter 2*, all other operations are left to the higher level packages.

3.4 The Middle Package Design

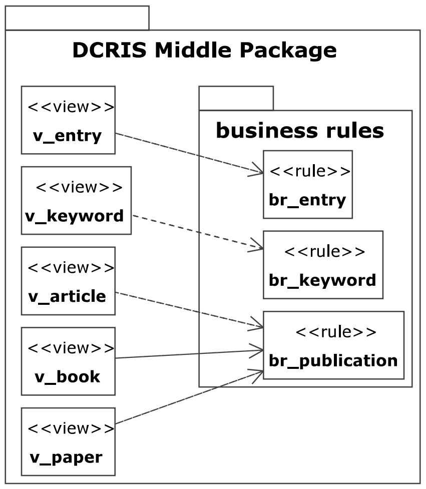


Figure 16 The DCRIS Middle Package

Figure 16 shows some of the classes that belong to the DCRIS middle package. As with the DCRIS repository, the middle package contains both business rules, identified with the `<<rule>>` stereotype, and view classes, identified with the `<<view>>` stereotype. Although both the middle package and the repository package contain business rule classes with the same names, such as *br_publication*, there are no conflicts between the classes as each occupies its own namespace, i.e. each is contained within a different package and can be referred to as such. As in the repository package, each of the middle package classes depend on business rules within the package. Thus, the *keyword* view depends on the business rules contained within the *br_keyword* class.

Three different views, *v_article*, *v_book*, and *v_paper*, depend on the *br_publication* class. There is no single *publication* view as there was a single *publication* table in the repository package. This shows that a 'Publication' owned by a Directory 'Member' must fall into one of three categories: it must be a journal article, a conference paper, or a book. Thus, although all 'Publications' may be stored together in the repository package (as in the *publication* table), the business rules of the middle package demands that each type of publication receive different treatment.

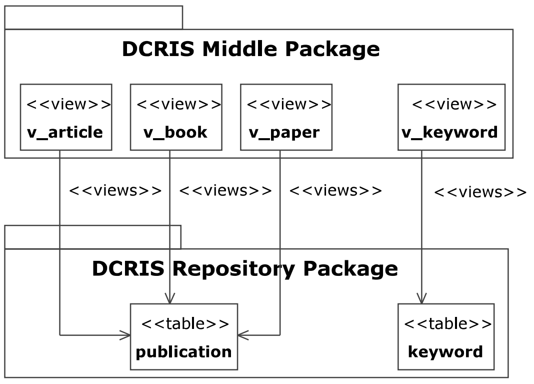


Figure 17 Middle to Repository Views

Figure 17 shows the relationships between the views of the middle package and the tables of the repository package. There is a one-to-one mapping between the *v_keyword* view and the *keyword* table and three different views of the *publication* table.

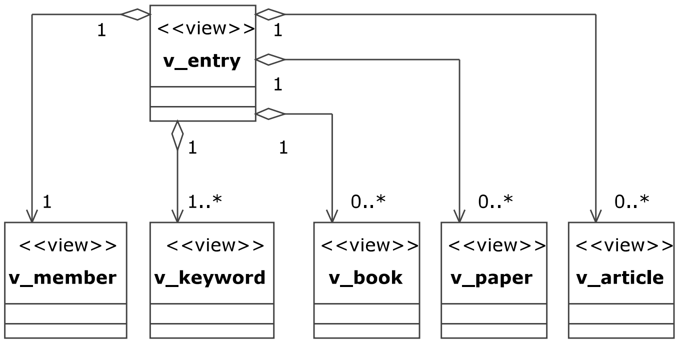


Figure 18 View Making up an Directory Entry

Figure 18 shows the structure of the *v_entry* view. The business rules of the *br_entry* class state that an entry in the Directory consists of member information, a list of keywords describing that 'Member's' expertise, and a list of books, conference papers, and journal articles written by that 'Member'. Thus the *v_entry* view is made up of the *v_member*, *v_keyword*, *v_book*, *v_paper*, and *v_article* views. The diamond shape at the *v_entry* end of each association in **Figure 18** is the UML symbol for aggregation, implying that the subordinate classes are part of a greater whole [16]. The multiplicity values displayed on the ends of each association show how many instances of each subordinate class are to be expected within a *v_entry* object. Thus every entry has one 'Member', one or entries from the set of ‘Keywords’, and may or may not have any publications at all, but more than one type of publication is allowed per entry.

The examples of the publications and the Directory entry demonstrate some of the flexibility of views within this framework. A view may view another class directly, but it may also do so indirectly. Thus complex views may be built up from simpler ones as in the *v_entry* example. Further, there may be multiple views of a class as in the example of the three different views of the *publication* table.

3.5 The Clients Package Design

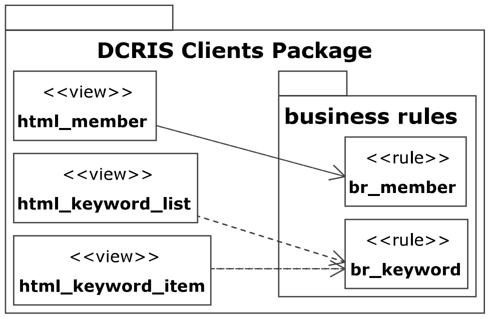


Figure 19 The DCRIS Clients Package

Figure 19 shows some of the classes that belong to the DCRIS clients package. As with the DCRIS repository and middle packages the clients package contains both business rules (identified with the `<<rule>>` stereotype) and view classes (identified with the `<<view>>` stereotype). As in the repository and middle packages, each of the clients package classes depend on business rules within the package. Thus, the *html_member* view depends on the business rules contained within the *br_member* class. Both *html_keyword_list* and *html_keyword_item* depend on the *br_keyword* business rules. *br_keyword* describes how a list of keywords should be displayed in HTML format.

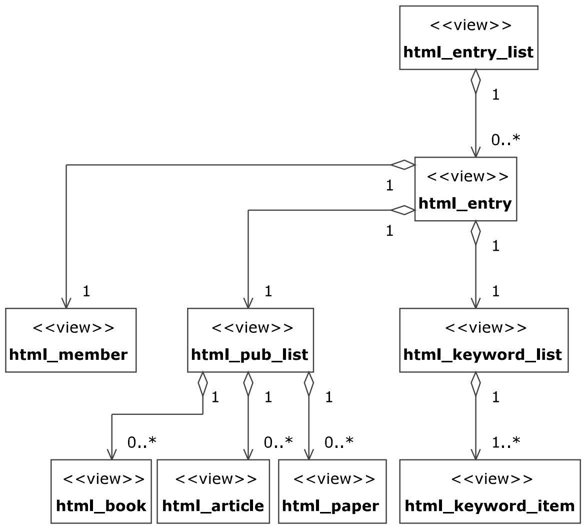


Figure 20 Views Making up an Entry List

Figure 20 shows how an HTML formatted list of entries is built from an aggregation of other views. Like the *v_entry* view in **Figure 18**, the *html_entry* view is made up component parts, the member, keywords, and publications views. *html_entry* then serves as an element of *html_entry_list*. As with the *v_entry* example, the multiplicities given with the associations between the views show how many sub-views each view may have.

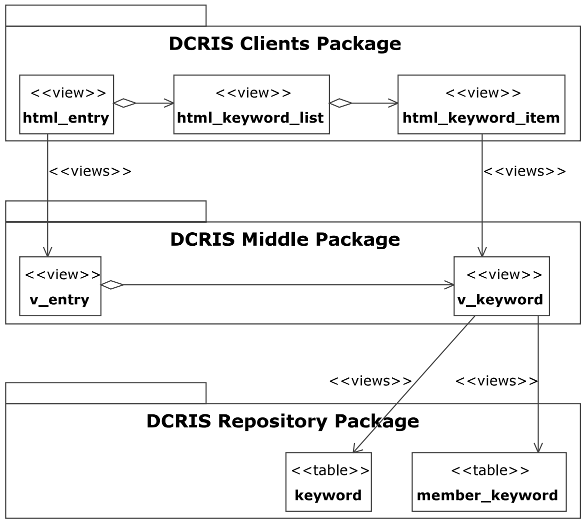


Figure 21 'Keyword' Example

Figure 21 shows how these views are tied together between the different packages. The example given is that of a list of ‘Keyword’ expertise held by a Directory ‘Member’. At the repository package level there are tables, *keyword* and *member_keyword* that hold the keyword data and how these keywords are associated with a particular ‘Member’ respectively. These two tables are viewed from the middle package by the *v_keyword* view. This view selects the appropriate keyword values from the repository tables for a particular ‘Member’. It does not, however, apply any display formatting to this data. *v_keyword* is also shown to be part of the *v_entry* view. (The rest of the elements of *v_entry* are not shown in this example). In the clients package the *html_keyword_item* view takes a *v_keyword* object and applies HTML formatting to it. The *html_keyword_list* wraps an HTML list definition about multiple *html_keyword_item* views, and then becomes part of an *html_entry* view.

Repository Package

keyword `<<table>>`

keyword_id	description
12	Arms Control

member_keyword `<<table>>`

keyword_id	member_id
12	20

Middle Package

v_keyword `<<view>>`

keyword_id	description	member_id
12	Arms Control	20

Clients Package

html_keyword_item `<<view>>`

```
<li>Arms Control</li>
```

html_keyword_list `<<view>>`

```
<h3>Keywords</h3>
<ul>
  <li>Arms Control</li>
</ul>
```

Output

Keywords

- Arms Control

Figure 22 ‘Keyword’ Display Example

Figure 22 shows a simplified example of how the different packages manipulate an object. In this example, a Directory ‘Member’ has only a single expertise, in this case ‘Arms Control’, and it is to be displayed as part of the ‘Member’s’ entry. At the repository package level this data is stored as part of the *keyword* and *member_keyword* table. At the middle package level the *keyword* table and *member_keyword* table are joined by the *v_keyword* view resulting in a list of all expertise for this

‘Member’ (whose *member_id* is 20). At the clients package level the keyword description alone is extracted from the *v_keyword* view and the HTML code for a list item applied to it by the *html_keyword_item* view. The result then has a title and the HTML code for an unordered list wrapped around it by the *html_keyword_list* view. The result is displayed by a browser as a bulleted list under the title “Keywords”. The actual implementation is more complex than this, but the implementation details are not important at this stage.

3.6 Discussion

The examples given in this chapter are specific instances of the general approach that this framework uses to promote separation of concerns during application design. This framework does indeed promote separation of concerns by its division into different packages:

- The repository package is concerned with persistent storage only. The objects stored in the repository are generally simple objects in comparison with the objects handled by the middle and clients packages.
- The middle package is concerned with selecting the desired objects from storage and constructing complex objects from simpler ones as necessary. The middle package views may also derive new attributes not contained within repository objects through aggregation or mathematical manipulation. The middle package views may also derive new objects based upon redefinition of repository objects.
- The clients package views are concerned with applying display formatting to the objects defined by the middle package views.
- Views within any package may have a `<<views>>` association to objects in another package, either directly or as a result of aggregating other views within the same package.

It is important to note that as well as emphasizing separation of concerns during the design process, a framework should improve with use. As more applications are designed using a particular framework, more elements of that framework can be recognized as common to the problem domain the framework addresses. These common elements can then provide more functionality to the next application built around that framework, thus making the framework ever more useful [16]. Further, the more these common elements are used the more they are tested, and thus the confidence in the reliability of these elements increases. The question of whether the design framework proposed by this paper improves with use is discussed in *Chapter 4*.

Chapter 4: Extending the Framework

One of the criteria for judging the effectiveness of a software design framework is to determine how amenable it is to change. Is the framework flexible enough to allow new elements to be added or old ones deleted? Is the framework a help or a hindrance to change in a software design [15]? Two specific examples of changes to the DCRIS public Web application will be examined in order to illustrate how the Web application framework accommodates change.

4.1 'Other Expertise'

The original design of the DCRIS Web application assigned a set of narrow expertise in security issues to a Directory 'Member'. These narrow expertise are called *'Supplementary Keywords'*, and there are a limited number of these to choose from. Examples of such expertise include 'Modern Peacekeeping' and 'Arms Control and Disarmament'. As wide is the range of expertise covered by these 'Supplementary Keywords', they cannot deal with all possible expertise. Some 'Members' wished to be able to display expertise that were not part of the existing lists. These new expertise were to be called *'Other Expertise'*. Each 'Other Expertise' was to be part of an expertise List in the same way that existing 'Supplementary Keywords' were.

To accommodate this request a number of design and implementation changes had to be made to the DCRIS public Web application. New business rules had to be created and views of these rules designed.

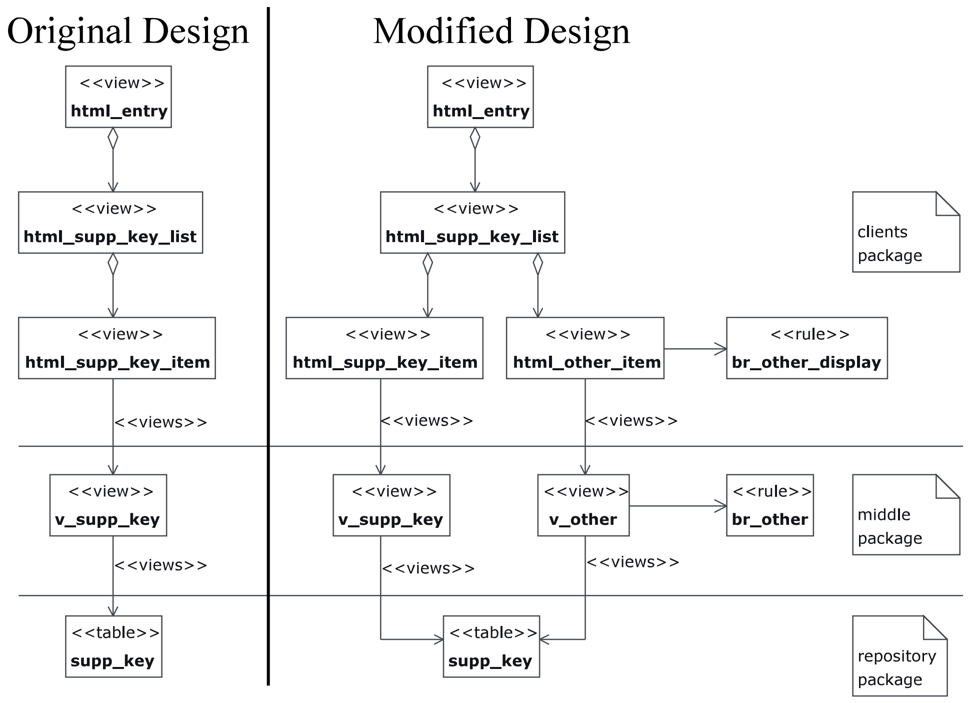


Figure 23 Addition of Other Expertise

Figure 23 shows the original design for handling 'Supplementary Keywords' and the changes made to that design to handle 'Other Expertise'. The left side of the diagram shows the classes in all three framework packages from *html_entry* view in the clients package down to the *supp_key* table in the repository package. This design to handle 'Supplementary Keywords' is very similar to the design to handle 'Keywords' as shown in *Chapter 3*. The right side of the diagram shows the additional classes necessary to handle the changes to design.

Examining the changes from the bottom up, it can be seen that the *supp_key* table in the repository package is now viewed by two different middle package views (*v_supp_key* and *v_other*), rather than by one (*v_supp_key*) as in the original design. The *supp_key* table was altered by the addition of a simple flag to each object (not shown) that categorizes the object as a 'Supplementary Keyword' or an 'Other Expertise'. Thus, each object in the *supp_key* table must be viewed by one of *v_supp_key* or *v_other*. This is similar to the way in which each object in the *publication* table is viewed as an article, book, or paper as discussed in *Chapter 3*. Note that *v_other* is dependent on a new business rule, *br_other*, that describes how an ‘Other Expertise’ should be dealt with. The business rule that *v_supp_key* depends on is unchanged.

Having extracted the appropriate data from the *supp_key* table, the *v_supp_key* view is itself viewed by the *html_other_item* view. This view, part of the clients package, applies display formatting to an ‘Other Expertise’. The view *html_other_item* depends upon the new *br_other_display* business rule that describes how a particular ‘Other Expertise’ should be displayed. ‘Other Expertise’ are to be displayed as part of a ‘Member’s’ list of narrow expertise, so the *html_supp_key_list* view has been expanded to include an association with the *html_other_item* view.

At the top end the *html_entry* view is unchanged. In the original design *html_entry* was associated with the list of narrow expertise in the *html_supp_key_list* view. In the altered design the fact that the elements that make up this list have changed is irrelevant to *html_entry*. The contents of the narrow expertise list are not the concern of *html_entry*, they are only the concern of *html_supp_key_list*. It is enough for *html_entry* to contain such a list as part of its display. One of the strengths of this views-based framework is that its elements can be isolated from one another, making changes easier to implement than if these items were tightly coupled. None of the other objects that make up a Directory entry such as member information, keyword expertise, and publications, are affected by the addition of the ‘Other Expertise’.

4.2 Other Language Support

The language of the user interface for the original DCRIS public Web application was English. When the DCRIS project began to receive funding from the Canadian federal government, one of the requirements for such funding was that the contents of the Directory be made available in French as well as in English. This applied to both the actual contents of the Directory, such as the lists of expertise, and the public Web application interface.

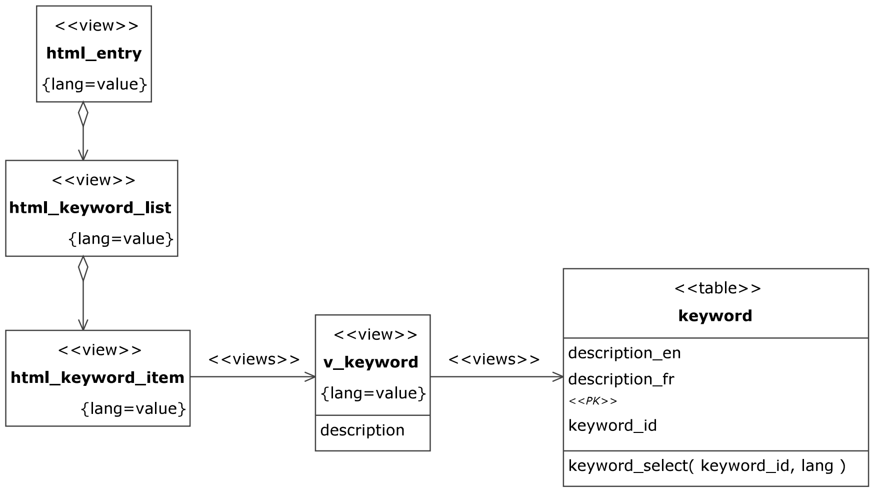


Figure 24 Addition of Language

Figure 24 shows how the public Web application design was altered to handle the use of French in displaying the broad expertise, i.e., 'Keywords', for a Directory 'Member'. Examining the changes from the top down, it can be seen that the changes to the clients package views *html_entry*, *html_keyword_list*, and *html_keyword_item* were minimal. It was sufficient to provide a flag named *lang* to indicate which language, English ('*en*') or French ('*fr*') should be used when displaying a Directory entry. Each view contains sub-views with matching *lang* attributes. The value of the *lang* attribute determines which language is actually displayed to the user.

The changes to the middle package view *v_keyword* were also small. It too was given a *lang* attribute that determined the language in which to display an expertise description, for example, choosing between displaying "Military History" or "Histoire militaire". Note that *v_keyword* still contained only one expertise *description*, but this description was in the language defined by its *lang* attribute.

The largest change was at the repository level. Each 'Keyword' now had to be stored in English in the field *description_en* and in French in the field *description_fr* in the *keyword* table. The operation that selects a keyword object from the *keyword* table, *keyword_select()*, was altered to accept a new parameter *lang*. This parameter identified which of the descriptions was to be extracted for any given object request.

It is important to understand that once it is decided which keyword description, English or French, is to be extracted from the repository, this keyword description is treated just as it was in the original design. A keyword description is, after all, simply a string of characters. The fact that the string is in English or French is not important so far as displaying it to the user goes. The same display rules are used whatever the language. The *lang* attribute that is now part of each view determines two things: the language of the information wrapped around a keyword description and the order in which the descriptions are to be sorted for display. Different languages may use different sorting orders.

This design is not limited to handling only English and French. By changing the value of the *lang* attribute and providing the appropriate keyword descriptions and text, this approach could handle any language.

4.3 Discussion

The time needed to design and implement the ‘Other Expertise’ portion of the DCRIS public Web application was approximately two hours. Once the actual representation of the ‘Other Expertise’ data and its place within the DCRIS repository package were determined the design and implementation of the appropriate views was very simple. This simplicity arose from a number of factors:

- As with most of the objects in the DCRIS public Web application the ‘Other Expertise’ needed only a limited number of views in the middle package. These views very similar to existing views (such as those for supplementary keywords), and those existing views were used as design and implementation templates.
- As with the middle views package there were a limited number of views to be designed and implemented for the client views package, and existing views were used as design and implementation templates.

- Because changes to the design views reached only as high as the list of narrow expertise, i.e., the *html_supp_key_list* view, no changes had to be made to the entry views or the Web pages that contained these views. The design and implementation updates were localized.
- Designing and implementing the ability to handle French was more involved, but mainly because of the author's extremely limited mastery of French as opposed to the flexibility of the design framework. Defining a language attribute for each view was simple, as was adding extra fields in the repository tables to contain the French versions of their text.

The addition of the 'Other Expertise' and multiple language capabilities to the DCRIS project illustrates the framework flexibility and how the framework accommodates change.

Chapter 5: Reusing the Framework

For a software design framework to be considered successful, it must be not only extensible, but reusable [16]. The DCRIS public Web application was used as a specific example of a successful design using a views-based framework. A second application using this framework will be examined in order to illustrate how the Web application framework accommodates reuse.

5.1 The DCRIS Administrative Web Application

The DCRIS administrative Web application allows the Directory administrators to update the contents of the Directory. The administrators may manipulate 'Member' information and modify expertise. The administrators may also view usage statistics for the public Web application. (See, for instance, the DCRIS Web applications use cases in *Appendix I*.) A Web-based application was felt to be the best option for the Directory administrators as it provided them with a number of benefits: a familiar, easy-to-use interface in the form of a Web browser; access to the Directory from anywhere there was Internet access; and rapid implementation. Further, many of the components of the public Web application could be reused in the administrative Web application.

The entire DCRIS repository package is shared between the two Web applications. Such sharing is easily accomplished as the repository is unaware of the fact that it is being viewed by any application, let alone two. The repository package does not need to make any special accommodation for the administrative Web application - the services it provides to the public Web application are sufficient for the administrative Web application as well.

The DCRIS middle packages for the public and administrative Web applications are almost identical. The objects used by the public Web application are a subset of the objects used by the administrative Web application. The only difference between the two packages is that the administrative application has an object containing data on the public Web application usage (*v_history*) that is not part of the public Web application. Casual users of the Directory do not have access to information about the usage of the Directory.

Similar to the relationship between the two applications' middle packages, the clients package of the public Web application is a subset of the clients package of the administrative Web application. The main differences lie in the fact that the administrative Web application allows the DCRIS administrators to edit the Directory data using Web-based forms.

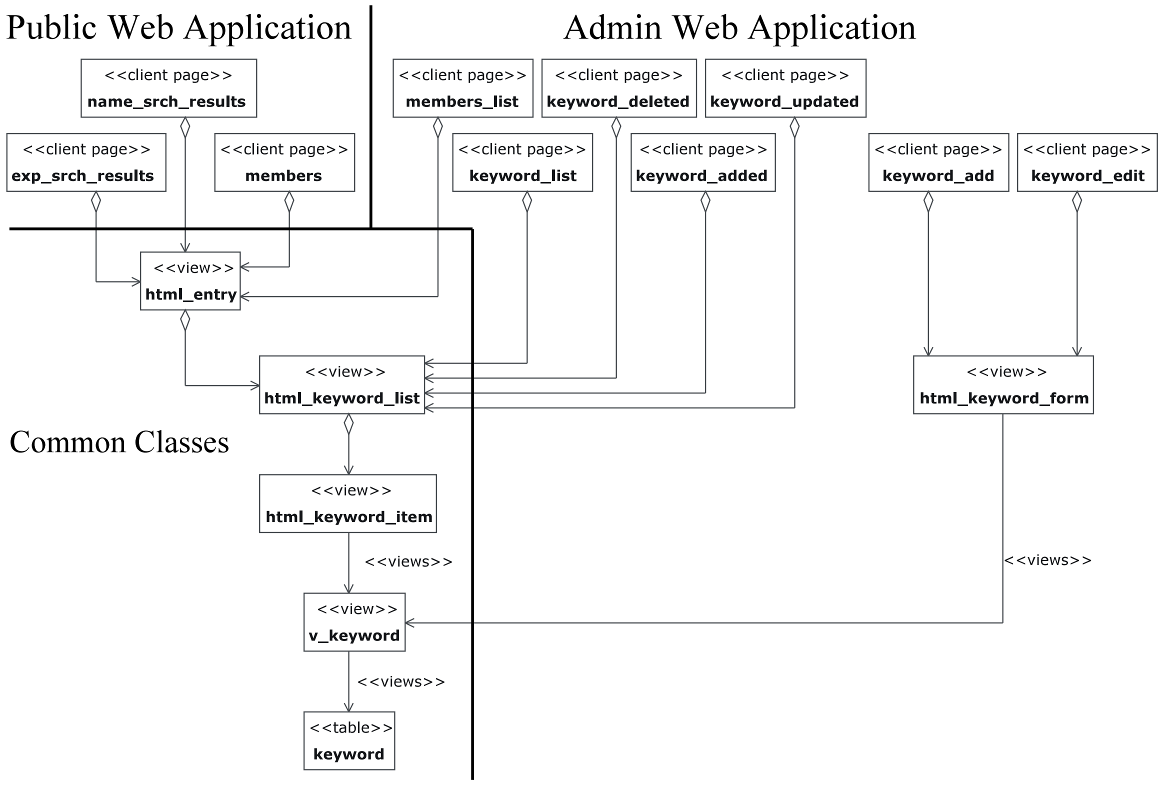

Public Web Application
Admin Web Application
Common Classes
<<client page>> name_srch_results
<<client page>> exp_srch_results
<<client page>> members
<<client page>> members_list
<<client page>> keyword_deleted
<<client page>> keyword_updated
<<client page>> keyword_list
<<client page>> keyword_added
<<client page>> keyword_add
<<client page>> keyword_edit
<<view>> html_entry
<<view>> html_keyword_list
<<view>> html_keyword_form
<<view>> html_keyword_item
<<views>>
<<view>> v_keyword
<<views>>
<<table>> keyword
<<views>>

Figure 25 Class Reuse Between DCRIS Applications

Figure 25 gives a simplified example of how the two applications share certain classes. The examples given are the classes that deal with 'Keywords'. The top left corner of the diagram, which is labeled *Public Web Application*, shows the classes that are unique to the public Web application. All of these classes are client pages, i.e., Web pages that are wrapped around 'Keyword' objects. The right side of the diagram, which is labeled *Admin Web Application*, shows the classes that are unique to the administrative Web application. The majority of the classes shown are client pages as well, with the exception of the *html_keyword_form* class. This class presents data for an individual 'Keyword' in an HTML form format – note that this is shared by two client pages. The bottom left corner of the diagram, which is labeled *Common Classes*, shows some of the classes that are common to the two applications. As noted earlier, these classes come from all three packages. The classes prefaced with *html_* come from the clients packages, the class prefaced with *v_* comes from the middle packages, and the *keyword* table comes from the repository packages. A similar situation applies to the other elements of the Directory: the 'Supplementary Keywords', 'Publications', 'Members', and 'Recommendations'.

5.2 Further Framework Reuse

Reusing this framework for the DCRIS administrative Web application was greatly facilitated by the fact that the two DCRIS Web applications are very similar. What, if any, reuse benefits accrue between two Web applications that lack the obvious common features of the DCRIS applications?

Another Web application designed by the author and used by the Department of Physics and Computing at Wilfrid Laurier University is called OwlData. It is used by the department's faculty, staff, and students to administer course information, student marks, TA assignments, and the like. Although it does not share any data with the DCRIS Web applications, it does share the three package structure and many of the client formats with the DCRIS Web applications.

A framework provides a conceptual model around which applications can be built. At the most fundamental conceptual level the framework tells a designer how to think about the design of an application. Completed applications provide examples and refined guidelines to the designer, thus improving the framework.

As the first concrete application of the views-based Web application framework, the DCRIS Web applications needed careful thought and some trial and error when applying the framework. Indeed, the framework was revised significantly during the process of building the DCRIS public Web application. Practical experience showed where the framework was lacking or needed improvement.

Applying the framework to the design of the OwlData Web application was significantly easier than to the DCRIS Web applications because of the experience gained through the earlier designs. First, at the conceptual level the experience of allocating elements of the DCRIS Web applications to the three packages - the repository package, the middle package, and the clients package - provided clear guidelines for the elements of the OwlData Web application. Thus, although the details differed, the basic approach to designing the tables and their relationships in the two repositories was the same between the two sets of applications. The basic approach to the materialization of objects from persistent storage in the middle package was the same. The design of views in the clients package and their relationships to objects in the middle package was similar between the two sets of applications.

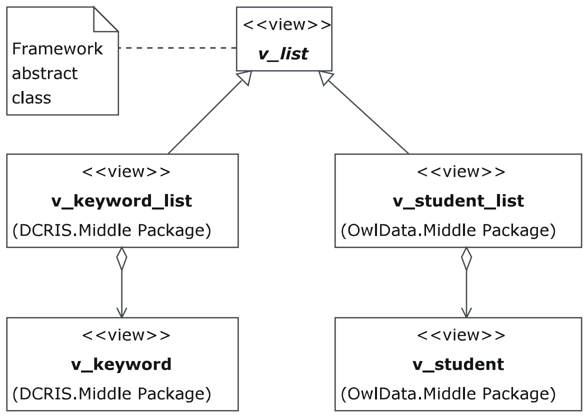


Figure 26 Framework Superclass Example

Once the framework's conceptual model has been applied to a Web application, the actual design must be laid out. In the design phase the framework provides common abstract classes that can be shared amongst applications. The DCRIS Web applications, for example, group together objects of the same type into lists within the middle package. **Figure 26** shows how both the *v_keyword_list* class from the DCRIS Administrative Web application and the *v_student_list* class from the OwlData Web application are specializations of the *v_list* abstract class.

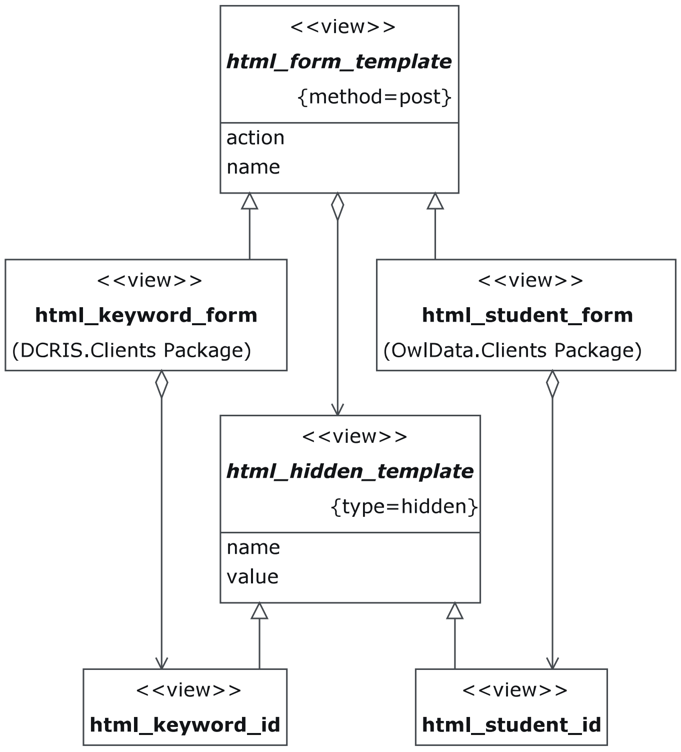


Figure 27 HTML Forms Templates

Further, the framework provides reuse in the form of code that is common to multiple applications. **Figure 27** shows how forms from both the DCRIS administrative Web application and the Owldata administrative Web application share common HTML forms templates. The abstract class *html_form_template* provides an outline for a standard Web form, including the access method and the action and name attributes. The forms from the DCRIS and Owldata applications use these attributes and provide their own values for them through parameters. Further, such forms may contain hidden values. The class *html_hidden_template* provides an outline for the standard HTML `<input type="hidden" ...>` code. As with the form template, the hidden type template contains name and value attributes that are inherited by the DCRIS and Owldata elements that use them. The existence of such templates greatly speeds up the design and implementation of the client pages that contain them. There are a large number of such common features between the DCRIS and Owldata Web applications in their respective clients packages.

Lastly, at a very practical level, having multiple applications based upon the same framework provides many opportunities for copying and pasting UML elements, documentation, and code from one application to another.

5.3 Discussion

Successful reuse is key to the success of the views-based framework for Web applications. The three level package structure has been applied to two applications that are very similar in content and a third that differs significantly from the first two. Although the use of common templates to speed up coding is extremely useful, more useful is the framework's approach to the separation of concerns. Proper use of each package in the framework imposes a coherent structure on a Web application design: object persistence at the repository package level, object manipulation at the middle package level, and object display at the clients package level. The hierarchical use of views within and between these packages promotes separation of concerns and reuse within an application. The examples illustrate that recognizing common features between applications allows the framework to improve with use.

Chapter 6: Implementing the Framework

The Web application framework described herein is modular, promotes separation of concerns and reuse, and can be represented using UML. This raises the question of how easy it is to implement. How do design elements map to implementation components? The storage, manipulation, and display of the 'Keyword' expertise in the DCRIS Web applications will be examined in order to illustrate one method of implementing the Web application framework.

6.1 The Repository Package

The two DCRIS Web applications, i.e., the public and administrative Web sites, share a common repository. This repository is implemented as an SQL database using Sybase SQL Anywhere Studio 7.04. The database is designed using standard techniques and for the most part is in Boyce-Codd Normal Form [24].

'Keywords' are stored in the *keyword* table, which consists of two attributes: *keyword_id*, a unique identifier for each 'Keyword' tuple and the primary key for the table; and *description*, a string containing the name of the expertise. The table has an index on the *description* attribute.

keyword_id	*description*
12	Military History
13	Civil-Military Relations
14	Defence Technology

Figure 28 Sample *keyword* Table Contents

Figure 28 shows a small subset of the *keyword* table data. The contents of the table may be searched, sorted, inserted, updated, and deleted using standard SQL '92 commands.

6.2 The Middle Package

The middle package views of the repository objects are based upon XML (Extensible Markup Language). XML is a set of rules and conventions for designing text formats for data that

- produces results that are easy to generate, read, search, sort, select, and project
- are unambiguous
- are extensible, allowing developers to create their own document definitions
- supports internationalization and localization
- is license-free, platform independent, and well-supported [21]

XML is ideal as a vehicle for the middle package objects. Object implementation is descriptive rather than code-based, making such implementations easily portable to multiple systems. Once such objects are defined using XML they can be manipulated by a wide range of software packages already available on the market. The Web application developer need only choose the software that fits their needs.

```
function keyword_xml( in @keyword_id tinyint default 0 )
returns long varchar
select
  f_tags(
  f_tags(keyword_id,'keyword_id')+
  f_tags(description,'description'),
   'keyword') into @string
  from keyword
  where keyword_id = @keyword_id;
return( @string )
```

Figure 29 SQL Script for the *v_keyword* View

Figure 29 shows the SQL script used to generate the XML tagged representation of a *keyword* table tuple. Although some SQL database packages now on the market can automatically generate XML tagged data from a database structure, SQL Anywhere Studio is not one of those packages. Thus all of the XML data in the DCRIS Web applications are generated from scripts written by the developer. Such scripts are trivial to write as the XML data generated closely follows the database table structures and all are very similar in design.

```
<keyword>
  <keyword_id>12</keyword_id>
  <description>Military History</description>
</keyword>
```

Figure 30 XML Representation of *v_keyword*

Figure 30 shows the XML data that results from executing the SQL script from **Figure 29** on the ‘Military History’ tuple. These objects may be part of a larger XML data structure.

```
<keyword_list>
  <keyword>
    <keyword_id>12</keyword_id>
    <description>Military History</description>
  </keyword>
  <keyword>
    <keyword_id>13</keyword_id>
    <description>Civil-Military Relations</description>
  </keyword>
  <keyword>
    <keyword_id>14</keyword_id>
    <description>Defence Technology</description>
  </keyword>
</keyword_list>
```

Figure 31 XML Representation of *v_keyword_list*

Figure 31 shows the XML representation of the *v_keyword_list* object. It is generated by simply wrapping appropriate tags around a list of *v_keyword* objects. The actual Keyword objects contained within a list depend on the context in which the list is generated. In a Directory Entry, a Keyword list contains all of the Keyword expertise held by a particular Member. In this case *v_keyword_list* is part of the *v_entry* object as illustrated in *Chapter 3*. In a different context all of the Keyword expertise available in the Directory is listed from which the user can choose.

The key to the use of XML as the basis for the middle package objects is simplicity. There is one, and only one, representation of a Keyword object in the middle package whether the keyword object stands alone or is part of a larger structure. This keeps the amount of code dedicated to retrieving objects from the repository to a minimum as an object can be copied out of the repository in only one way. As far as the middle package is concerned all data objects are atomic. The extraction of particular attributes from an XML object is the job of the views in the clients package. Nor does the middle package need to concern itself with sorting as that is also the job of the views in the clients package. The major job, then, of views in the middle package is to select data from the repository and wrap the data items in XML tags.

6.3 The Clients Package

The middle package objects are not usually suitable for display. They contain XML tags and attributes that should be invisible to the end user. It is the job of the views in the clients package to project and sort the contents of the middle package objects. This projection and sorting is implemented using Extensible Stylesheet Language Transformations (XSLT) stylesheets.

XSLT provides a descriptive rather than code-based method of converting XML data to some other format. As with the use of XML in the middle package, this means that the display transformations are easily portable to multiple systems. Once stylesheets are defined using XSLT they can be used by a wide range of software packages already available on the market. The Web application developer need only choose the software that best fits their needs.

In Web applications the client views transform the data from the middle package objects into HTML formatted data. In other contexts this data could be transformed into other display formats. The same ideas of simplicity and portability apply whether the end results are Web pages, charts, text, or word processing documents.

```
<xsl:template match="keyword_list">
  <ul>
    <xsl:apply-templates select="keyword" mode="item">
      <xsl:sort select="description"/>
    </xsl:apply-templates>
  </ul>
</xsl:template>

<xsl:template match="keyword" mode="item">
  <li><xsl:value-of select="description"/></li>
</xsl:template>
```

Figure 32 XSLT Template for a List of Keywords

```
<ul>
  <li>Civil-Military Relations</li>
  <li>Defence Technology</li>
  <li>Military History</li>
</ul>
```

Figure 33 HTML Code for a List of Keywords

- Civil-Military Relations
- Defence Technology
- Military History

Figure 34 Output of a List of Keywords

Figure 32 shows the portion of an XSLT stylesheet designed to transform a *v_keyword_list* object into an HTML formatted object. **Figure 33** shows the HTML code generated by applying the XSLT stylesheet to a XML formatted list of Keywords. **Figure 34** shows the output of this code as displayed in a Web browser.

Compare the final output of **Figure 34** with the sample XML data of **Figure 31**. The XSLT stylesheet has removed the *keyword_id* attribute from the final output and sorted the contents of the Keyword list by the *description* attribute. No sorting or projection code needed to be provided by the Web application developer - such capabilities are inherently part of the XSLT standard. The developer need merely indicate within the XSLT template the sorting and projection desired.

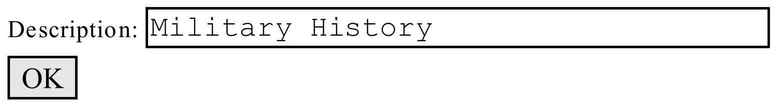
```
<xsl:template match="keyword_form">
  <xsl:call-template name="form"/>
</xsl:template>

<xsl:template match="keyword" mode="form">
  <xsl:apply-templates select="keyword_id" mode="hidden"/>
  <table border="0" cellpadding="0" cellspacing="2">
    <tr>
      <td>Description: </td>
      <td><input type="text" name="description" size="65"
        maxlength="65" value="{description}"/></td>
    </tr>
  </table>
</xsl:template>

<xsl:template name="form">
  <form method="post">
    <xsl:apply-templates select="@*" mode="copy-attributes"/>
    <xsl:apply-templates select="hidden/@*" mode="hidden"/>
    <xsl:apply-templates select="*" mode="form"/>
    <p><input type="submit" name="submit" value="OK"/></p>
  </form>
</xsl:template>

<xsl:template match="*|@*" mode="hidden">
  <input type="hidden" name="{name()}" value="{.}"/>
</xsl:template>
```
Figure 35 XSLT Template for a Keyword Form

```
<form action="Name_Search_Results.htm" method="post">
<input type="hidden" name="keyword_id" value="12">
<table border="0">
<tr>
<td>Description: </td>
<td><input type="text" size="65" maxlength="65" name="description"></td>
</tr>
</table>
<p><input name="submit" type="submit" value="OK"></p>
</form>
```
Figure 36 HTML Code for a Keyword Form


Description: Military History

OK

Figure 37 Keyword Form Output

Figure 35 is an example of the XSLT template used to create an HTML input form for a Keyword. **Figure 36** shows the resulting HTML code, and **Figure 37** shows the output as displayed by a Web browser. Note that all the attributes of the *v_keyword* object are used by the form, though the *keyword_id* attribute is hidden from the user's view. Part of the XSLT template contains references to generic template named *form* and a mode named *hidden*. These are examples of the reusable elements mentioned in *Chapter 5*. These particular stylesheet elements can be used by any XML data that needs to be presented as a generic HTML form. This is an example of how repeated use of a framework can lead to the discovery of common features between applications, and how these common features can be implemented in shared templates or libraries.

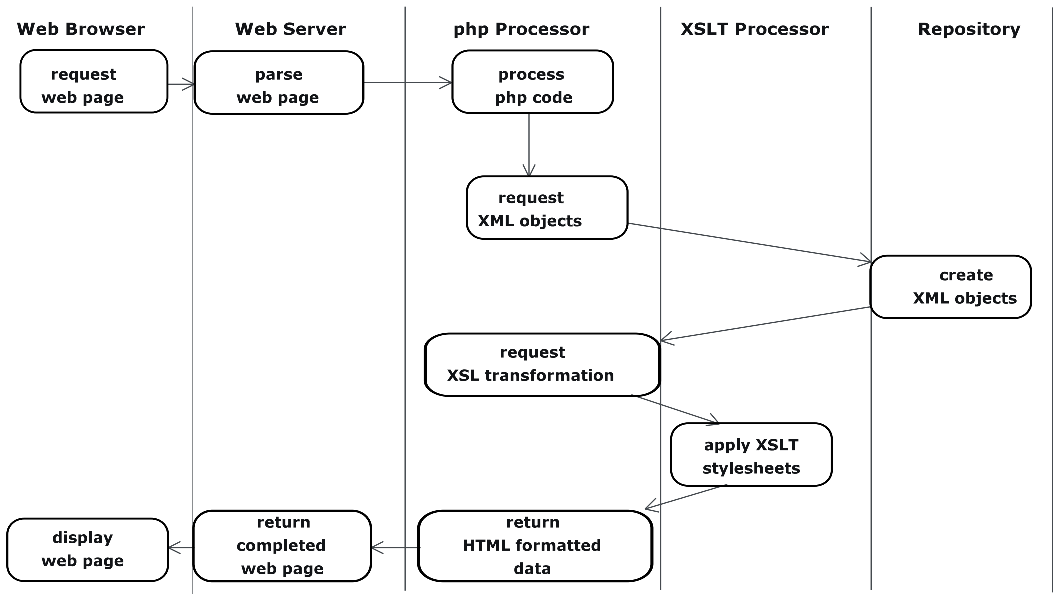


Figure 38 Web Page Request Processing

Figure 38 shows the steps necessary to display a Web page using this implementation. Once a Web browser requests a Web page, the Web server (currently Apache) parses that page looking for Web application server code. If found, this code is passed to the Web application server for execution. The current DCRIS implementations uses PHP: Hypertext Preprocessor. If required the Web application server requests XML formatted objects from the repository. It then passes these objects to a XSLT processor which applies the desired XSLT stylesheets to the XML formatted data to produce

HTML formatted objects. The Web application server then returns these HTML formatted objects to the Web server, which sends the completed Web page to the Web browser. The only code that needs to be written by the application developer is the Web application server code that makes calls to the repository and to the XSLT processor.

6.4 Further Implementation Notes

The DCRIS system has been implemented on a number of different systems. The original implementation was done with Perl, but as a prototype it did not follow entirely the framework design as laid out in this thesis. The first production implementation used Janna System's LivePage Web application server. LivePage took the place of PHP in the immediately preceding description. The move to PHP was prompted by the withdrawal of LivePage from the market. The change from LivePage to PHP was very easy. There were no code changes at the repository level as none of the existing SQL functions and procedures needed to be altered. None of the XSLT stylesheets needed to be changed as the stylesheets are independent of the XSLT software that processes them. Only the Web application server code imbedded in the DCRIS Web pages needed to be altered as LivePage and PHP use a different syntax to make SQL and XSLT calls. This trouble-free change-over demonstrates that the key strengths of this framework, separation of concerns and reuse, apply even at the implementation level. The repository deals with XML-formatted data objects, the XSLT processor deals with applying display formatting, and the Web application server deals with selecting the objects to be processed.

Chapter 7: Conclusions and Future Work

This thesis has described a views-based framework for designing Web applications. It focused on three main points:

- how views promote the separation of concerns into persistence, manipulation, and presentation
- how a framework promotes reuse
- how UML and extensions to it provide a representation of views and the framework

In support of these points this thesis has:

- discussed the concepts behind views and frameworks
- demonstrated how a combination of the two was used to design and implement an actual Web application
- showed how such a framework could be extended and reused
- gave sample representations of the framework using UML extensions
- showed how such a framework could be implemented

7.1 Discussion

In our opinion, the approach to Web application design discussed in this thesis looks promising and would be worth further refinement. The proposed framework is a useful way to conceptualize and implement a Web application. The three packages of this framework promote separation of concerns in both design and implementation by clearly separating data storage, extraction, and presentation. This separation is accomplished by using views that are each part of one of the three packages that make up the framework. The use of views promotes reuse as a given view can be reused within a particular Web application or within a similar application. By clearly separating different elements of the application, each element can be treated as an object and thus have object-oriented software design techniques applied to them. By using a hierarchy of views certain changes can be made to one level of views without necessarily forcing changes in the views above or below that level.

From an implementation standpoint breaking an application up into distinct views through the use of XML and XSL allows elements of an application to be implemented and tested separately from other elements. By using XML as a intermediary between a data source and an output format, anything from which XML can be generated can be used as a data source, such as a relational database, text file, or executable. Further, output formats may be in forms other than HTML, such as text or SVG [45].

In our opinion UML, with appropriate extensions, is a suitable language for representing this Web application design framework. It is already in broad use in object-oriented software design and is thus familiar to many software designers. It is extensible, and this thesis has shown how it can be further extended to handle views and elements particular to Web applications.

Lastly, this Web application design framework has been in use for over two years for various Web applications in the Department of Physics & Computing at Wilfrid Laurier University. The framework has evolved over that time, but has shown its worth within that environment.

7.2 Future Work

There are a number of areas of views-based frameworks for Web application design which deserve further investigation.

The use of metadata would allow an application framework to be made even more generic, and therefore more extensible and reusable, than the one discussed here. The key to using metadata is to detail a method for standardizing the description of an application's data and how it is to be manipulated and displayed.

Formalizing a grammar for representing business rules would allow the creation of applications that dynamically alter themselves to fit changes in the rules that define them. Indeed, such a set of formalized rules could be thought of as a complement to the previous point about metadata. Because rules describe the structure of the data in an application, they are metadata about metadata.

This thesis gave examples of Web applications that were implemented using relational databases for their data sources, but claimed that other technologies could be used in their place. This claim could be followed up with examples that use other technologies to determine their degree of independence. Further, the claim has been made that HTML is not the only output format supported by this framework. Some work involving output formats such as text or Scalable Vector Graphics (SVG) [45] has already been done, and should be followed up by further work.

The issue of performance needs to be investigated. Although the real-world applications described in this thesis perform well anecdotally, there is no evidence of how this approach would scale up with larger implementations. Separation of concerns is very useful in design terms, but are there trade-offs in terms of performance? Comparisons of different implementation methods at different scales would help answer this question.

We believe that the approach to Web application design promoted in this thesis has proven itself useful in a practical way for the case studies discussed herein and points the way to a wider range of design and implementation possibilities.

Appendix I: The Directory of Canadian Researchers in International Security (DCRIS)

I.1 DCRIS Description

DCRIS supports research on all aspects of international security by identifying Canadian researchers and their specific areas of expertise, and by facilitating communication to, within, and from the community of Canadian researchers in international security.

DCRIS Members are recognized Canadian researchers in all areas of international security. The Directory lists them with institutional affiliations and addresses, as well as keywords describing areas of research expertise and citations of up to three recent publications or projects. The Directory is searchable by name, institutional affiliation, or keyword (or combination of keywords). DCRIS also lists links to Web application addresses as recommended by Members. Recommendations are searchable both by name and by URL. Although efforts are made to keep the list accurate and up-to-date, DCRIS cannot guarantee either availability or relevance.

The DCRIS database is maintained on the Wilfrid Laurier University site on the world-wide web. The address for direct access is http://www.wlu.ca/DCRIS. Access is also available from the Web application of the Laurier Centre for Military Strategic and Disarmament Studies, http://www.wlu.ca/~wwwmsds. Hard copies of the Directory are printed from time to time, and are available to Members and others upon request.

DCRIS Members are Canadian researchers, defined as individuals who have produced recognized research and who are Canadian or work in Canada. Normally, government employees and students are not considered to be researchers, but exceptions can be made; students at the PhD level are especially invited to apply. Decisions regarding the Directory and listings in it are be made by the Directory Administrators, Dr. James Keeley of University of Calgary and Dr. Marc Kilgour of Wilfrid Laurier University. Continuing support of DCRIS is provided by the Laurier Centre for Military Strategic and Disarmament Studies. DCRIS was created and expanded with support from the International Security and Outreach Program (ISROP) of the Department of Foreign Affairs and International Trade, Canada.

The DCRIS Administrator is

D. Marc KILGOUR, Ph.D.
Director, Laurier Centre for Military Strategic and Disarmament Studies
Wilfrid Laurier University
WATERLOO, Ontario N2L 3C5
CANADA
Tel: (519-)884-1970 or -0710, ext. 4208
Fax: (519-)886-5057
Email: mkilgour@wlu.ca

The DCRIS data is stored in a SQL database, as described in *Appendix III: The DCRIS Repository*. There are two separate DCRIS Web applications: the public Web application that gives access to Directory information, and the administrative Web application that allows the Directory administrators to update its contents. These are described in *Appendix V: The DCRIS Public Web application* and *Appendix VI: The DCRIS Admin Web application* respectively. *Appendix IV: The DCRIS Views* describes the various views of the Directory data that are at the core of the Web application designs.

I.2 DCRIS Entries

A typical Directory entry contains information about a Member: their name, institutional affiliation, telephone number, fax number, and email address. Each Member has areas of broad expertise in military and security issues: these are listed as a Member's Keywords. Members also have other areas of narrow expertise: these are called Supplementary Keywords and are organized by topic into Lists. Lastly, Members may have Publications related to military and security topics. **Figure 39** shows a typical Directory entry.

KILGOUR, Dr. Marc

Institution:	Wilfrid Laurier University
Address:	Laurier Centre for Military Strategic and Disarmament Studies Wilfrid Laurier University Waterloo ON N2L 3C5
Telephone:	(519) 884-1970 x4208
Fax:	(519) 886-5057
Email:	mkilgour@wlu.ca

Keywords

- Arms Control and Non-Proliferation Studies
- Peacekeeping/Peace Implementation
- Strategic Studies

Lists

- Arms Control and Non-Proliferation Studies
    - Confidence Building Measures
    - Monitoring, Verification, Compliance, Detection, and Enforcement
    - Quantitative Analyses
- Geographic Focus
    - Global or General
- Peacekeeping and Peace Implementation
    - Modern Peacekeeping
- Strategic Studies
    - Approaches to Peacekeeping
    - Arms Control and Disarmament
    - Theoretical Aspects
    - Weapons Proliferation
- Weapons Systems
    - Conventional - General
    - Nuclear - General
    - Nuclear - Safeguards

Publications

- Rudolf Avenhaus, Morton Canty, D. Marc Kilgour, Bernhard von Stengel, and Shmuel Zamir, "Inspection Games in Arms Control." *European Journal of Operational Research*, 90, 3, 1996
- Kilgour, D. Marc, Liping Fang, David Last, Keith W. Hipel, and Xiaoyong Peng, *Peace Support, GMCR II, and Bosnia*, Analysis for Peace Operations, Canadian Peacekeeping Press, 1998
- Frank C. Zagare and D. Marc Kilgour, *Perfect Deterrence*, Cambridge University Press, 2000

Figure 39 Typical DCRIS Entry

I.3 DCRIS Use Cases

The DCRIS Web applications' actions can be described using UML use cases. There are two actors in this view of the DCRIS Web applications. The *User* represents a general user's access to the Directory through the public Web application. The *Administrator* represents a DCRIS administrator's access to the Directory through the admin Web application. **Figure 40** shows the DCRIS actors and their corresponding use cases.

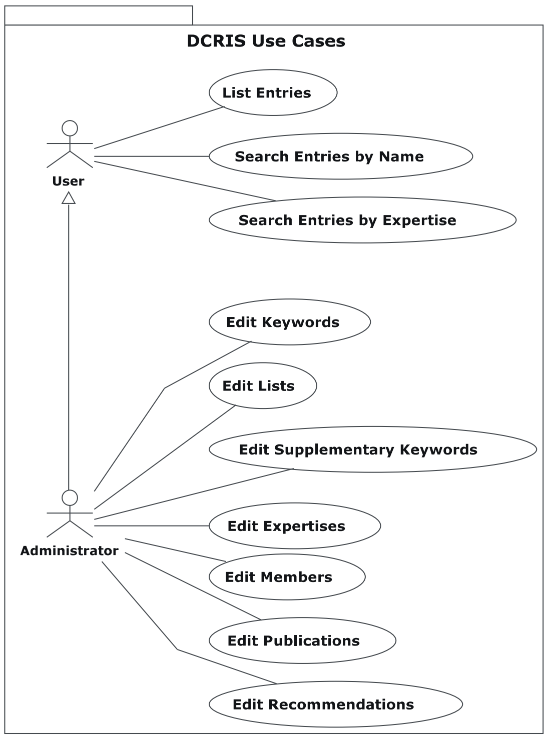


Figure 40 DCRIS Use Cases

The *Administrator* is a special type of *User* who inherits all of the *User*'s capabilities while having extra privileges denied to a *User*. In practical terms this simply means that the DCRIS public Web application makes no distinction between general users and administrators. There are no login procedures on the public Web application, and a User may not make any changes to the contents of the Directory. An *Administrator* must login to the admin Web application and has various options for editing the contents of the Directory.

Each of these use cases maps to one or more web pages in the appropriate DCRIS Web application. These mappings are covered in more detail in *Appendix V: The DCRIS Public Web application* and *Appendix VI: The DCRIS Admin Web application*.

The User Use Cases

List Entries

Displays all DCRIS Entries.

Search Entries by Expertise

Subsets of entries may be displayed based upon a Member's expertises.

Search Entries by Name

Subsets of entries may be displayed based upon a Member’s name, an Institution's name, or both.

The Administrator Use Cases

Edit Expertises

Various expertises may be assigned to a Member.

Edit Keywords

Add, update, or delete a Keyword.

Edit Lists

Add, update, or delete a List.

Edit Members

Add, update, or delete a Member.

Edit Publications

Add, update, or delete a Publication.

Edit Recommendations

Add, update, or delete a Recommendation.

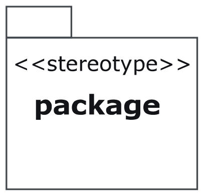
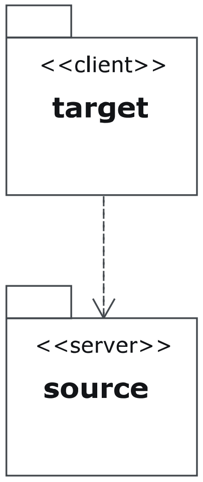
Edit Supplementary Keywords

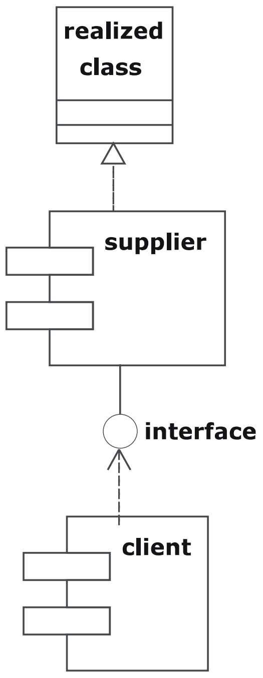
Add, update, or delete a Supplementary Keyword.

Appendix II: Using Unified Modeling Language (UML)

II.1 Basic UML Elements

UML provides semantics and constraints for modeling software architectural elements. This paper uses a number of UML elements to graphically illustrate its points. This appendix describes these graphical elements in detail [7].

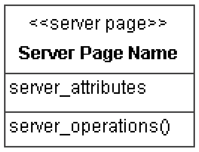
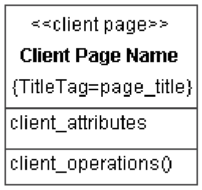
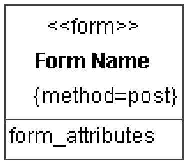
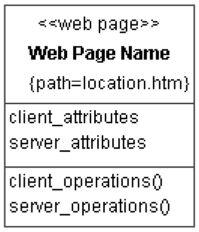
`<<stereotype>>`	A typical UML graphical element can represent a wide rage of things, so long as those things fall within certain broad categories (e.g. elements may be static or dynamic). A *stereotype* classifies and defines a particular model element. Thus, a class model element representing a table in an SQL database may be given a `<<table>>` stereotype to identify it as such.
	A package is a high-level container for UML elements, and is most often used to break a large model up into smaller, more manageable pieces.
	Packages may be related by a *dependency*. In this diagram, the dashed line between packages indicates a dependency. A dependency relationship specifies that a change in the source may affect target that uses it, but not necessarily the inverse. In a views context then, a change in a viewed object affects a view of it, but a change in a view does not affect the object being viewed. Dependencies may be stereotyped.

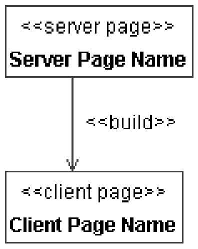
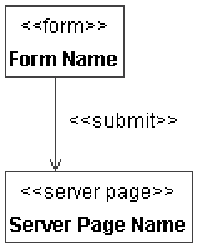
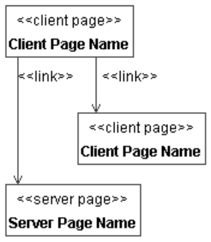
	A *component* represents the implementation elements of a model. A component maps to a real-world element such as a text file, xml data, a java class file, a code library, or a database. Every class in a software system is realized (implemented) by at least one component, and may be realized by more. Components may depend on other components. In such dependencies a client component depends on a supplier component, and connects to that component through an *interface* (also called a *specification*). For example, in a software system implemented using C++, a component represents a source code (.cpp) file, and an interface represents a header (.h) file.

II.2 The WAE Model

Building Web applications with UML by Conallen [6] describes modeling Web applications using the Web application Extension (WAE) for UML. Conallen provides semantics and constraints for modeling web architectural elements.

The foundation element of Web applications is the web page. In WAE, a number of different stereotypes are applied to a web page depending on the role it plays in a Web application. Web pages can be subdivided into various components depending on the functions of those components. (In implementation terms each of these components can be mapped to HTML elements.) There is a large set of WAE items defined for classes, components, packages, and other UML elements. Conallen suggests class descriptions for a number of other web elements not listed here, such as frames, client-side scripts, and java servelets. Further, he demonstrates the use of sequence diagrams, component diagrams, and other UML elements for describing the form and function of a Web application. However, the narrow subset of WAE elements used in this paper is sufficient as a basis for the topics under discussion. The following table present only those WAE items used in this paper.

 <<server page>> Server Page Name server_attributes server_operations()	A `<<server>>` page is a web page that contains operations executed by a web server or application server. The server page itself cannot be viewed - instead the execution of its operations builds a client page that is then viewed by a browser.
 <<client page>> Client Page Name {TitleTag=page_title} client_attributes client_operations()	A `<<client page>>` can be thought of as a virtual page - it has no existence until it is built by a server page. It may contain data, images, client operations, and further WAE elements such as forms. A client page 'TitleTag' tagged value is the title of the client page when viewed by the end user.
	A `<<form>>` is an element of a web page or client page. In implementation terms it is a set of HTML input fields surrounded by a `<form>` tag. It has no operations since these are contained within its owning page. In UML forms are shown as part of their owning page through an aggregate association. The 'method' tagged value is the HTML submission method used by the form. Forms cannot stand alone and must be part of a client page or a web page. One client page or web page may contain many forms.
 <<web page>> Web Page Name {path=location.htm} client_attributes server_attributes client_operations() server_operations()	A `<<web page>>` is an HTML formatted text file. It may contain client or server logic. The 'path' tagged value is the page's location with respect to the root of the web server.

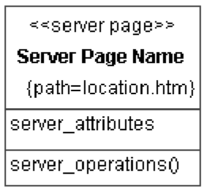
 <<server page>> Server Page Name <<build>> <<client page>> Client Page Name	A `<<builds>>` association connects a server page to the client page it build when the server page's logic is executed. Because server pages may contain conditional logic, one server page may build more than one class of client pages. (Note that this stereotype is called both `<<build>>` and `<<builds>>` at various places in Conallen [6]. This paper uses `<<builds>>` throughout.)
 <<form>> Form Name <<submit>> <<server page>> Server Page Name	A `<<submit>>` association shows the connection between a form and the server page its data is submitted to. A form may have only one `<<submit>>` association. A `<<submit>>` may have tagged value parameters that are passed along with the submission request.
 <<client page>> Client Page Name <<link>> <<link>> <<client page>> Client Page Name <<server page>> Server Page Name	A `<<link>>` association shows hyperlinks between pages and is mapped to the HTML anchor element. Web pages and client pages may link to all other page types. (Not all server pages need form input data in order to be executed and may thus be linked to directly). Note that the link from a client page to another client page, and the server page that builds this second client page are essentially the same. A `<<link>>` may have tagged value parameters that are passed along with the page request.
`<<input>>`	An `<<input>>` attribute stereotype maps to the HTML <**input**> tag. It may have the tagged values: *Type* - The type of input control to be used: one of Text, Number, Hidden, Password, Checkbox, Radio, Submit , and Reset . *Size* - Specifies how large an area to allocate on screen, in characters. *Maxlength* - The maximum number of characters the user can input..

`<<select>>`	A `<<select>>` attribute stereotype maps to the HTML <**select**> tag. It may have the tagged values: *Size* - Specifies the number of lines of text to display at once on screen. *Multiple* - Specifies that multiple values may be selected at once.
`<<textarea>>`	A `<<textarea>>` attribute stereotype maps to the HTML <**textarea**> tag. It may have the tagged values: *Rows* - The number of visible text lines. *Cols* - The visible width of the control in average character widths.

Table 1: WAE Elements

II.3 Extending the WAE Model

The WAE classes already discussed do not explicitly make use of the views concept. One of the purposes of his paper is to demonstrate how the WAE model can be extended to include views. This extension greatly enhances the usefulness of the WAE model. This section discusses the details of this suggested extension. The following table shows changes and additions to the WAE elements from **Table 1** and how views can be related to these elements.

 <<server page>> Server Page Name {path=location.htm} server_attributes server_operations()	The original definition of a `<<server page>>` stereotype lacks the 'path' tagged value that is part of the `<<web page>>` stereotype. This identifies the location of the server page for implementation purposes. As well, in order for a form to submit its values it must know the address of its submission target (the form's 'action'). The 'path' provides this address.

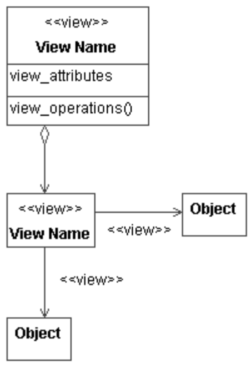
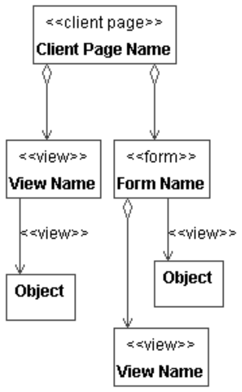
`<<view>>`	The `<<view>>` association implies that one object views another. A view association is a one-way association, with the viewed object unaware of the existence of the viewer. In UML this is shown as an association that is navigable only from the viewer to the viewed object. In this sense, forms are really just a special type of view.
 <<view>> View Name view_attributes view_operations() <<view>> View Name <<view>> Object <<view>> Object	A `<<view>>` class represents an object that views another object. A view may have attributes an operations, may view more than one object, and may have an aggregate association with other views. One object may be viewed by more than one view.
 <<client page>> Client Page Name <<view>> View Name <<form>> Form Name <<view>> <<view>> Object Object <<view>> View Name	A client page may be made up of views, including forms, which are special types of views. In their turn, forms may view objects directly or be made up of other views. Server pages may not contain views. They may contain logic that invokes a view, but the view is not considered part of the server page. A view can be part of a client page that is built when the server page's logic is executed. Web pages may contain views, but these views are static views, since web pages are hard coded. Thus a form that is part of a web page is a static form, unlike a form that is part of a client page.

Table 2: WAE Extensions

These extensions are not numerous, but they are significant. Allowing views to be part of the WAE model means that web pages can be treated as shells for containing views. This makes use of the concept of separation of concerns. With this approach, a view and a web page become different things in design terms. A web page’s various elements (header, links, style, etc.) can be dealt with separately from the views contained in the web page. Similarly a view can be designed without worrying about the web page that will eventually contain it. This separation of has a significant impact on reuse. If a number of different web pages display the same information, then they can be designed and implemented to make use of the same views. Should those views change, the web pages containing them need not be altered as the views they contain are updated automatically.

Appendix II: The DCRIS Repository

The DCRIS Repository is implemented using an SQL database. The database product used is Sybase Adaptive Server Anywhere 7.0. **Figure 41** is a UML diagram of the overall database design showing the database tables, attributes, and relationships. These are explained in more detail in the rest of the Appendix.

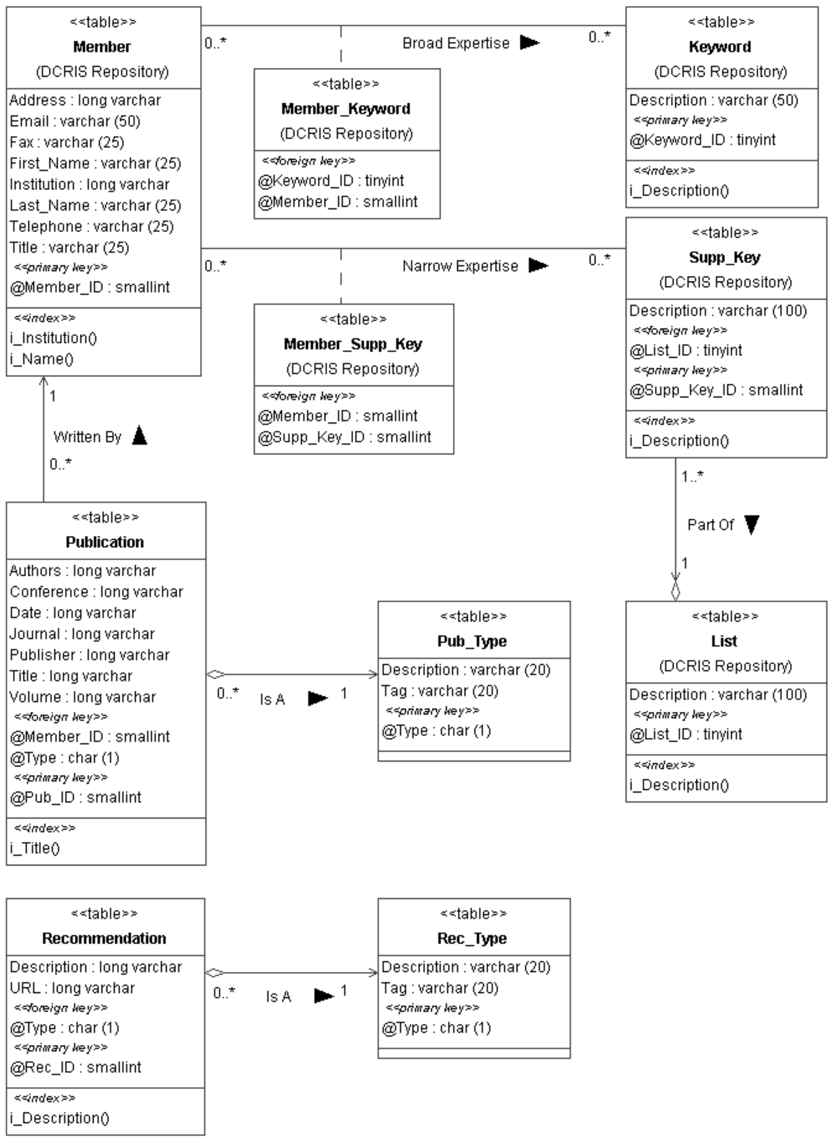


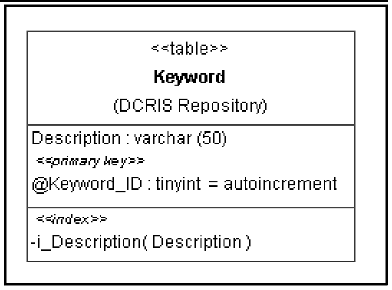
Figure 41 DCRIS Repository Overview

The database is designed using standard techniques and for the most part is in Boyce-Codd Normal Form. (Its few deviations from BCNF are done for simplicity's sake. In the *Member* table, for example, the *Institution* field contains a few instances of repetitive data where more than one DCRIS member serves at the same institution. In the *Publication* table, the *Publisher* field contains a few instances of repetitive data where more than one publication has the same publisher. Given the relatively small number of such repetitions, it was decided during the design process that no important functional purpose would be served by breaking these elements out into separate tables). The Repository consists of two sections: recommendations, as contained in the *Recommendation* and *Rec_Type* tables; and entries, consisting of the contents of the remaining tables. Recommendations stand alone as they are not connected to any particular DCRIS member once they are submitted to the Directory. The other elements of the Directory - the expertises and publications - are related to DCRIS members.

III.1 DCRIS Tables

This section describes the tables and attributes that make up the DCRIS database.

Keyword Table



<<table>>
Keyword
(DCRIS Repository)
Description : varchar (50)
<<primary key>>
@Keyword_ID : tinyint = autoincrement
<<index>>
-i_Description(Description)

Figure 42 *Keyword* Table

Contains categories of broad security expertise.

Attributes

Description: varchar(50), not NULL, unique

The Keyword description.

Keyword_ID: tinyint, autoincrement, primary key

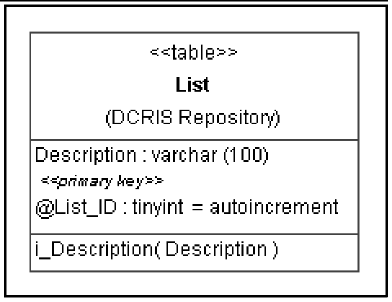
Arbitrary unique identifier for a Keyword. Autoincremented upon insertion.

Indices

i_Description

An index on the Keyword *Description*.

List Table



<<table>>
List
(DCRIS Repository)
Description : varchar (100)
<<primary key>>
@List_ID : tinyint = autoincrement
i_Description(Description)

Figure 43 *List* Table

Contains titles of areas of narrow security expertise.

Attributes

Description: varchar(100), not NULL, unique

Supplementary Keyword List description.

List_ID: tinyint, autoincrement, primary key

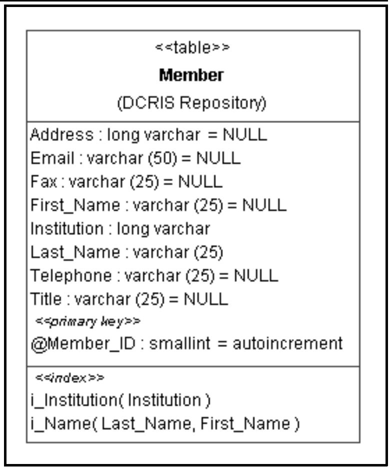
Arbitrary unique identifier for a List. Autoincremented upon insertion.

Indices

i_Description

An index on the List *Description*.

Member Table



<<table>>
Member
(DCRIS Repository)
Address : long varchar = NULL
Email : varchar (50) = NULL
Fax : varchar (25) = NULL
First_Name : varchar (25) = NULL
Institution : long varchar
Last_Name : varchar (25)
Telephone : varchar (25) = NULL
Title : varchar (25) = NULL
<<primary key>>
@Member_ID : smallint = autoincrement
<<index>>
i_Institution(Institution)
i_Name(Last_Name, First_Name)

Figure 44 *Member* Table

Contains information about DCRIS Members.

Attributes

Address: long varchar

The Member's contact address.

Email: varchar(25)

The Member's email address.

Fax: varchar(25)

The Member' s fax number.

First_Name: varchar(25)

The Member's first name.

Institution: long varchar, not NULL

The Member's institution of study or work.

Last_Name: varchar(25), not NULL

The Member's surname.

Member_ID: smallint, autoincrement, primary key

Key for the Member table. Autoincremented upon insertion.

Telephone: varchar(25)

The Member's contact number.

Title: varchar(25)

The Member's title (i.e. Dr., Professor, etc.)

Indices

i_Institution

An index on the Member *Institution*.

i_Name

An index on the Member name, *First_Name* within *Last_Name*.

Member_Keyword Table

A lookup table that joins the *Member* and *Keyword* tables.

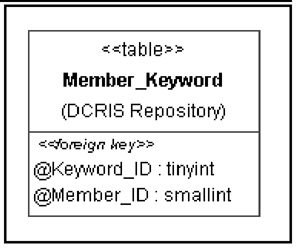


Figure 45 *Member_Keyword* Table

Attributes

Keyword_ID: tinyint, primary key

Foreign key referencing *Keyword* (*Keyword_ID*).

Member_ID: smallint, primary key

Foreign key referencing *Member* (*Member_ID*).

Member_Supp_Key Table

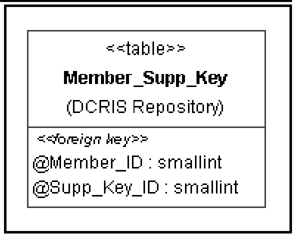


Figure 46 *Member_Supp_Key* Table

A lookup table that joins the *Member* and *Supp_Key* tables.

Attributes

Member_ID: smallint, primary key

Foreign key referencing *Member* (*Member_ID*).

Supp_Key_ID: smallint, primary key

Foreign key referencing *Supp_Key* (*Supp_Key_ID*).

Publication Table

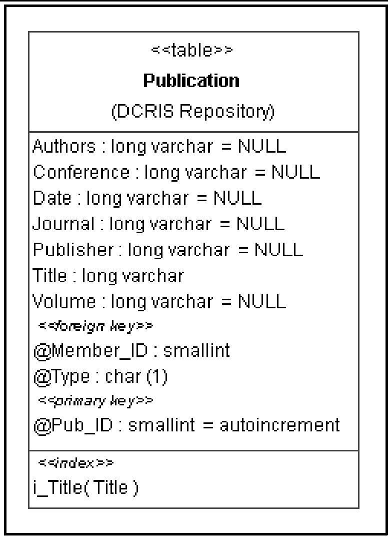

<<table>>
Publication
(DCRIS Repository)
Authors : long varchar = NULL
Conference : long varchar = NULL
Date : long varchar = NULL
Journal : long varchar = NULL
Publisher : long varchar = NULL
Title : long varchar
Volume : long varchar = NULL
<<foreign key>>
@Member_ID : smallint
@Type : char (1)
<<primary key>>
@Pub_ID : smallint = autoincrement
<<index>>
i_Title(Title)

Figure 47 *Publication* Table

Contains representative Publications of Members.

Attributes

Authors: long varchar

Authors (including Member) of Publication.

Conference: long varchar

Conference Publication was presented at. (Not applicable to all publications).

Date: long varchar

Date Publication appeared (or will appear).

Journal: long varchar

Journal Publication appeared in. (Not applicable to all publications).

Member_ID: smallint

Foreign key referencing *Member* (*Member_ID*).

Pub_ID: smallint, autoincrement, primary key

Arbitrary unique identifier for a Publication. Autoincremented upon insertion.

Publisher: long varchar

Publisher of Publication.

Title: long varchar, not NULL

Title of Publication.

Type: char(1)

Foreign key referencing *Pub_Type* (*Type*).

Volume: long varchar

Volume Publication appeared in. (Not applicable to all publications).

Indices

i_Title

An index on the Publication *Title*.

Pub_Type **Table**

Contains Publication Type codes and descriptions.

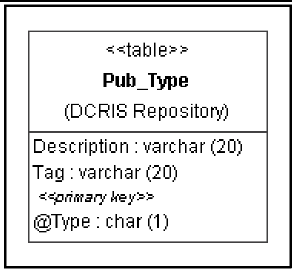


Figure 48 *Pub_Type* Table

Attributes

Description: varchar(20), not NULL, unique

Publication Type description.

Tag: varchar(20), not NULL, unique

XML tag for Publication Type.

Type: char(1)

Primary key identifier for Type. Value is supplied by the designer.

Recommendation Table

Contains representative Publications of Members.

Attributes

Description: long varchar, not NULL, unique

The Recommendation description.

Rec_ID: smallint, autoincrement, primary key

Arbitrary unique identifier for a Recommendation.

Autoincremented upon insertion.

Type: char(1)

Foreign key referencing *Rec_Type* (*Type*).

URL: long varchar, not NULL, unique

The URL for the Recommendation.

Indices

i_Description

An index on the Recommendation *Description*.

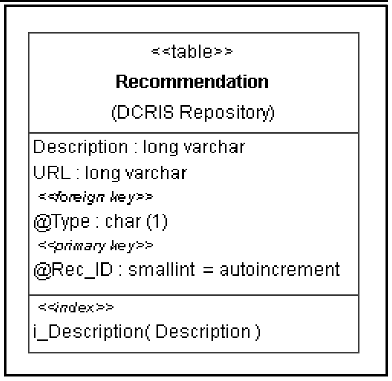


Figure 49 *Recommendation* Table

Rec_Type Table

Contains Recommendation type codes and descriptions.

Attributes

Description: varchar(20), not NULL, unique

Publication Type description.

Tag: varchar(20), not NULL, unique

XML tag for Publication Type.

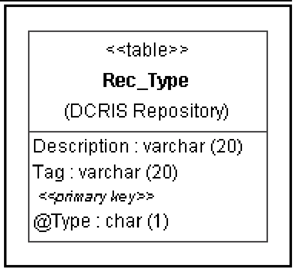
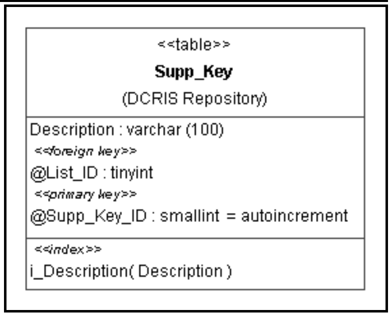


Figure 50 *Rec_Type* Table

Type: char(1)

Primary key identifier for Type. Value is supplied by the designer.

Supp_Key Table



<<table>>
Supp_Key
(DCRIS Repository)
Description : varchar (100)
<<foreign key>>
@List_ID : tinyint
<<primary key>>
@Supp_Key_ID : smallint = autoincrement
<<index>>
i_Description(Description)

Figure 51 *Supp_Key* Table

Contains categories of broad security expertise.

Attributes

Description: varchar(100), not NULL, unique

The Keyword description.

List_ID: tinyint

Foreign key referencing *List* (*List_ID*).

Keyword_ID: tinyint, autoincrement, primary key

Arbitrary unique identifier for a Keyword. Autoincremented upon insertion.

Indices

i_Description

An index on the Supplementary Keyword *Description*.

III.2 DCRIS Table Relations

This section describes the relationships between the tables of the DCRIS database. **Figure 52** shows the same relationships as in **Figure 41**, but the attributes have been left out of the table descriptions in order to simplify the diagram. All table relationships in this database are implemented using foreign keys.

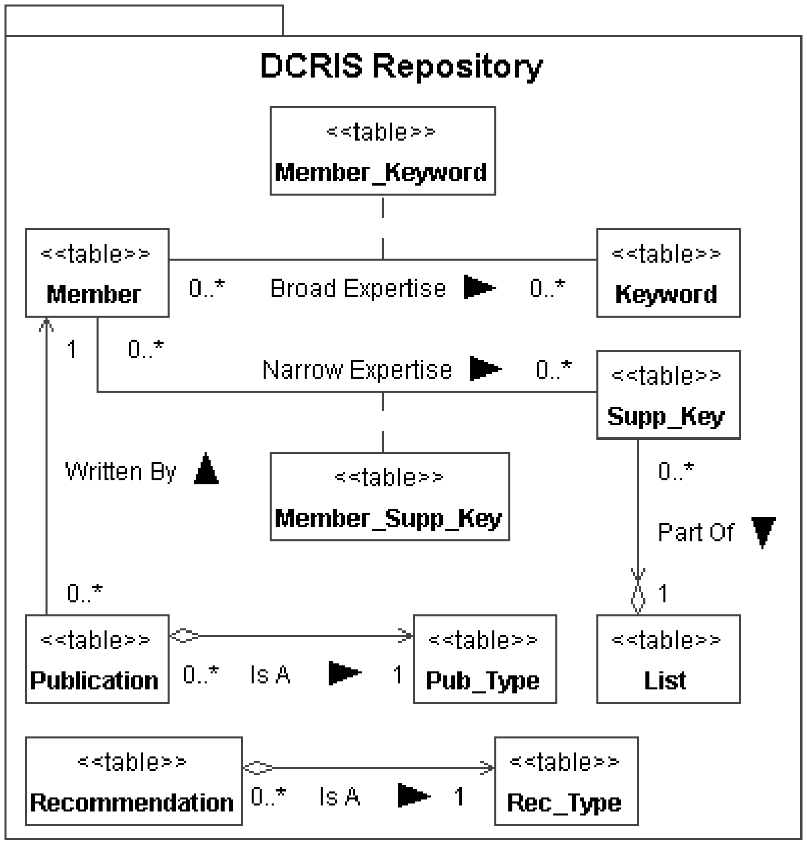


Figure 52 Table Relationships

Member - Keyword Relationship: *Broad Expertise*

Each Member has zero or more broad expertises in areas represented by a Keyword. Each area of broad expertise may be held by zero or more Members. The lookup table *Member_Keyword* implements this many-to-many relationship. *Member_Keyword* contains the foreign key columns *Member_ID* (from *Member*) and *Keyword_ID* (from *Keyword*).

Member - Supp_Key Relationship: *Narrow Expertise*

Each Member has zero or more narrow expertises in areas represented by a Supplementary Keyword. Each area of narrow expertise may be held by zero or more Members. The lookup table *Member_Supp_Key* implements this many-to-many relationship. *Member_Supp_Key* contains the foreign key columns *Member_ID* (from *Member*) and *Supp_Key_ID* (from *Supp_Key*).

Supp_Key - List Relationship: *Part Of*

Each Supplementary Key must be part of a List of areas of narrow expertise. The *Supp_Key* table contains the foreign key field *List_ID* relating it to the *List* table.

Member - Publication Relationship: *Written By*

Publications are written by Members. A Publication may belong to one Member; one Member may write many Publications. The *Publication* table contains the foreign key field *Member_ID* relating it to the *Member* table.

Publication - Pub_Type Relationship: *Is A*

Each Publication must have a publication type (book, journal, paper, etc.). The *Publication* table contains the foreign key field *Type* relating it to the *Pub_Type* table.

Recommendation - Rec_Type Relationship: *Is A*

Each Recommendation must be of a given Type, such as email List (L) or URL (U). The *Recommendation* table contains the foreign key field *Type* relating it to the *Rec_Type* table.

Bibliography

[1] D. D. Cowan and C. J. P. Lucena, “Abstract Data Views: An Interface Specification Concept to Enhance Design for Reuse”, *IEEE Transactions on Software Engineering*, Vol. 21, No. 3, pp 229 - 243.

[2] P. S. C. Alencar, D. D. Cowan, C. J. P. Lucena, and L. C. M. Nova. “The Views Relationship in Object−Oriented Analysis & Design”, Technical Report CS−97−13, Computer Science Department, University of Waterloo, Ontario, Canada, 1997.

[3] P. S. C. Alencar, D. D. Cowan, and C. J. P. Lucena, “A Logical Theory of Interfaces and Objects”, *IEEE Transactions on Software Engineering*, Vol. 28, No. 6, June 2002.

[4] P. S. C. Alencar, D. D. Cowan, and L. C. M. Nova. “A Formal Model for the Views-a Relationship”, *Proceedings of the Third Northern Formal Methods Workshop*, Ilkley, U.K., 1998.

[5] L. C. M. Nova, “A Formalization of an Extended Object Model Using Views,” PhD thesis, Computer Science Department and Computer Systems Group, University of Waterloo, Waterloo, Ontario, Canada, March 2000.

[6] Jim Conallen, *Building Web applications with UML*, Addison Wesley Longman, 2000.

[7] Pierre-Alain Muller, *Instant UML*, Wrox Press Ltd. 1997.

[8] Hans-Erik Eriksson and Magnus Penker, *UML Toolkit*, John Wiley & Sons, 1998.

[9] Mohamed Fayad and Douglas C. Schmidt, “Object-Oriented Application Frameworks”, Special Issue on Object-Oriented Application Frameworks, *Communications of the ACM*, Vol. 40, No. 10, 1997.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*, Addison-Wesley, 1995.

[11] David Garlan and Mary Shaw, *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[12] Thomas J. Mowbray and Raphael C. Malveau, *CORBA Design Patterns*, John Wiley & Sons, 1997.

[13] Michael Stonebreaker with Dorothy Moore, *Object-Relational DBMSs: The Next Great Wave*, Morgan Kaufmann Publishers, 1996.

[14] P. S. C. Alencar, J. Attard, D. D. Cowan, D. M. Germán, Luis Nova, B. Fraser, and J. Roberts, *Hypermedia Documentation - Implementation Strategies*, 2001.

[15] Craig Larman, *Applying UML and Patterns*, Prentice-Hall, 1998.

[16] Perdita Stevens with Rob Pooley, *Using UML: Software Engineering with Objects and Components*, Pearson Education, 2000.

[17] Robert C. Martin and James Newkirk, *A Case Study of OOD and Reuse in C++*, Object Mentor, 1996.

[18] Don Roberts and Ralph Johnson, *Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks*, University of Illinois.

[19] Daniel Austin, Abbie Barbir, and Sharad Garg, ed., *Web Services Architecture Requirements: W3C Working Draft 29 April 2002*, W3C (World Wide Web Consortium) 2002, http://www.w3.org/TR/wsa-reqs.

[20] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler, ed., *Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation 6 October 2000*, W3C (World Wide Web Consortium) 2000, http://www.w3.org/TR/2000/REC-xml-20001006.

[21] James Clark, ed., *XSL Transformations (XSLT) Version 1.0, W3C Recommendation 16 November 1999*, W3C (World Wide Web Consortium) 1999, http://www.w3.org/TR/xslt.

[22] Toacy C. Oliviera, Paulo S. C. Alencar, and Donald D. Cowan, *Towards a Declarative Approach to Framework Instantiation*, 2002.

[23] Marcus E. Markiewiez, Carlos J. P. Lucena, Donald D. Cowan, “Taming Access Security: Extending Capabilities using the Views Relationship”, PUC-RioInf.MCC19/00, May 2000.

[24] Ramez Elmasri and Shamkant B. Navathe, *Fundamentals of Database Systems*, Addison-Wesley, 2000.

[25] Stephen R. Schach, *Classical and Object-Oriented Software Engineering*, 4th ed., McGraw-Hill, 1999.

[26] Martin Fowler with Kendall Scott, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 2nd ed., Addison-Wesley, 2000.

[27] Ezra Ebner, Weiguang Shao, and Wei-Tek Tsai, “The Five-Module Framework for Internet Application Development”, *ACM Computing Surveys*, Vol. 32, No. 1es, March 2000.

[28] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach, “The <bigwig> Project”, *ACM Transactions on Internet Technology*, Vol. 2, No. 2, May 2002.

[29] Daniel Schwabe, Robson Mattos Guimarães, and Gustavo Rossi, “Object-Oriented Design Structures in Web Application Models”, *Annals of Software Engineering* 13, pp 97 - 110, 2002.

[30] D. Schwabe, R. Mattos Guimarães, and G. Rossi, “Cohesive Design of Personalized Web Applications”, *IEEE Internet Computing*, Vol. 6 No. 2, pp 34 - 43, 2002.

[31] J. Gomez, C. Cachero, and O. Pastor, “Conceptual Modeling of Device-Independent Web Applications”, *IEEE Multimedia*, Vol. 8 No. 2, pp 26 - 39, 2001.

[32] *OMG Unified Modeling Language Specification Version 1.3*, Object Management Group, June 1999, http://www.omg.org/cgi-bin/doc?ad/99-06-08.pdf.

[33] Jim Conallen, “Modeling Web Application Architectures with UML”, *Communications of the ACM*, Vol. 42, No. 10, pp 63 - 70, 1999.

[34] Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins, “Modeling Software Architectures in the Unified Modeling Language”, *ACM Transactions on Software Engineering and Methodology*, Vol. 11, No. 1, pp 2 - 57, 2002.

[35] Lidia Fuentes, Jose M. Troya, and Antonio Vallecillo, “Using UML Profiles for Documenting Web-Based Application Frameworks”, *Annals of Software Engineering* 13, pp 249 - 264, 2002.

[36] Grant Larsen and Jim Conallen, “Engineering Web-Based Systems with UML Assets”, *Annals of Software Engineering* 13, pp 203 - 230, 2002.

[37] E. Bertino and E. Ferrari, “XML and Data Integration”, *IEEE Internet Computing*, Vol. 5, No. 6, pp 75 - 76, 2001.

[38] E. Bertino and B. Catania, “Integrating XML and Databases”, *IEEE Internet Computing*, Vol. 5, No. 4, pp 84 - 88, 2001.

[39] R. Klapsing, G. Neumann, and W. Conen, “Semantics in Web engineering: Applying the Resource Description Framework”, *IEEE Multimedia*, Vol. 8, No. 2, pp 62 - 68, 2001.

[40] Iris Reinhartz-Berger, Dov Dori, and Shmuel Katz, “OPM/Web – Object-Process Methodology for Developing Web Applications”, *Annals of Software Engineering* 13, pp 141–161, 2002.

[41] Piero Fraternali, “Tools and Approaches for Developing Data-Intensive Web Applications: A Survey”, *ACM Computing Surveys*, Vol. 31, No. 3, 1999.

[42] *Cascading Style Sheets, level 2, CSS2 Specification*, World Wide Web Consortium, 1998, http://www.w3.org/TR/REC-CSS2/.

[43] *Extensible Stylesheet Language (XSL) Version 1.0*, World Wide Web Consortium, 2001, http://www.w3.org/TR/xsl/.

[44] *PHP: Hypertext Preprocessor*, The PHP Group, 2002, http://www.php.net.

[45] *Scalable Vector Graphics (SVG) 1.0 Specification*, World Wide Web Consortium, 2001, http://www.w3.org/TR/SVG/.

[46] P. Alencar, D. Cowan, T. Nelson, M. F. Fontoura, and C. J. Lucena, "Viewpoints and Frameworks in Component-Based Design", *Building Application Frameworks: Object-Oriented Foundations of Framework Design,* M. Fayad, R Johnson, and D. Schmidt (editors), John-Wiley, 163-165, 1999.