

Visual Definition of Virtual Documents for the World-Wide Web

Mark Minas[†] and Leon Shklar^{*‡}

Abstract. Trying to support the presentation of large amounts of heterogeneous data on the World-Wide Web normally results in relocating and restructuring the original data. Our approach avoids these disadvantages by generating metadata imposing an arbitrary logical structure on existing and new data. This paper proposes a new high-level visual language as a user-friendly means to control the process of generating metadata, i.e., information repositories. The language has been designed to be useful even for unexperienced users. Its applicability is demonstrated by a real example, creating a repository of judicial opinions from publicly available raw data.

1.0 Introduction

Information technology is expected to expand faster than any other technology in history. Large amounts of data are already available on international data-networks, e.g., the Internet and in particular the World-Wide Web (WWW). Two main categories of data have to be distinguished in this context:

1. Data already prepared for the WWW, i.e., formatted using *Hypertext Mark-up Language* (HTML), etc.
2. Unformatted, heterogeneous data, e.g., legacy data, or data processed not primarily for WWW use, e.g., judicial opinions from the U.S. Supreme Court.

Until recently, the only way of homogeneously integrating the second kind of data into the WWW was to reformat the data and place it at a WWW site. Moreover, even HTML documents may be fairly primitive and require restructuring to achieve desired presentation. Such reformatting and restructuring are often impractical because of the amount of human and computing resources they require for the initial conversion and maintenance of information.

As a solution, we have presented the *Harness* system, which provides rapid access to large amounts of new and existing heterogeneous information through WWW browsers without any relocation or restructuring of data [9]. The idea of the system is to impose a desired logical structure on raw data by performing an analysis of the original information and generating metadata to encapsulate portions of this information. *Harness* offers an opportunity to easily integrate existing heterogeneous information into the WWW and to support new sophisticated presentation of data already available on the WWW. The system constitutes a user-adjustable, parameterized, high-order filtering scheme for arbitrary information.

Initially, the generation of metadata entities, which contain knowledge of how to in-

* Computer Science Department, Rutgers University, New Brunswick, NJ 08902, shklar@cs.rutgers.edu

‡ Now with Pencom Systems, Inc., 40 Fulton St., New-York, NY 10038, leon@pencom.com

† Computer Science Department, Univ. of Erlangen, Martensstr. 3, 91058 Erlangen, Germany, minas@informatik.uni-erlangen.de

terpret, filter and compose the original information, has been controlled by a textual modeling language [10]. In this paper, we present a new visual language VRDL (Visual Repository Definition Language) replacing the textual language. There are two main reasons why the introduction of the new language serves to increase the usability of the system:

1. VRDL is easier to comprehend than the original textual language. The language design is inspired by Nassi-Shneiderman diagrams, which are quite popular when teaching programming to novice programmers [8]. Although results from “real-life” experiments are still missing, we expect non-programmers to be able to easily use our visual language.
2. Using an automatic diagram editor generator [7], we have built a graphical editor dedicated to syntactic editing in VRDL. This way, the user gets maximal help when using VRDL.

The rest of the paper is structured as follows: In the next section, we provide a short overview of our object framework. Then, we discuss the main highlights of VRDL. In section 4.0, we present an example of using VRDL to define a structured repository of judicial opinions from the U.S. Supreme Court that are available as plain text files for anonymous ftp from ftp.cwru.edu. Finally, section 5.0 briefly discusses related work and is followed by conclusions.

2.0 Building Information Repositories

An important advantage of our approach is in providing access to a variety of heterogeneous information without making any assumptions about its location and representation. This is achieved by generating metadata entities, which determine processing needed for presenting the associated portions of raw data. We begin with describing the underlying object model and proceed to discussing method sharing between objects.

2.1 The Object Model

The most basic concept in our approach is that of an *encapsulation unit* that is defined as a metadata entity, which encapsulates portions of the original information of interest to end-users. An encapsulation unit may be associated with a file (e.g., the text file representing this paper), a portion of a file (e.g., a section within the text file), a set of files (e.g., the set of images used in this paper), or an operation (e.g., a database query). The text file and a section that occurs in this file may be encapsulated by different units because, in different contexts, each may present a unit of interest.

An *Information Object* in the Harness model (IHO) is either a *simple* object, composed of a single encapsulation unit, or a *collection* object, composed of references to other objects, or a *composite* object, combining a simple object and a set of references to other objects. Each object may contain an arbitrary number of additional attributes (e.g., owner, last update, security information, etc.). A sample composite object may encapsulate this paper’s abstract, combined with a set of references to simple objects that encapsulate text, html, and postscript versions of the full paper.

Collection objects may contain references to multiple independent indices that reference their child objects (Figure 1). An index may be created either from encapsulated contents of child objects or from their attributes (an information source of the index). By abuse of notation, we refer to such collection objects as *indexed collections*, and say that an object belongs to an indexed collection if it is a child of a collection object.

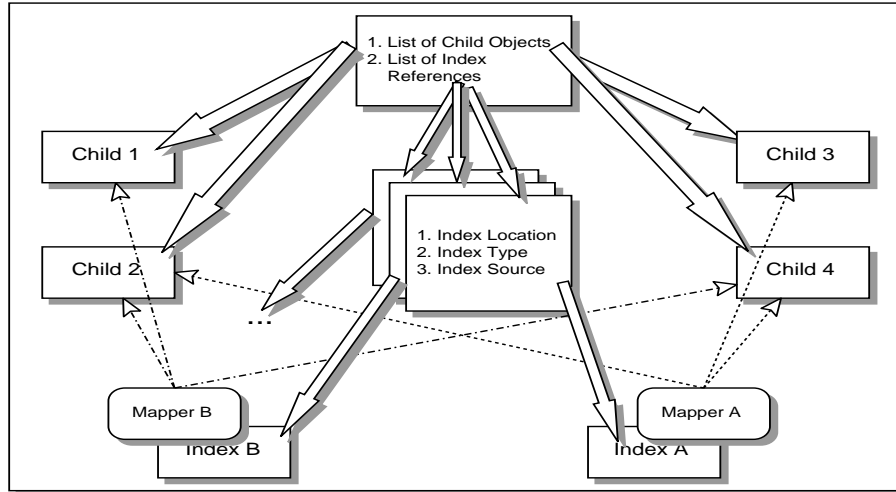


Fig. 1. Indexed Collections.

It is not necessary for every index to reference all child objects. An indexed *sub-collection* is a metadata entity that contains information about a single index, including its source, type, and the location of the index data structures. The type ensures the proper selection of query and mapping methods. These mapping methods are responsible for mapping selected information into Harness objects (Figure 1). Consequently, any indexed collection may make use of external data retrieval methods that are not parts of the system, making it possible to utilize existing heterogeneous index structures.

2.2 Method Sharing

Object encapsulation and presentation methods need not be designed from scratch. Rather, we provide class hierarchies in an object-oriented manner. The hierarchies (Figures 2 and 3) distinguish between *abstract* and *terminal* classes. Abstract classes, which may not be instantiated to IHOs, provide method sharing between groups of terminal classes.

The abstract class hierarchies are *stable* because we do not foresee any need for additional abstract classes to model different kinds of encapsulation and presentation. This notion of stability does not preclude evolutionary changes to the abstract class hierarchies to take advantage of new technology (e.g., Java) or support new functionality (e.g., better flexibility in presenting collection objects). Whenever an appropriate terminal class is defined, it inherits data access and presentation methods from an existing abstract class. The class hierarchies are open in that new terminal classes may be defined to accommodate the vast variety of information.

Structure of the Encapsulation Hierarchy

This section discusses the encapsulation hierarchy (Figure 2) and its role in supporting the metadata extraction process. It is necessary for terminal data encapsulation classes to provide the following methods:

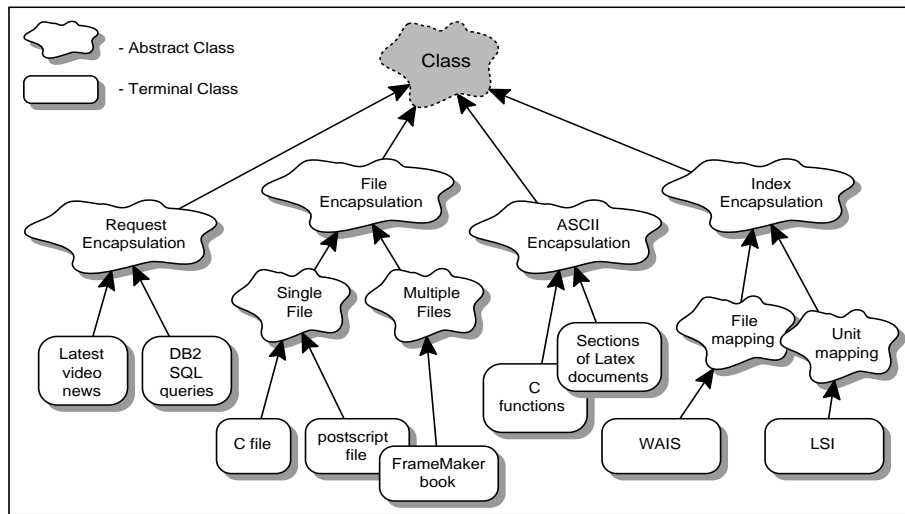


Fig. 2. Encapsulation Class Hierarchy with Sample Terminal Classes.

1. A data encapsulation method, which supports the generation of *access-descriptive* metadata containing information about the location of the encapsulated data, its format and encoding, and its presentation type.
2. A data analysis method, which supports the generation of *content-descriptive* metadata containing additional information about the encapsulated data in the form of attribute-value pairs. For example, an object that encapsulates a tutorial document may contain additional information about the intended audience.

The complexity of data analysis methods may vary greatly. As a minimum, they should assign meaningful names to the Harness objects. The focus of our work is on supporting the generation of access-descriptive metadata. We not attempt to provide any general support for the generation of content-descriptive metadata, but we do provide a general framework for utilizing existing third-party methods designed to perform complex data analysis.

The *ASCII Encapsulation* class is used when Harness encapsulation units are associated with portions of files (e.g., C functions, Prolog predicates, sections of Latex documents, etc.). The *File Encapsulation* class is used to associate the encapsulation units with files (e.g., postscript files, Frame documents, image bitmaps, etc.) and groups of files (e.g., Frame books, judicial opinions related to a single case, etc.). Finally, the *Request Encapsulation* class is used to encapsulate requests to external applications (e.g., SQL queries and requests for regularly updated video clips). A declarative approach to generating encapsulation and data analysis methods for new terminal classes is a subject of our current work.

The encapsulation classes for content-based indices are designed to support methods, which synchronize the generation of index structures and structures that support presentation-time one-to-one mapping between indexing units and some set of Harness objects. The indexing units are defined as portions of information that may be searched for, given particular index structures. The two different abstract classes in the encapsulation hierarchy are designed to support different levels of openness of third-party in-

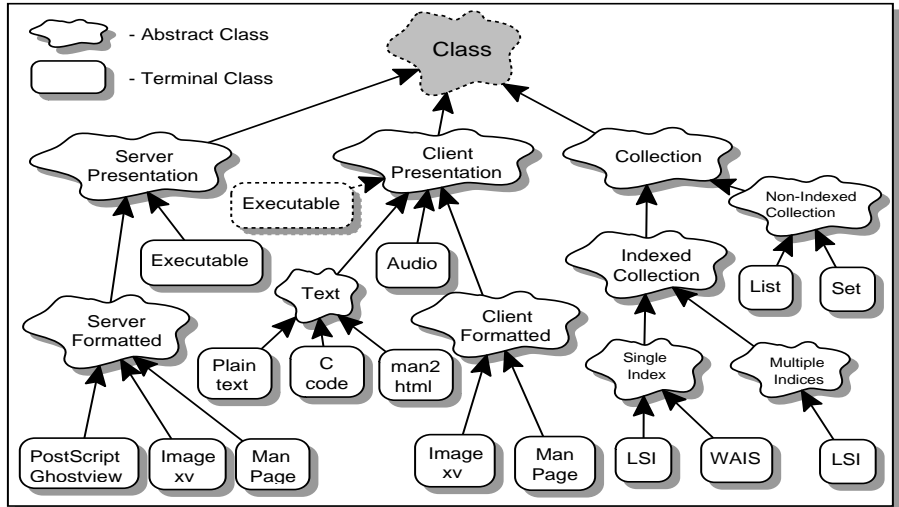


Fig. 3. Presentation Class Hierarchy with Sample Terminal Classes.

dexing technologies. The *File Mapping* class is used when the indexing technology's query method returns file names, while the *Unit Mapping* class is used when the method returns unit identifiers.

Structure of the Presentation Hierarchy

This section discusses the presentation class hierarchy (Figure 3) that was constructed to serve a triple purpose:

1. Support presenting the encapsulated information.
2. Support pre-processing the encapsulated data for building independent indices.
3. Support executing queries against independent indices and presenting the results of these queries.

Each instance of the *Collection* class stores a set of parent-child relationships between Harness objects. In addition, instances of the *Indexed Collection* class are associated with a physical index (or indices) that is used at run-time to select members of the collection. Instances of subclasses of the *Indexed Collection* class are presented through query interfaces, while instances of classes that inherit from *Non-Indexed Collection* are presented as full lists of members.

The distinction between the *Server Presentation* class and the *Client Presentation* class is based on the differences in the execution site of the presentation tools while the rationale for their subclasses is in the choice of these tools. The classes don't differ in their treatment of content pre-processing when building independent indices.

The *Server Presentation* abstract class helps to group objects, for which the encapsulated information is accessed at run-time by running a process at the Harness server. The subclasses of this class are *Server Formatted Data* and *Server Executable*. The subclasses of *Server Formatted Data* serve to access raw data by executing external viewers at the server but displaying the windows at the client. Instances of the class *Executable* serve to encapsulate application programs that get executed at the server.

The subclasses of the *Client Presentation* class include *Client Formatted Data*, *Text*, and *Audio*. For instances of terminal subclasses of *Client Presentation*, data is first transferred to the client and then accessed by running external viewers at that client. Initially, for security reasons, there was no support for the *Client Executable* class, but the emergence of *Java* [42] and other mobile code technologies has helped to make it available.

Most of the data types may be defined by either a subclass of *Server Presentation* or a subclass of *Client Presentation*. The exceptions are *Audio* and *Text* that are always defined by instantiating the *Client Presentation* class. The special treatment of audio files is determined by the need to play the recording at the client machine for it to be heard. The special treatment of text is for convenience in presenting plain text, as well as documents that use a mark-up language known to Web browsers. In either case, the text is presented by the Web browser and not by an external viewer.

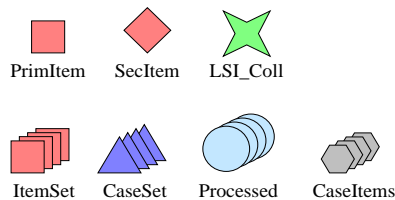
3.0 The Visual Repository Definition Language

In this section, we discuss the main highlights of the *Visual Repository Definition Language* (VRDL). We limit our description to specifying the construction of metadata objects and thus building information repositories. We do not describe adding support for new kinds of data and new indexing technologies by introducing new terminal classes to the class hierarchy.

A VRDL program, when interpreted, generates objects that make up a structured information repository. VRDL contains language constructs for building simple objects by encapsulating raw data, collection objects by putting together references to different objects and possibly indexing their encapsulated contents, and composite objects by combining simple objects and a set of references. Basic data types are IHOs and sets of IHOs. Other VRDL types are omitted in this discussion.

VRDL is a high-level language, a program's intermediate and final results are stored in variables which have to be declared first. Each VRDL program consists of a declaration block and a sequence of statements (Figure 4).

Here, we show a declaration block. Each variable is represented by an icon together with its name. Shape and color provide easy distinctions between different variables. Set objects are indicated by the stacks of icons:



The statement sequence is represented by a rectangle, which is subdivided into smaller rectangles for individual statements (Figure 4). We will now discuss a subset of VRDL statements, selected to support the example in the next section.

The assignment statement assigns a value to a variable. Values are defined by expressions, which, in the simplest case, may be either constants or other variables. Here, the empty set is assigned to variable 'Processed':



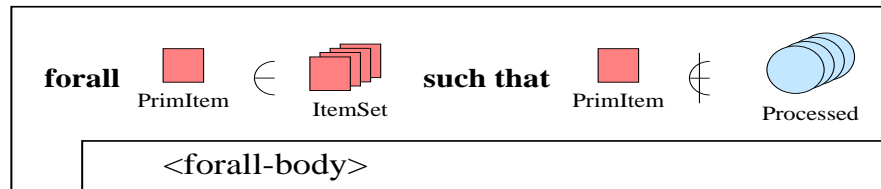
A variant of the assignment statement is the add-to-set statement, indicated by the double arrow. The variable must be a set variable. The value of the right-hand-side expression is then added to the set as an additional element.



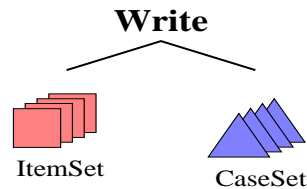
If the right-hand-side expression is set-valued, the set is added to the set variable:



The forall-statement provides selective access to set members. In this example, the forall-body (here represented by `<forall-body>`) is executed for every element contained in 'ItemSet', which is not contained in the set accumulated in the variable 'Processed'. For each iteration, the element of 'ItemSet' is assigned to the variable 'PrimItem'. Apparently, the forall-statement exceeds similar statements of other languages by providing additional flexibility in defining the forall-head conditions:



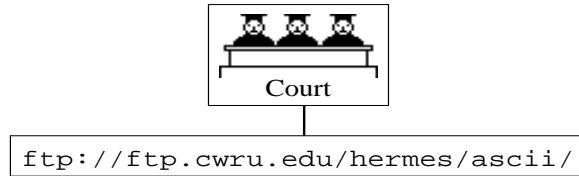
The last statement showed here is the write statement used for storing generated metadata entities, e.g., the contents of the variables 'ItemSet' and 'CaseSet' (of course, input statements also exist in VRDL). The stored metadata entities eventually build up the information repository described by the VRDL program:



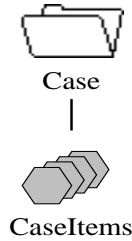
So far, we have described how to access existing objects and sets of objects. In the rest of this section, we present more complicated expressions that help to create new objects.

The encapsulation expression creates simple IHOs by analyzing raw data. In this example, objects of type 'Court' are built from the contents of the given ftp-directory (the

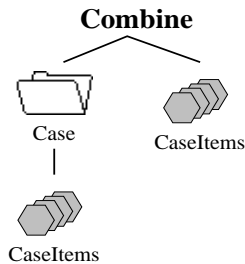
new type 'Court' together with its icon is defined outside of VRDL). The type determines how the original data is analyzed and presented. Here, 'Court' objects encapsulate multiple files related to the same case:



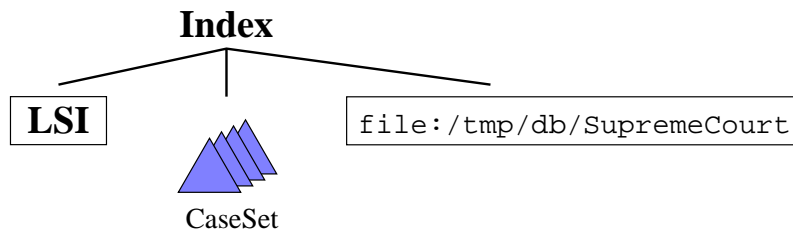
A variant of the encapsulation expression allows to build metadata objects from other metadata objects, providing a different view of the same data:



The combine-expression is used for building composite objects. A simple object is combined with a set of references to other objects. Here, the object created by the encapsulation expression is combined with objects contained in the set variable 'CaseItems':



Finally, the index expression creates collection objects that contain a set of references to other IHOs and to a searchable index. Here, a collection object is built from the contents of the set variable 'CaseSet', building a new index at the given position. Latent Semantic Indexing (LSI) [3] is used as an indexing scheme:



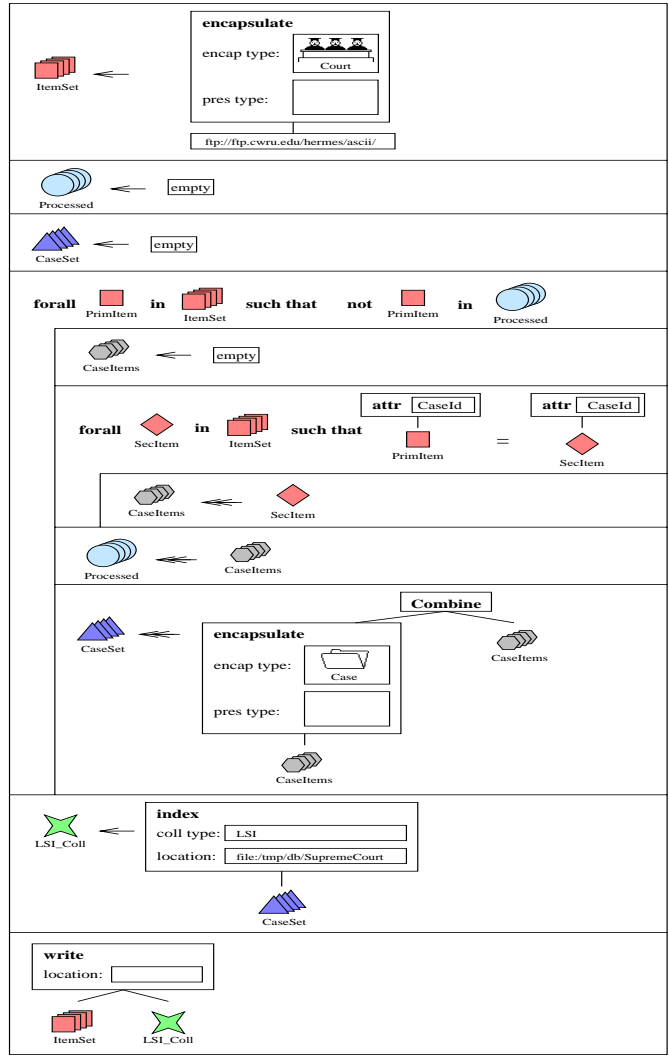


Fig. 4. VRDL program for example in section 4.0.

4.0 Example

In this section, we discuss how to best search and present the judicial opinions from the U.S. Supreme Court that are available at ftp.cwru.edu. Here, information related to a single case may be distributed between multiple files. The example impressively demonstrates how information not prepared for the WWW can effectively be presented on the Web.

Given the location of the original information, the desired run-time presentation of individual cases, and the desired indexing technology, the following steps would result

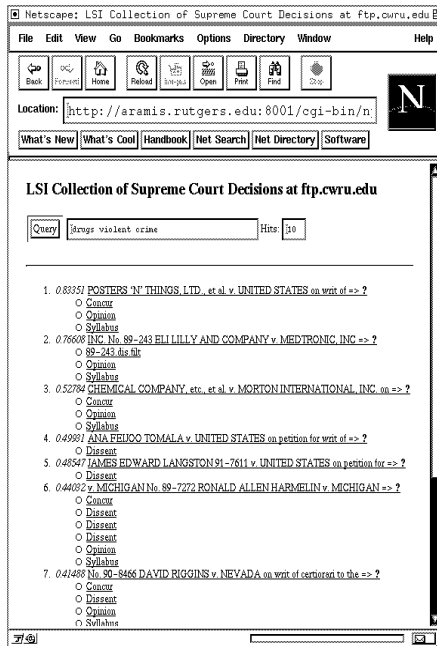


Fig. 5. Query interface for judicial opinions from the U.S. Supreme Court.

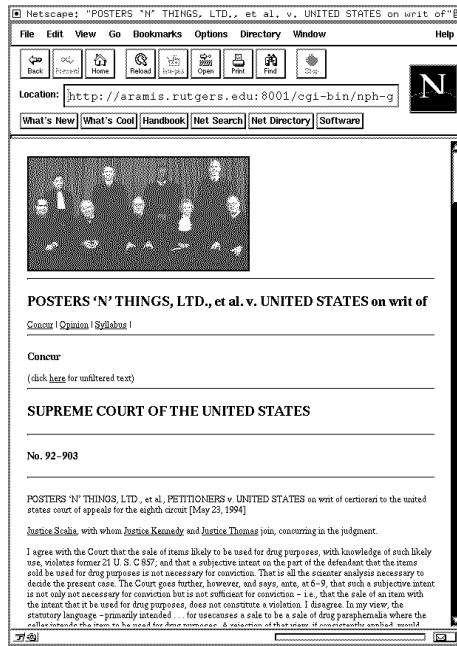


Fig. 6. Presentation of legal cases from the U.S. Supreme Court.

in building a repository of the Supreme Court cases:

1. Create simple objects that encapsulate individual judicial opinions (one per file). The encapsulation method should determine the case numbers for the opinions and store them as attributes of the encapsulating objects.
2. For each object created in step one, find other objects related to the same case, encapsulate them together with the presentation type 'Case', and exclude them from any further consideration. The presentation method for this type should be responsible for generating internal hyperlinks to individual opinions and external hyperlinks to related information (the Supreme Court photo, bios of the judges, etc.).
3. Create an indexed collection of the objects created in step 2 using the Latent Semantic Indexing (LSI) technology.

These steps are implemented by the VRDL program in Figure 4. It assumes that the LSI indexing technology is supported and that the encapsulation and presentation methods for types 'Court' and 'Case' are available in the type library.

The first statement of the program serves to encapsulate individual opinions located at "ftp://ftp.cwru.edu/hermes/ascii/" and assigns the generated set of simple objects to 'ItemSet'. The encapsulation type is always required because it determines how to analyze the original data. When the presentation type is not explicitly specified, it is assumed to be the same as the encapsulation type. The next two statements initialize 'Processed' and 'CaseSet' variables. 'Processed' is used to accumulate objects from 'ItemSet' that should be excluded from further consideration, while 'CaseSet' is used for grouping together objects, which encapsulate opinions that belong to the same cases.

Next, the forall-statement serves for iterating over the objects in 'ItemSet' and uses 'Processed' in the such-that condition to avoid assembling the same case for every member opinion.

Objects that are related to the same case are determined using the 'CaseID' attribute, which is set by the encapsulation method for the type 'Court'. The discussion of this feature was omitted in the last section. All objects related to the same case are grouped together using the encapsulation type 'Case'. We then use the combine-operation to create composite objects that both encapsulate all case-related information and contain references to simple objects encapsulating individual opinions. Finally, when all opinion objects are grouped together, an indexed collection is created for objects in 'CaseSet' and the collection object is assigned to the 'LSIColl' variable. 'LSIColl' and 'ItemSet' are stored; together, they constitute an information repository properly formatted for the Harness server.

Figure 5 shows a Web page that gets generated after searching for decisions using keywords `drugs`, `violent`, and `crime`, i.e., using the index referenced by the object stored in 'LSIColl'. The result of the query is a list of the members of the set in 'CaseSet'. Since each member is a composite object, we see not only a hyperlink for its content but also hyperlinks for the individual opinions. The dynamic Web page for the case object which is referenced by the first hyperlink in Figure 5 is shown in Figure 6.

5.0 Related Work

The access and retrieval of heterogeneous information has historically concentrated on different application areas, including software reuse [1], digital libraries [5], geospatial data [4], etc. Software reusability now extends beyond code to include other software assets such as specifications, designs, test cases, plans, data, and documentation. The construction of digital libraries and information repositories of geospatial data require assembling a variety of media types, both structured and unstructured, and consequently ensuring ease of access and manipulation. The basic trade-off for these applications lies in balancing the cost of constructing a storage system versus the cost of locating and browsing relevant resources.

Furthermore, VRDL is related to visual programming languages and environments as well as visual programming [2]. The visual approach has been quite successful for restricted domains and programming tasks as well as for teaching inexperienced users. Examples include attempts to generalize spreadsheets to real (visual) programming languages, and *Prograph*, a visual programming language and environment currently used for moderately sized software projects [6]. This success with casual and inexperienced programmers was the motivation for our design of VRDL. We used a Nassi-Shneiderman diagram representation [8], which is not (as far as we know) incorporated in prominent software products, but which successfully serves as a visualization aid in teaching novice programmers.

6.0 Conclusions

We have briefly introduced VRDL, a visual language describing how to build Web information repositories from large amounts of heterogeneous data. VRDL is the result of our efforts to design a simple, yet powerful, language that supports modeling heterogeneous information based on the underlying notions of sets and types, and to make using the language simple enough for unsophisticated programmers. Whereas the concept of

a simple, declarative, textual language to support modeling is not new (e.g., [8], for a detailed discussion of related languages see [10]), the visual language is a unique feature increasing user-friendliness.

7.0 Acknowledgements

Work on the Web presentation of legal information is performed jointly with L. Thorne McCarty from Rutgers University.

References

- 1 V.R. Basili. *Support for comprehensive reuse*. Software Engineering Journal, pages 303-316, 1991.
- 2 M.M. Burnett and A.L. Ambler. *Interactive visual data abstraction in a declarative visual programming language*. Journal of Visual Languages and Computing, 5:29-60, 1994.
- 3 S. Deerwester, S.T. Dumais, G.W. Furnas, T.K. Landauer, and R. Hashman. *Indexing by latent semantic indexing*. Journal of the American Society for Information Science, 41(6), 1990.
- 4 Federal Geographic Data Committee. *Content standards for digital geo-spatial metadata*. Federal Geographic Data Committee, June 1994.
- 5 C. Fisher, J. Frew, M. Larsgaard, T. Smith, and Q. Zheng. *Alexandria digital library: Rapid prototype and metadata schema*. In Advances in Digital Libraries, pages 173-194. Springer-Verlag, New York, 1995.
- 6 E.J. Golin. *Tool review: Prograph 2.0 from TGS systems*. Journal of Visual Languages and Computing, 2(2):189-194, June 1991.
- 7 M. Minas and G. Viehstaedt. *DiaGen: A generator for diagram editors providing direct manipulation and execution of diagrams*. In Proc. 11th IEEE Int. Symp. on Visual Languages (VL '95), Darmstadt, Germany, pages 203-210. Sept. 1995.
- 8 I. Nassi and B. Shneiderman. *Flowchart techniques for structured programming*. ACM SIGPLAN Notices, 8(8):12-26, Aug. 1973.
- 9 L. Shklar, A. Sheth, V. Kashyap, and K. Shah. *InfoHarness: Use of Automatically Generated Metadata for Search and Retrieval of Heterogeneous Information*, Lecture Notes in Computer Science #932, Springer-Verlag, 1995, pp. 217-230.
- 10 L. Shklar, K. Shah, and C. Basu. *Putting legacy data on the Web: A repository definition language*. Computer Networks and ISDN Systems, 27(6):939-952, April 1995. Special Issue on the Third International WWW Conference'95.
- 11 J.H. Taylor. *Toward a modeling language standard for hybrid dynamical systems*. In Proc. 32nd Conf. on Decision and Control, San Antonio, Texas, USA, pages 2317-2322 Dec. 1993.