FEATURE IDENTIFICATION FRAMEWORK AND APPLICATIONS (FIFA)

A Thesis

by

MICHAEL NEAL AUDENAERT

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2005

Major Subject: Computer Science

FEATURE IDENTIFICATION FRAMEWORK AND APPLICATIONS (FIFA)

A Thesis

by

MICHAEL NEAL AUDENAERT

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Richard Furuta
Committee Members,	John J. Leggett
	Eduardo Urbina
	Brian Imhoff
Head of Department,	Valerie E. Taylor

December 2005

Major Subject: Computer Science

ABSTRACT

Feature Identification Framework and Applications (FIFA). (December 2005)
Michael Neal Audenaert, B.S., Texas A&M University
Chair of Advisory Committee: Dr. Richard Furuta

Large digital libraries typically contain large collections of heterogeneous resources intended to be delivered to a variety of user communities. One key challenge for these libraries is providing tight integration between resources both within a single collection and across the several collections of the library with out requiring hand coding. One key tool in doing this is elucidating the internal structure of the digital resources and using that structure to form connections between the resources. The heterogeneous nature of the collections and the diversity of the needs in the user communities complicates this task. Accordingly, in this thesis, I describe an approach to implementing a feature identification system to support digital collections that provides a general framework for applications while allowing decisions about the details of document representation and features identification to be deferred to domain specific implementations of that framework. These deferred decisions include details of the semantics and syntax of markup, the types of metadata to be attached to documents, the types of features to be identified, the feature identification algorithms to be applied, and which features should be indexed. This approach results in strong support for the general aspects of developing a feature identification system allowing future work to focus on the details of applying that system to the specific needs of individual collections and user communities.

ACKNOWLEDGMENTS

I would like to thank my committee chair, Dr. Richard Furuta, and my committee members, Dr. Leggett, Dr. Urbina, and Dr. Imhoff, for their guidance and support throughout the course of the research. I would also like to thank Dr. Shipman for graciously agreeing to sit in as a substitute for Dr. Leggett during my defense. Also, many thanks to Unmil Karadkar for his many helpful comments about both my thesis work and life in graduate school.

Research for this thesis as part of the Cervantes Project was funded by the Cervantes Chair (University of Castilla-La Mancha), and the Grupo Santander.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

INTRODUCTION

As the *Cervantes Project* [51] has matured we have begun to shift our focus from providing tools to present and analyze the writings of Cervantes [26][29][36] to collecting and integrating resources that serve the research needs of scholars from a variety of academic domains [3]. This includes the iconography project [52], the music collection [38], and the Sliwa collection of historical documents [45]. This has raised two key questions: First, how can we provide tight interlinkages between resources developed by scholars in diverse fields without requiring extensive hand coding–an unaffordably labor intensive process? Second, how can we provide tools that both allow and promote sophisticated reading strategies that will help scholars best utilize these unique resources?

One approach to addressing these questions is elucidating the internal structure of documents in a collection. Once this internal structure is made explicit, it can serve as the basis for establishing connections between documents and for providing information visualizations, advanced search tools, and navigational links that more adequately meet the needs of the communities using the collection. A first step toward this goal is identifying key features within the documents. In the context of a mature digital library, feature identification systems face the challenges of dealing with heterogeneous document types and supporting the needs of a variety of user communities from both the academic and public sectors [11].

In this thesis, I describe an approach to developing a feature identification system for the Sliwa document collection–a collection of official records pertaining to Miguel de Cervantes Saavedra (1547-1616) and his family originally assembled by Prof. Kris Sliwa [45]. This collection contains descriptions, summaries, or transcriptions in Spanish of nearly 1700 documents originally written from 1463 to 1681. Ultimately it is intended that this work will be extended both to other collections within the *Cervantes Project*

This thesis follows the style of the International Journal of Digital Libraries

(*CP*) and to collections outside the *CP* such as the records of early Spanish expeditions into Texas [24] and the narrative components of the *Picasso Project* [33]. Accordingly, the system presented here provides a general framework that can be tailored to meet the specific needs of individual collections and user communities.

To achieve this generality, the feature identification system developed as part of my thesis is broken into three layers. As diagramed in Figure 1, the core of the system is a "feature identification framework" (FIF). This framework provides a set of tools for working with documents, identifying features within documents and building indices for searching document collections based on the identified features. This framework provides the major structural elements of the system, while deferring decisions about how documents are to be represented and stored, what types of features and feature identification algorithms are to be used, and how the collection is to be indexed to “implementations” of the framework. Domain specific applications then use this framework, along with the appropriate set of customized modules to implement visualizations, navigational linking strategies, and searching and filtering tools.

This approach to developing a feature identification system opens a number of potential research directions that offer trade-offs between pursuing depth in a particular area of the

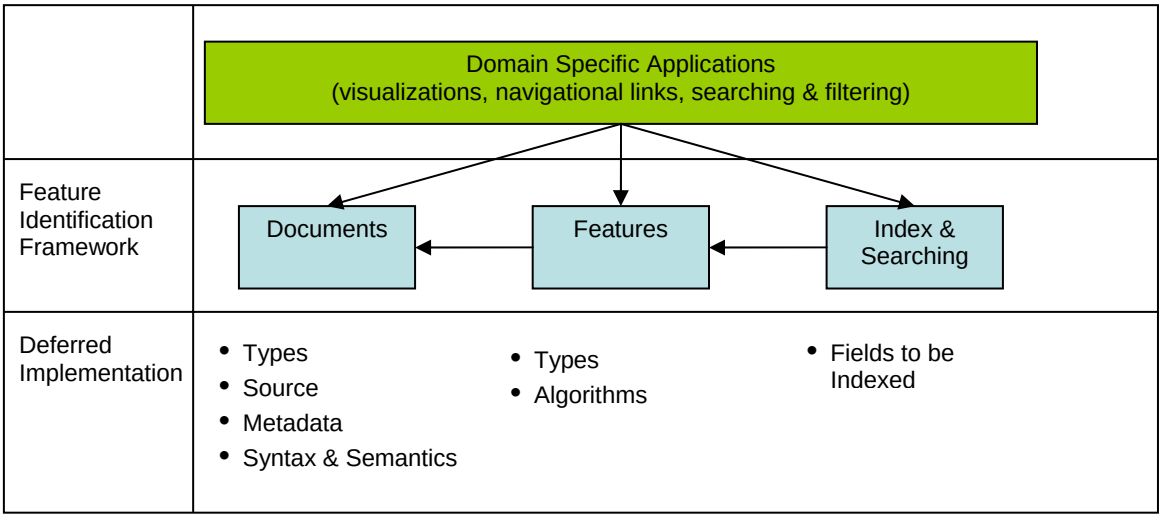

Figure 1: High-level components of a feature identification system

system (e.g., better natural language processing based feature identification and disambiguation algorithms or a novel visualization to meet a challenging research need) or breath across the system as a whole. I have taken a horizontal approach. The resulting system provides an implementation of the core feature identification framework along with a basic implementation of the document model and feature identification components. This framework is applied to develop a web-based interface for the Sliwa collection.

This work is presented with two main objectives. First, it describes a system which fulfils a tangible need within the Cervantes Project–namely presenting the documents of the Sliwa collection in a way that facilitates scholarly use. Second, by exploring the needs of a general feature identification system, this work will serve as an example to other digital libraries of how to integrate flexible feature identification strategies across their collections.

BACKGROUND

When it comes to integrating documents in a collection and providing information to support user interfaces, encoding document level metadata is the dominant strategy for supplying the needed information. Numerous, well-established metadata standards exist for digital librarians to choose from (Dublin Core [19], METS [31], MPEG-21 DIDL [6][32] and MARC [30] to name a few of the more prominent standards). In many ways, metadata is a digital analog to the notions of a card catalog in a physical environment. One limitation of using metadata as the sole method for enhancing a collection is its relatively coarse grained nature. In a physical library, where librarians are not free to modify the contents of their collections to facilitate information finding tasks, the limited granularity of the catalog cannot be helped. In a digital library, however, the contents of the documents can be modified–reshaped to meet the needs of the tools that will use them.

In a digital environment, explicitly identifying important features within the documents themselves provides an important complimentary approach for enhancing cultural archives. While this approach bears many similarities to traditional editorial practices, it also raises unique challenges that humanities scholars have recently begun to address [44][25][48]. Unlike traditional edited editions in physical libraries, the enhancements made by digital editors can be made available for use by the tools and services provided by the library. In addition to structural markup of pages such as line numbers and speaker changes, identifying people, places, monetary units and morphological structures have allowed humanists more complete access to the wealth of information contained in cultural archives [16][17]. Once identified, this information can be used to support georeferencing map based interfaces and timeline visualizations [13]. Advanced linking and searching tools provide readers with powerful tools to find biographical, historical and linguistic information to enhance their understanding of texts.

Assigning metadata is a notoriously resource intensive and time consuming process. Consequently, a number of approaches have been explored to mitigate this cost including tools to better support collection editors [4] and automatically assigning metadata [39]. Editorial tasks are far more resource intensive and a number of algorithms have been developed to assist in this process, many used in the context of online news services.

Many of these algorithms solve problems that fall under the category of Named Entity Recognition (NER). The named entity task was formally introduced as part of the Message Understanding Conference/Competition, MUC6 [47] in which participants were to identify seven types of entities (people, organizations, location, date, time, money and percentages) in texts. The state of the art systems that participated in this and the following MUC7 [8] conference were able to achieve extremely good results, often with recall and precision rates higher than 90%.

Most NER systems make extensive use of large gazetteers such as the Getty Thesaurus of Place Names [22] used extensively in [12]. These can be extremely expensive to build and maintain and often represent a limiting factor in building NER systems. Krupke and Hausman [27][28] have found that the quality of the names in a gazetteer is far more important than the overall size, suggesting that smaller name lists more closely tailored to the domain in which they are to be used might provide more cost effective alternatives to general purpose gazetteers. To provide higher quality name lists, [46] uses manually annotated training data to build corpus derived name lists and [35] presents a system that uses rule-based grammars and statistical models in conjunction with relatively small gazetteers to achieve good results on the texts in the MUC-7 competition. Recent work has tended to emphasize statistical approaches, notably Hidden Markov Models (HMM) [5] and semi-Markov models [9].

Whereas most NER systems attempt to identify patterns that hold for all documents in a collection, [7] report on a system that attempts identify document specific heuristics that better identify names in tables, lists and other visually structured data. While their

approach is limited to environments where there is considerable internal document structure, they were able to achieve significantly higher recall than traditional methods without a large drop in precision. Their results are particularly important in the context of cultural archives given the fact that many significant documents contain structured presentation similar to the structure they have found in web pages [12].

While state of the art NER systems have met with significant success, they are limited by their focus on broadly defined categories (e.g. names, places, organizations and products). [37] observes that in an information seeking context, the ability to ask much more refined questions from multiple organizational perspectives is needed.

From this brief sketch of related work, a number of general principles for the design of a general purpose feature identification system can be inferred. The system will need to be able to handle multiple document formats and the metadata associated with those documents. It will also need to provide support for many different types of feature identification algorithms and facilitate editorial customizations of the name lists used to support these algorithms. User communities will need tools tailored to closely support their idiosyncratic information finding needs. These tools may be able to share feature identification algorithms and indexing strategies, but will likely not overlap completely. Hence, the system should be implanted to permit feature identification resources to be shared, but applied flexibly to meet specific needs. Together, this calls for a strong separation between the user-centric applications, document representation and feature identification components of the system.

FRAMEWORK IMPLEMENTATION

The Feature Identification Framework (FIF) forms the core of my thesis work. This framework provides the basic structures for supporting general feature identification that can be customized and applied to meet the specific goals of a project or sub-project. The FIF allows tasks common to all feature identification systems to be implemented in a general fashion. Tasks that are specific to particular document collections or research needs can then be implemented as extensions of this general framework. This allows subject area and task specific development projects to focus more directly on meeting the needs of end users while relying on the framework to provide the overall structure and generic functionality needed in a feature identification system. This section presents detailed descriptions of the components of the FIF and discusses the design decisions that have motivated them.

Feature Identification Framework

Figure 2 diagrams the major structural components of the Feature Identification Framework. Items in blue represent the document model, items in green represent the feature identification model, items in yellow represent indexing and searching components, and items in red represent external tools used by the framework. Components whose names are italicized are abstract components whose implementations are intended to be deferred to domain specific applications if the default implementation is insufficient.

The Document Manager manages document persistence and named document collections. The Document Collection component provides management for the directory space in which collection information is stored and serves as an abstraction layer on top of the Lucene search engine [1][23]. It supports indexing and searching two general classes of information: the full text of documents and the features identified in the documents. The Document Collection uses an Index Filter (wrapping a document) to provide customized support for determining how a document should be processed for

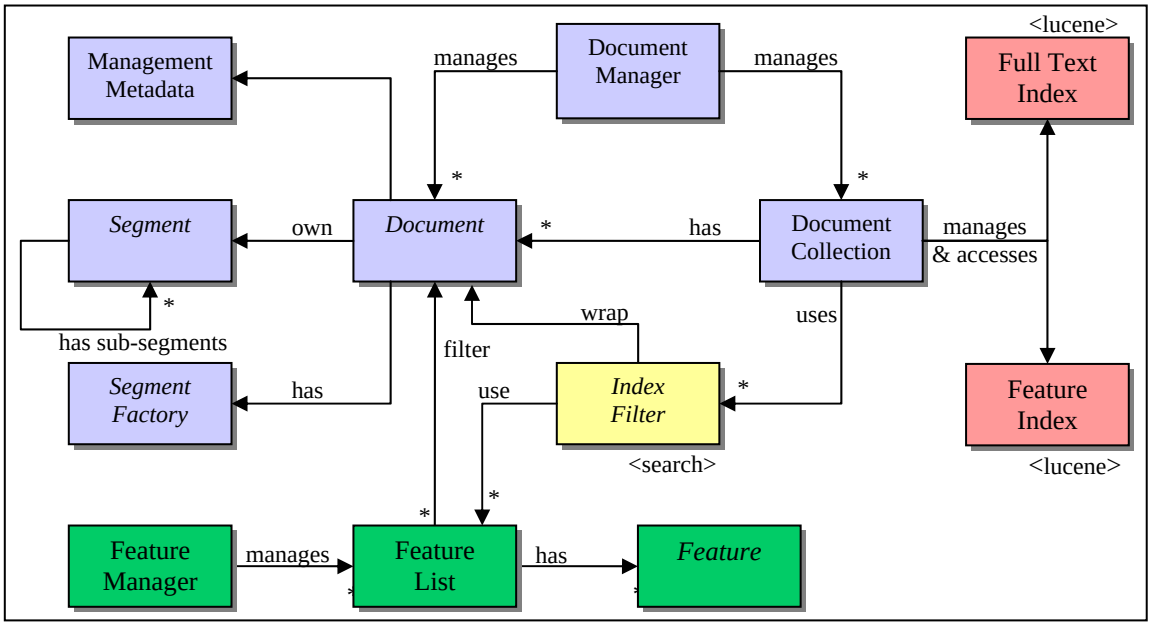

Figure 2: Structural components of the feature identification framework

indexing. Each document is composed of management metadata (containing both information specific to a customized document implementation and general metadata used by the Document Manager) and a content tree of Segment nodes. This tree structure is suitable for representing XML documents and corresponds to the theoretical view of a document as an Ordered Hierarchy of Content Objects. Every document has a Segment Factory that can used to enforce constraints on the structure of the content tree. A Feature Manager is responsible for the creation and persistence of named Feature Lists. Each Feature List contains a number of Features and is responsible for the persistence of those features. The Feature List works in cooperation with the Document component and the individual Features to support feature identification and markup within a document. The Index Filter may use multiple feature lists to support the identification of features to be indexed. The details of this will depend on the details of custom index filter implementations.

Document Model

At the heart of any system for enhancing documents in a digital archive will be the set of tools it provides for managing and representing those documents. The implementation of the FIF document model has two primary goals. The first is to provide an implementation of core functionality needed to represent documents, ensure their persistence, provide an interface to searching mechanisms and facilitate the automatic identification of features. The second is to "play well with others." That is, it must be flexible enough so that documents managed by other tools and digital libraries can be ingested into the FIF with minimal effort and then used without adversely affecting the other environments in which those documents are being used. The following guidelines have helped shape the design of the document model:

1. **Document types:** XML is the dominant technology for storing digital resources–and the technology that will be used to implement my thesis work–but it is by no means the only one. The document model must be capable of supporting multiple types of documents (e.g., XML, PDF, RTF, HTML, etc) without modifying the APIs with which the rest of the system will interact.

2. **Markup and encoding formats**: Mature digital collections will contain documents encoded using a variety of standards–both within a single collection as well as between document collections in a single library. The document model must allow implementations that provide support for arbitrary encoding formats. The framework should provide support for explicitly specifying these formats and testing documents to ensure their validity, especially to ensure that the markup of identified features does not violate the document encoding standards.

3. **Metadata standards:** The document model must allow implementations to attach metadata conforming to arbitrary metadata standards to documents.

4. **Document persistence:** The document model needs to provide mechanisms for uniquely identifying and retrieving documents. In some cases, these documents

will be managed and stored by systems external to the FIF. In other cases, the FIF will need to provide a persistence mechanism for storing these documents. The document model should be robust enough to support both of these cases. In addition to storing the documents themselves, the document model will need to account for the storage of indices that will be built to support searching.

5. **Services:** In addition to the services provided by an used by the FIF, the document model should allow extensions that implement services and functionality specific to the types of documents being used and the needs of the research contexts in which they are being used.

To meet these requirements, the document model has been designed as diagrammed in Figure 3. The document manager supports the unique identification and retrieval of documents and the management of named document collections. The document collection module provides services for grouping individual documents and working

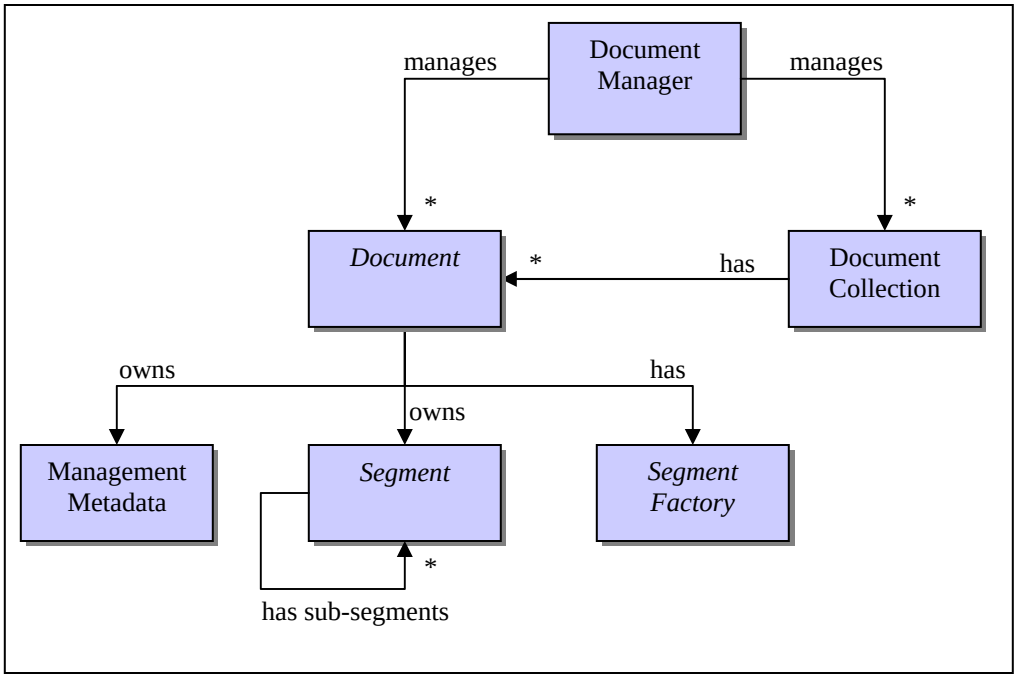

Figure 3: The document model

with those collections. This includes interfacing with the search engine and providing abstracted access to the file system for use in maintaining custom data specific to a particular document collection. The document module provides the basic document level services needed by the FIF and can be extended to provide customized support as needed by applications of the framework. The document module also allows arbitrary customizations to the metadata associated with each document. The document content is provided by a hierarchical tree of segments whose structure can be controlled by a segment factory. The remainder of this section describes each of these features of the document model and how they may be extended in detail.

The document module wraps the segment-based content of the document to provide an extensible interface. This interface is then used by the document manager to support the unique identification and persistence of documents and by the feature identification sub-system as the entry point for filtering documents. From the framework's point of view, a document is an instance of the abstract class, `edu.tamu.csdl.documents.Document`. These instances can be decomposed into three main parts: document management information, implementation information, and the document contents. The document contents are implemented by the segment model that will be discussed in detail below. Together, the document management information and the implementation information comprise the management metadata shown in Figure 3. The document manager uses the document management information to support serialization and restoration of documents. Most notably, this information specifies a unique document id and describes which concrete sub-class of the `Document` class should be used to restore a document. The concrete sub-class implementation uses the implementation information to store arbitrary configuration information for its own use.

This implementation information is intended to support two primary uses. First, it allows a document to utilize data sources external to the FIF. By default, a document serialized and returns its contents in the XML document returned by the `toXml()` operation.

These contents are then stored by the document manager. Alternatively, to utilize an external data source, a document implementation can specify the information needed to identify the data store and retrieve its contents in the implementation information, and return an empty XML tree as its contents. This allows the document manager to provide a consistent approach to identifying and retrieving documents while deferring document persistence details to individual implementations of the `Document` class. Second, it provides a container that document can use to store implementation dependent metadata that is stored separately from the contents of the document.

Sub-classes of the `Document` class must implement two operations for handling this local configuration information. The first is `getLocalConfig(config: Element)` which takes an `org.w3c.dom.Element` object and populates its XML structure with any necessary configuration information and returns the `Element` object. The second operation is `initialize(config: Element)`. This also takes an `Element` object, this time one which already contains the XML based configuration information as generated by `getLocalConfig()`, and extracts the information from this `Element` and configures the documents internal structures accordingly.

The `Document` class provides an operation, `filter(filter: FeatureList)`, that is used to initiate the feature identification process. This method should be overridden by sub-classes to process the contents of the document and pass the appropriate `Segment` objects to the `filter()` operation of the provided `FeatureList`. By default, the document class passes the root content segment to the FeatureList for filtering. This mechanism allows document implementations the flexibility to filter only certain segments of a document (for example, the body, but not the title or bibliographic information) or to provide different segments of the document to different filters. The details of the filtering operation are discussed in more detail in the section Feature Model.

Finally, the `Document` object maintains a `SegmentFactory` instance that governs the construction of `Segment` instances and hence controls the hierarchical structure of the document. Details of which `SegmentFactory` implementation to use and how to configure it are deferred to specific implementations of the `Document` class.

The `edu.tamu.csdl.documents.DocumentCollection` class represents a named collection of Document objects and provides a much simplified interface to the underlying search engine. This will be discussed in more detail below in Indexing and Searching. It is worth noting here that documents are not added directly to a document collection, instead they must first be wrapped with an `IndexFilter` that will be used to prepare the document for indexing. The `DocumentCollection` is also responsible for maintaining space on the file system for the storage of the indices as well as space for custom use by applications. The operation `getCustomDirectory()` will return a `File` object for a directory that can be used by an application for the storage resources related to this collection.

A full application using this framework will likely be based around multiple document collections. For example, if the *Cervantes Project* were to use the framework throughout the entire scope of the project there would likely be a collection used to store the texts transcribed from the *Princeps* and other early facsimile editions, one collection for bibliographies, one for the documents in the Sliwa collection, and perhaps several associated with the music collection. Others would be added as our holdings expand. Each of these different collections could then be filtered for different strategies and indexed to meet different research requirements. This allows the individual the framework to be tailored not just to the needs of the project as a whole, but also to the needs of individual sub-projects. Applications of the framework are responsible for determining what document collections are needed to most effectively represent the contents that application is working with.

The `edu.tamu.csdl.documents.DocumentManager` class provides an API for creating and managing document collections and for saving and retrieving documents. Each document collection is given a unique name (by the application that creates it). The `DocumentManager` allows applications to determine if a collection with a specified name already exists (using the `hasCollection(String)` operation), to retrieve a collection based on its name (using the `getCollection(String)` operation) and to create a new collection (using the `createCollection(String)` operation). The creation operation ensures that each collection name is unique. The manager also provides an operation, `getCollections()` that returns a set containing the names of all the collections that have been created.

The document manager also provides for the storage and retrieval of documents. Specifically it provides operations to save a document (`save(Document)`) and to retrieve a document based on its id (`getDocument(Long)`). Each document implementation is required to obtain a unique document id from the document manager by calling the `getNextDocumentId()` operation when an instance is first created. This id is then used to by the manager to uniquely identify the document for storage and retrieval purposes. The document manager uses the document management information stored in serialized form of the document to determine which concrete implementation of the Document class should be created when a document is restored and instantiates an instance of the appropriate class using the Java reflection API. Figure 4 shows the structure of the `DocumentManager`, `Document`, and `DocumentCollection` classes including the methods relevant to the above discussion.

Figure 4: A class diagram of the DocumentManager, Document and DocumentCollection classes

This content model for the FIF is designed to reflect the Ordered Hierarchy of Content Objects (OHCO) perspective on document content, which, despite its limitations, reflects the structure of XML, currently the dominant encoding strategy for textual resources [20]. Conceptually, the content model for documents is comprised of segments that may serve as content object, structural nodes in the hierarchy, or both. Each segment spans a (possibly empty) sequence of the underlying text of the document. Arbitrary attribute-value pairs may be assigned to a segment depending on the constraints enforced by the segment factory governing a particular document.

The API for the content model is provided by the `Segment` interface. This interface is intended to be implemented by applications as needed to support different document types (e.g., RDF, PDF, XML) while providing a common interface to the feature identification system. The `Segment` interface defines operations that support four major groups of functionality: assigning and accessing attribute-value pairs, appending, inserting and removing child segments, indexed based identification of sub-segments, and creating and inserting sub-segments. The first two of these groups are relatively straightforward. The second two are discussed in more detail below.

Each segment spans a (possibly empty) length of text. In addition to standard methods for navigating a tree structure, the segment interface defines three methods for navigating its hierarchical structure based on the underlying text. Each segment is aware of and responsible for manipulating only the portion of the total text of the document that it spans. Commands to manipulate the structure of the hierarchy (for example, creating a sub-segment with that starts at a particular index and ends at another) will often be initiated at a segment high in the hierarchy and passed to successively lower segments until the command can be completed. At each step down the hierarchy, the original indices specified in terms of the parent segment must be adjusted to reflect their relative position of the sub-segment to which the command will be passed. To aid in this, the `Segment` interface specifies a `getRelativeIndex(int, Segment)` whose first parameter is the index into the segment on which the operation was called and whose second parameter is a descendent of this segment. The operation returns the specified index value relative to the descendent provided in the second parameter. The second operation, `getTextSegmentAt(int)` in this group allows the terminal text segment at the specified index (relative to the segment in which the operation was called) to be retrieved. The third operation, `getPathToSegmentAt(int)`, returns a list of segments such that for any `i > 0`, `list[i - 1]` is the parent of `list[i]` and `list[0]` is the segment on which the operation was invoked and `list[size – 1]` is the terminal segment at the specified index.

The second group of operations that warrants further discussion deals with inserting and creating sub-segments. The `createSubSegment(name, attributes, start, end)` operation should be implemented to create a sub-segment that begins and ends at the specified indices. The sub-segment will have the specified name and be assigned the attributes contained in the second parameter. Similarly, the `insertEmptySegment(name, attributes, index)` operation creates a segment with the specified name and attributes at the specified index position that contains no textual content. Both of these operations leave the underlying text unchanged. Implementations of the `Segment` interface can use the `SegmentFactory` maintained by the document these segments are a member of to enforce the syntactical structure of the hierarchy. If something prevents the construction or insertion of a new segment (for example, if it cannot form a properly nested hierarchy or if it violates the constraints imposed by a segment factory) these operations should throw a `SegmentException`.

The framework provides a basic implementation of the `content model` that is suitable for representing XML documents. This implementation consists of three classes, the `BasicSegment`, `TextSegment` and `EmptySegment`, as shown in Figure 5.

The `BasicSegment` implements the `Segment` interface in a way that supports the structural nodes of the hierarchy. Instances of `BasicSegment` do not directly contain any textual content. Textual content must be contained within a `TextSegment`. This class also directly implements the `Segment` interface. The `TextSegment` class does not allow any attributes to be assigned or children to be added. It does, however, provide non-trivial implementations of the `createSubSegment()` and `insertEmptySegment()` operations that restructure a simple `TextSegment` into an appropriate hierarchical `Segment` tree. The third class, `EmptySegment`, provides an implementation for terminal segments which do not contain either text or children, but have a named representation and may have attributes. Empty segments correspond to the line break tags in the TEI standard and are a key strategy in implementing some of

the more advanced versions of the OHCO model, including those that view a document as a set of concurrent or overlapping hierarchies [42].

A segment factory is used in close conjunction with the content model to abstract the details of constructing new segments based on the context in which those segments will be used and to enforce syntactic and semantic constraints on the content hierarchy. The FIF provides an abstract class, `edu.tamu.csdl.documents.SegmentFactory`, that defines the interface for a segment factory. The goal of this abstraction is to allow a content model implementation to construct segment instances indirectly via the factory, thus reducing the degree of coupling between implementations of the content model and the tools that enforce constraints on the structure of that content model. This will allow a single content model to use a variety of approaches to defining the document structures that may be permitted for a particular type of document. This approach only partially reduces the coupling between these two areas of the framework. It would not make sense, for example, to use a `SegmentFactory` implementation that constructed PDF based segments in a document whose root segment was XML based. How particular `SegmentFactory` and content model implementations are paired is a decision that is left to the applications using the framework. They do this by implementing a `Document` that specifies a concrete implementation of the `Segment` interface to act as its document root and a `SegmentFactory` to govern how the document tree is grown from that root. A single `SegmentFactory` instance will govern all segments in a given content tree.

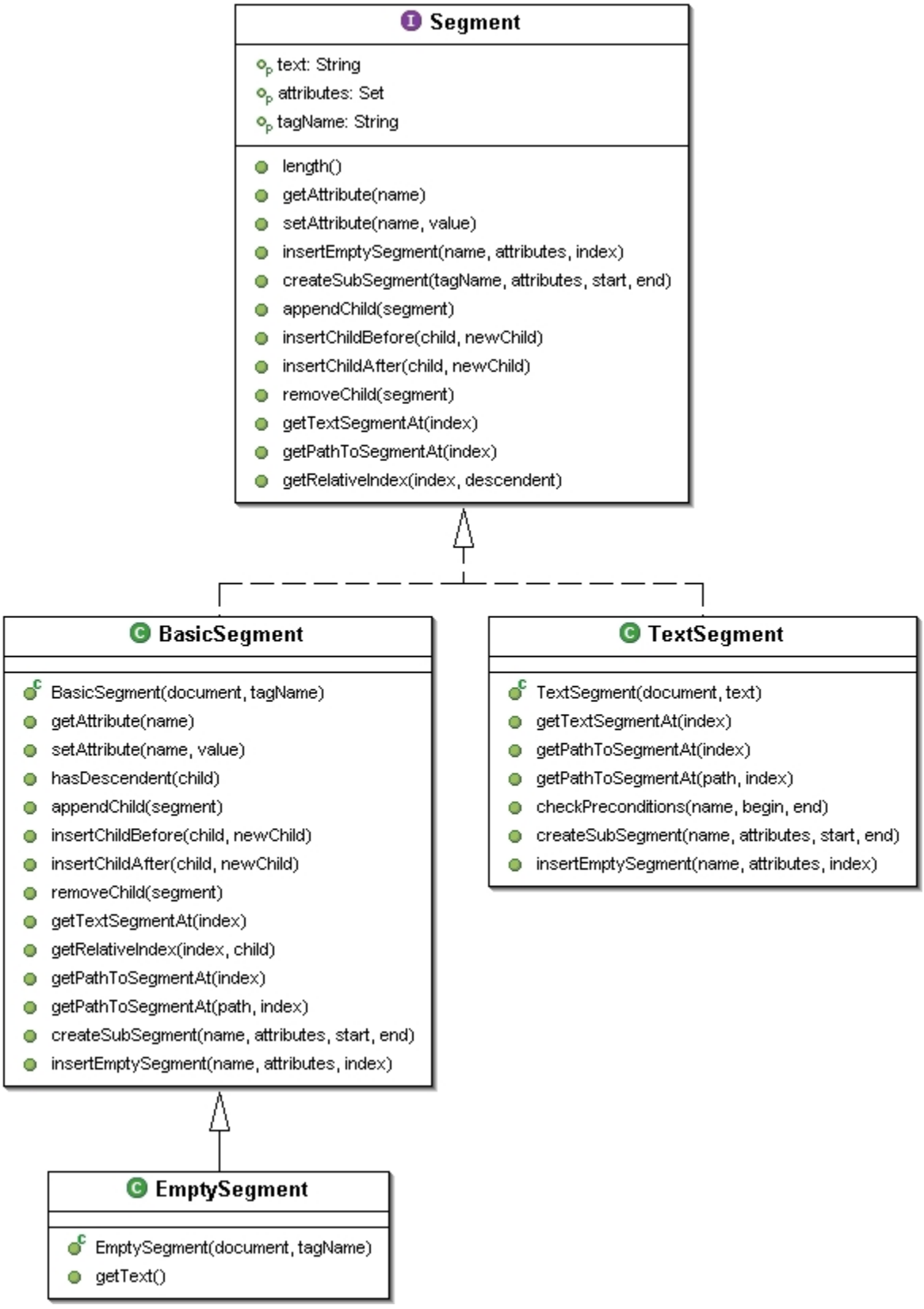

Figure 5: The Segment interface and default implementation

The `SegmentFactory` class defines three abstract operations for constructing segments and another three operations for checking the validity of content. The first three parameters of the segment construction operation are identical. The first parameter is the `Document` instance that the constructed segment will be a member of, the second is the name of the segment to be constructed, and the third is a `Map` of the attributes to be assigned to the segment. The final, optional, parameter specifies the content of the constructed segment. If no parameter is provided, the constructed segment will have no content. If a `Segment` instance is provided, that segment will be assigned as the only sub-child of the created segment. If a `List` of segments is provided, each segment in the list will be added as a child of the created segment, maintaining the same order as the list. The concrete sub-classes of `SegmentFactory` should implement these operations to examine the examine the information provided in these parameters together with any configuration information and instantiate and configure an appropriate concrete implementation of the `Segment` interface or throw a `SegmentConstructionException` if no such instance can be created.

The remaining three methods are intended to be called by the content model to determine if a particular modification to a content tree is valid. The `isValidChild(parent, child, pos)` operation is used to determine if a particular segment is a valid child for the provided parent at the specified index position (relative to the parent). The `isValidAttribute(segment, name, value)` operation is used to determine if a particular attribute name-value pair is valid for the specified segment. The `isChildRequired(parent, child)` indicates whether or not the specified child is required to be present for the parent segment to be valid. It is important, for example, to check whether a parent requires a segment to be present prior to removing it. If so, removing it would create an invalid tree (e.g., a header is required to have a title and an author). While the construction operations enforce the construction of segments with valid children, they do not check the context into which the constructed segments will be placed. Instead, it is the responsibility of the operation

The name of the descriptor corresponds to the name of the Segment instances that it governs. The `ChildDescriptors` describes: 1) the name of the child, 2) whether it is required or optional, and 3) whether multiple children of this type are allowed. The `AttributeDescriptor` specifies the name of the attribute and a regular expression pattern that the value of that attribute must match. The SDSet class serves to aggregate a set of SegmentDescriptors and provides XML serialization operations so that configurations can be saved to and loaded from files. A web-based editing tool is provided for this segment factory implementation, making it relatively easy to configure the structural constraints to be imposed on XML based documents. This tool is shown in Figure 7 and Figure 8.

Feature Model

The feature model provides the basic set of structures for analyzing the contents of a document or document collection and identifying features within those documents.

Feature Manager

The `FeatureManager` class provides a façade controller for the feature model. It allows for the creation and persistence of feature lists, when unique identification of features is important, it provides utilities for assign unique identifiers to `Feature` objects, and it supports the web-based tools by implementing the registration model for `Feature` and `FeatureWebPeer` classes. It is implemented using the singleton pattern, so there may be only one `FeatureManager` instantiated at any given time.

inserting the segment to call the validation operations of the segment factory to ensure that newly created segments can be added at a given place in the content hierarchy.

The FIF provides a default implementation of the `SegmentFactory` (in the form of the `BasicSegmentFactory` class) that supports relatively simple constraints on the structure of a document. This implementation, shown in Figure 6, is configurable either programmatically using the `SDSet` and `SegmentDescriptor` classes (both in the `edu.tamu.csdl.documents.impl` package) or via a web-based interface. Configuration information is be stored in XML documents.

The `SegmentDescriptor` class represents the description of a the structures allowed for a single segment. Instances of this class describe what children and attributes are allowed for segments of a given type. Each segment descriptor has the following properties:

- a name (corresponding to the name of the Segment that it describes)
- a list of `ChildDescriptors` specifying which child segments are allowed
- a list of `AttributeDescriptors` specifying which attributes may be attached
- a flag indicating whether text segments are allowed
- a flag indicating whether mixed textual and non-textual children are allowed
- a flag indicating whether or not the order of the children of the segment must be in the same order as the `ChildDescriptors` in the list of children.

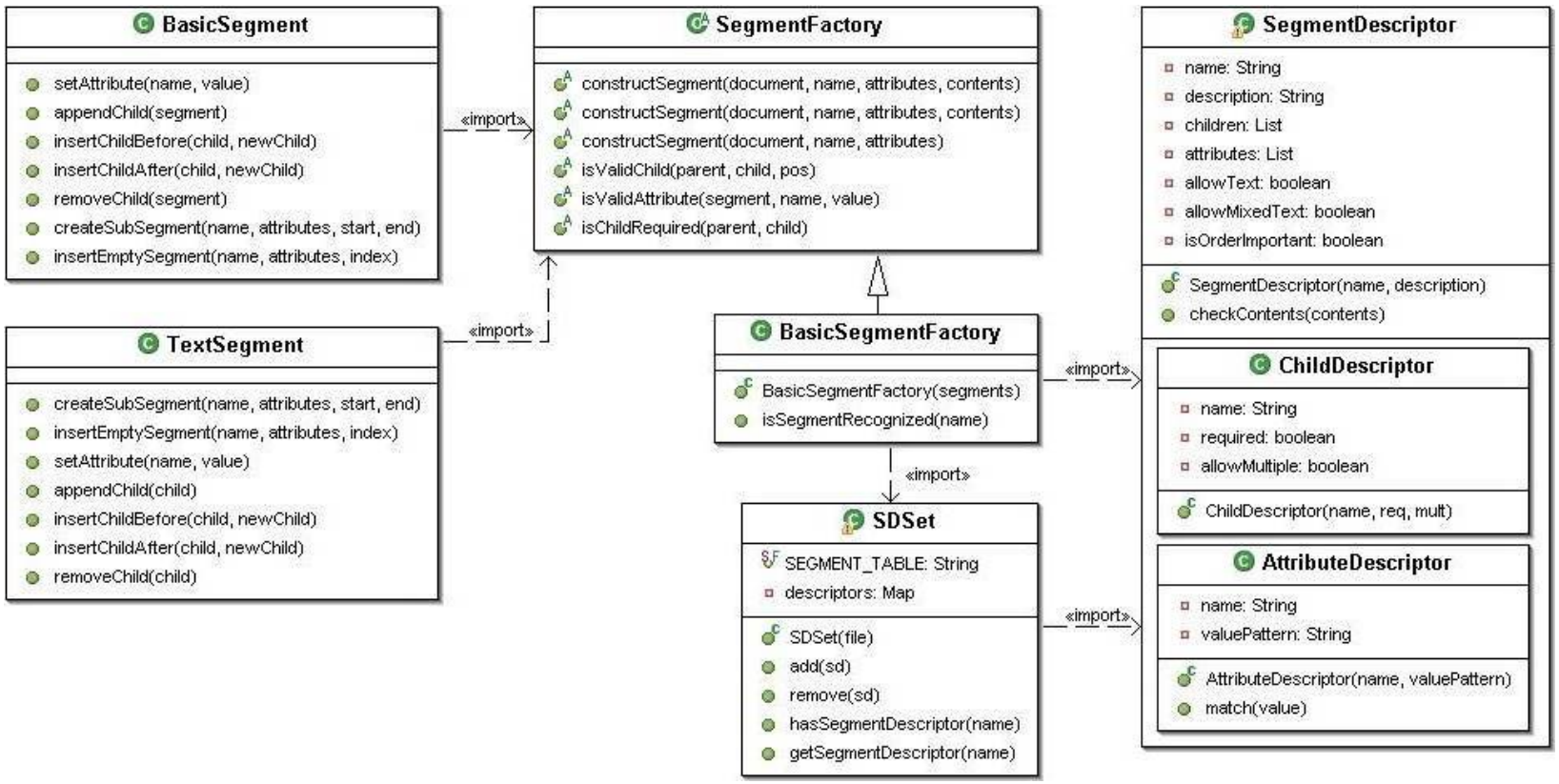

Figure 6: Class diagram of the default segment factory implementation

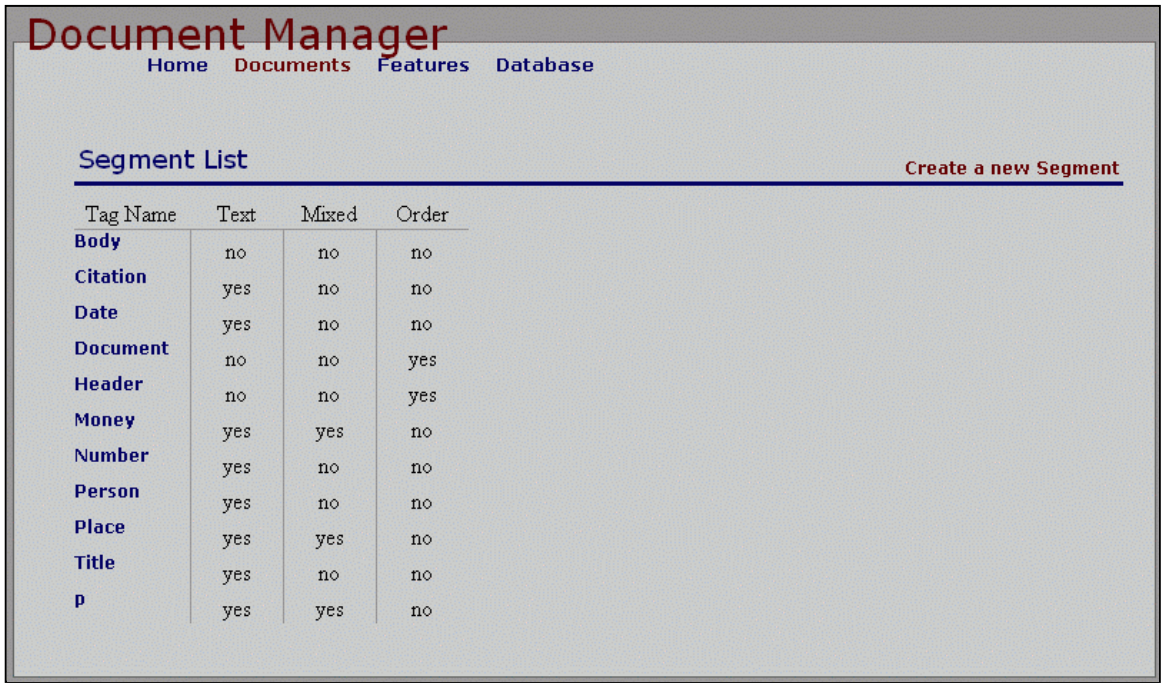

Figure 7: The segment factory configuration interface—segments in a simple document configuration

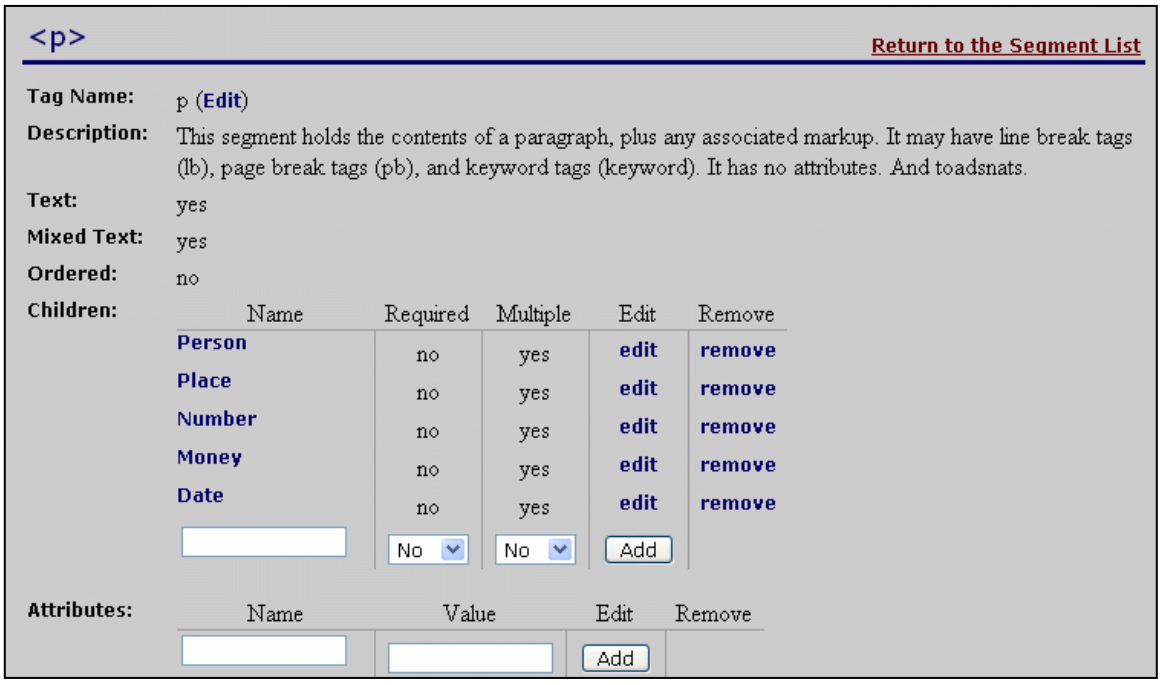

Figure 8: The segment factory configuration interface—editing the <p> segment

Features

The `edu.tamu.csdl.features.Feature` interface defines the API for classes that identify features in documents. This interface is intended to be implemented by applications extending the FIF in order to allow feature identification systems to craft custom algorithms or to employ standard algorithms in a variety of environments. Custom implementations can take advantage of specific knowledge about a set of documents, implement a new, state of the art algorithm, or meet highly the specific needs of a particular research project.

A `Feature` object embodies both an algorithm for identifying a particular type of feature in a text string, and configuration information that provides more information about the particular features that should be identified. This configuration information may or may not be used directly by the feature identification algorithm. For example, a feature class that identifies the names of people might have configuration information that specifies the name of the person to be identified and also a short biographical sketch–information that can be displayed to readers, but is not intended to be used in the identification algorithm. Once identified in a text, a feature will be marked by creating a `Segment` that spans the text at which the feature was identified. A `Feature` object specifies the name (`tagName`) of the segment to be created and a set of attributes that should be applied to that segment. As a rule of thumb, the name and attributes should provide enough information so that the original object that identified the feature can be restored and any relevant details about the identified feature can be retrieved.

The main functionality of the `Feature` interface is specified by the `Feature.match(text: String)` method. This method should be implemented to accept a text string and identify all instances of a feature within that text string. The function returns a `SortedMap` whose keys are the `Integer` valued indices into the text at which each identified feature was matched and whose values are instances of the identified feature suitable for marking that feature in the underlying text. The `Feature` objects returned in this `Map` must have three characteristics: their `value` property must

exactly match the surface form of the feature in the text `String` in which it was identified, their attributes should specify any important information about the identified feature that may be needed by an application, and finally, the `tagName` property should specify the appropriate segment name for the identified feature. Since specific implementations of the `Feature` interface will often rely on statistical algorithms, gazetteers, or pattern matching, it is likely the case that this information will not be "known" by the `Feature` object that identified a particular feature in a text. In this case, the `match()` operation should construct a new, simple `Feature` object that is specifically configured to represent this information. In most cases, the `BasicFeature` provided as a default implementation of the Feature interface will be sufficient for this task.

The `BasicFeature` class provides a default implementation of the `Feature` interface and performs two main tasks. First is can be used as a base-class for more sophisticated `Feature` implementations, providing a basic implementation supporting the `tagName`, `value`, and `attributes` properties. Second, it can be used as a simple `Feature` instance to represent the concrete features identified by more complex feature identification algorithms. It provides simple matching for the exact string stored in its `value` property.

Feature Lists

The `FeatureList` class provides the primary unit for grouping and working with individual `Feature` objects. It is objects of this class that are passed to the `Document.filter()` operation in order to initiate the feature identification process. This class is also responsible for the persistence of the `Feature` objects it contains. Note that this is not like the Document class where the `DocumentManager` rather than the `DocumentCollection` is responsible for the persistence of Document objects. While it is possible to create a sub-class of the `FeatureList`, this class is intended to provide all of the core functionally needed for matching groups of features and sub-

classing should not be necessary. It provides a number of operations for maintaining the list of features including operations to add and remove features and to retrieve features based on their id or their place in the list. The most significant operations of the `FeatureList` class are the `filter()` and the `getIndices()` operations. Both operations take a `Segment` object and then attempt to find any features in the list that match the text in that `Segment`. The filter operation returns a modified version of the Segment that was passed to the operation. The returned Segment object will be restructured so that all identified features have been appropriately marked within the Segment. The getIndices() operation will identify all matching features within the list, but, rather than marking those features within the Segment, will return a `SortedMap` whose keys are the `Integer` valued indices into the text at which each identified feature was matched and whose values are instances of the identified feature. This operation is primarily intended to be used by the indexing and searching sub-system.

Displaying and Editing Features

The task of selecting which features to identify is critical to the success of a feature identification system and depends heavily on expert knowledge of the subject area of a particular document collection. While implementing sophisticated algorithms for feature identification is beyond the ability of most “corpus editors,” skillfully selecting and applying these algorithms to identify important features within a collection is a task best performed by subject matter experts.

For example, within the Sliwa collection, certain phrases might indicate that a document is of potential interest because it discusses legal, military, financial, familial, or literary matters. A scholar familiar with the collection and with the relevant research questions would be able to identify such phrases quickly, whereas I, lacking both the scholarly background and knowledge of Spanish would find this difficult or impossible. Similarly, automatic algorithms might be able to identify certain key information that could be further refined by focused hand editing. For example, short biographical information might be added by hand to key individuals in a list of automatically identified people.

Again, it is the “corpus editors” with extensive training in the subject domain who have the expertise to prioritize the strategic areas to be enhanced by hand editing.

The underlying infrastructure to support these and other similar tasks is already present in the design of the system. Implementations of the `Feature` interface can be provided that would allow detailed information to be added–either automatically or by hand. `Features` can be grouped into lists allowing features specific to a particular analytical domain (e.g., music, finance, military) to be associated. Using these elements of the underlying system, however, requires extensive programming experience, a rare luxury for humanities projects. Consequently, I have developed a tool to make these elements of the underlying system architecture accessible to domain experts–without requiring extensive technical expertise. This web-based tool allows editors to create new lists, add and remove features from lists, and edit the configurable elements of a feature (e.g., the name, description, date of birth and death, and alternate names of a person). This tool also assists in displaying information about features in a web environment.

This tool for displaying and editing features is based on the model-view-controller (MVC) pattern. Figure 9 shows this MVC architecture and the steps involved in editing a feature. The `Feature` objects implement the data model that is to be displayed to the users, the view is provided by a set of JSP pages, and updates to the model are processed by the `FeatureControlAction` class that serves as the controller.

One of the great strengths of the feature model is the degree of flexibility that is supported for implementing the feature classes. This flexibility becomes a challenge when trying to present and edit those features via a single, web-based interface. Since the details of the `Feature` implementation are not known in advance, the view cannot be connected directly to the model. Instead, an adapter class, the `FeatureWebPeer`, is used. The `FeatureWebPeer` classes (or just peer classes) are designed to interface with a single `Feature` instance and generate HTML fragments that can be used to display a title, brief and long views of the contents, and form elements for editing the

modifiable contents of the `Feature`. The peer classes are also responsible for processing the HTTP response generated when this form is submitted and updating the underlying `Feature` as appropriate.

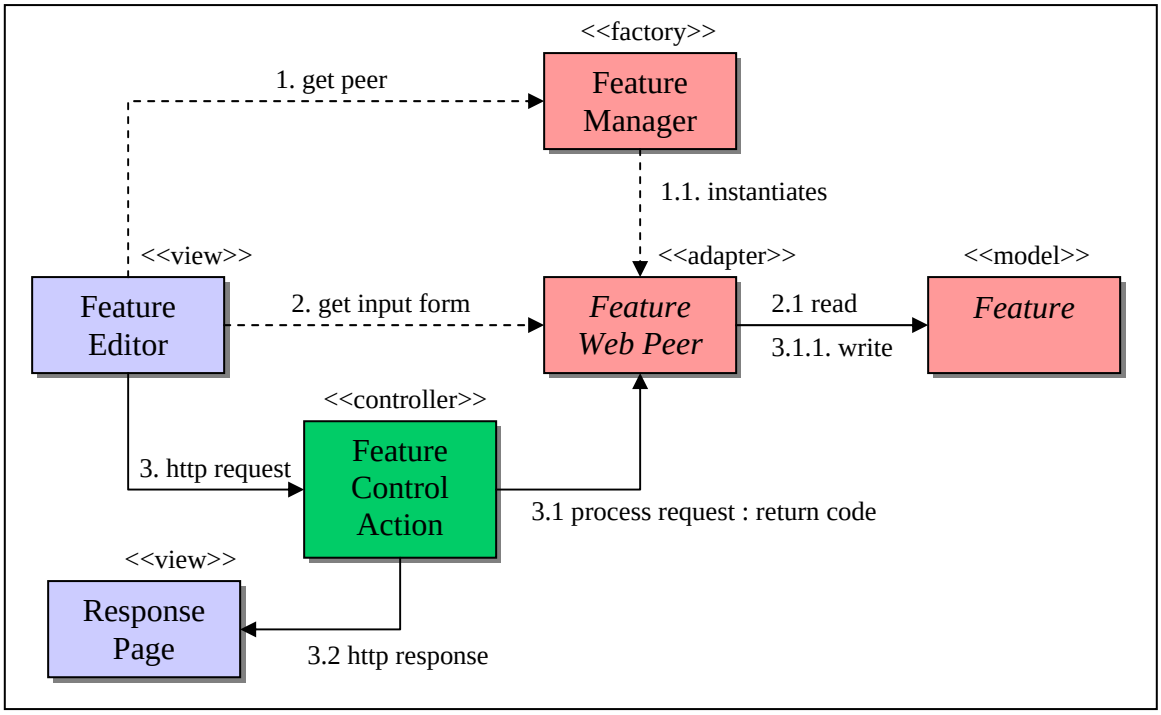

Figure 9: The procedure used by the web-based feature editing tool

Having used an adapter class to mediate the connection between the view and the model components, the next challenge is how the view will obtain instances of the appropriate adapter classes. To support this task the `FeatureManager` implements a registration model that allows information about concrete implementations of the `Feature` class to be registered with the `FeatureManager`. This information includes a human readable name and description for the class. More importantly, it allows a `Feature` implementation to be paired with a `FeatureWebPeer` implementation that will be used to adapt instances of that `Feature` class for web-based applications. A web-based

user interface is provided to support the feature registration process. Once a `Feature` has been registered, the `FeatureManager` serves as a factory class that instantiates an instance of the appropriate peer class given an instance of a registered `Feature`. While not all `Feature` classes used by an application need to be registered, all such classes that need to be made available via the web interfaces should have a peer class implementation and be registered.

To edit a particular feature, then, the user first selects a feature to edit. This will bring the user to the feature editor, a JSP generated page. This page first obtains an instance of the appropriate peer class from the `FeatureManager` (step 1 in Figure 9). Having obtained the peer, the feature editor queries it to obtain an HTML fragment containing the contents of the input form to be used to edit the feature (step 2). In doing this, the

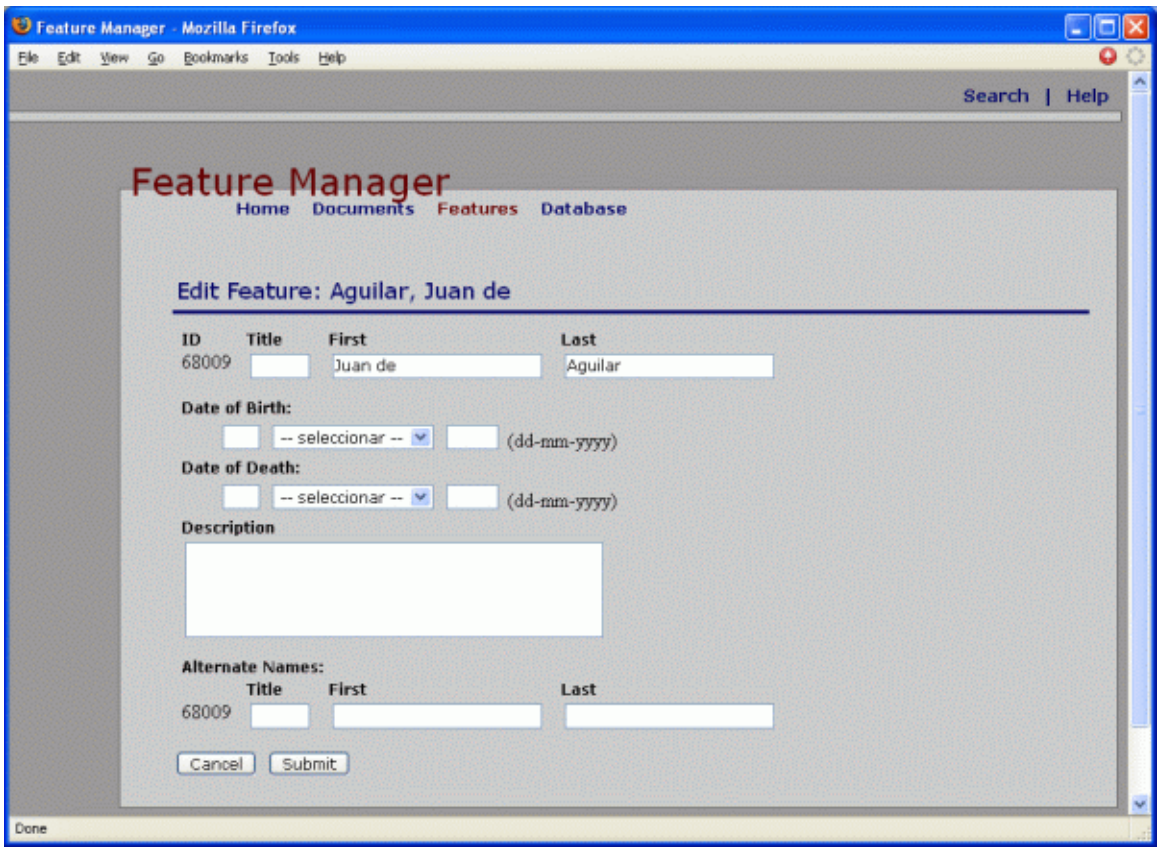

Figure 10: Editing a SliwaPersonFeature using the feature editor

peer reads the current state of the feature that it is representing and builds the HTML fragment to reflect this (step 2.1). Figure10 shows the feature editor being used to edit a `Feature` class designed to represent people in the Sliwa collection. The user then makes any necessary edits to the feature and submits the form (step 3). This posts the edits to a Feature Control Action that passes the request to the peer object (step 3.1). The peer is then able to process the information in this request and update the underlying feature object as appropriate (step 3.1.1). The peer returns a code that indicates to the controller what response action should be taken, for example, to display the update feature or continue editing. Based on this return code, the feature control action returns an appropriate HTTP response object (step 3.2). A similar procedure is followed for displaying a title for the feature or its contents. The primary difference is that step three, submitting the HTTP request, is not needed.

Indexing and Searching

Once key features have been identified within a document, an application will need to be able to index the document based on these features, as well as on the full text of the document. The specific indexing needs will vary significantly between applications making the task of indexing and searching one of the most open-ended aspects of the framework. There are three main requirements: 1) provide a simple, easy to understand API for searching document collections, 2) support extensions of its indexing scheme that allow applications to specify the textual components and identified features that should be included in the system indices, and 3) use an existing search engine.

Many robust and well-designed search engines have been build and made available in the open source community. The FIF is designed to work in close conjunction with the widely used Lucene search engine. Lucene creates and accesses its indices via `IndexWriter` and `IndexReader` classes. When instances of these classes are constructed, they are configured to write to or read from an index stored in a particular directory. New documents can be added to the indices by creating an instance of the Lucene `Document` class (which I will also refer to as a ‘document proxy’) and passing

it to the `addDocument()` operation of an `IndexWriter`. The document proxy defines a set of fields that will be indexed and can subsequently be searched using an `IndexReader`.

The `DocumentCollection` class provides a wrapper around the search engine that abstracts many of the details of working with the Lucene search engine and provides a few utility functions. As stated previously, a named document collection is responsible for maintaining a reserved space on the file system for information pertaining to the collection. Part of this file space is used to store two indices, one for the full text of the documents in the collection, and one for the features identified in the documents.

As documents are added to the collection, they are not added directly, but rather must first be wrapped in a class that implements the `IndexFilter` interface. The index filter should be implemented by the application to read a particular document format, process that document to identify any of the features that will be indexed, and construct two document proxies, one for use in the full text index, the other for use in the feature index. The details of how these document proxies are constructed is differed to the class that implements the `IndexFilter` with the exception that document collection will add a special field that allows retrieval of the document proxy based on the document id assigned by the `DocumentManager`. The proxy for feature based indexing should be built in such a way that each field contains a list of the ids of the features contained in that document. For example, a document in which the names of people have been identified, might have a feature index field called ‘people’ that contained the id values of all the features representing people found in that document. The IndexFilter class defines two operations to retrieve these proxies, `getTextDocument()` and `getFeatureDocument()`. The interface also defines a third method, `getRootDocument()`, that returns the FIF `Document` object that this particular filter instance is processing.

Documents are indexed as they are added to a collection. Two methods are provided to facilitate access to these indices. The first, `search(query, field)`, searches for the provided query string in the specified field. A ranked searching algorithm is used. The second, `search(keywordIds, field, requireAll)`, searches over the feature index. It will look in the specified field for all documents containing the feature ids provided in the `keywordIds` list. The `requireAll` parameter indicates whether all of the ids in the provided list must be present in a document in order for the document to match that search. This allows relatively simple, boolean queries. More complex results may be obtained by using the union and intersection operations provided by the `HitList` objects returned from multiple searches.

SLIWA COLLECTION APPLICATION

The second major task of my thesis work was to apply the framework to create a web-based interface for a collection of 1700 historical documents assembled by Prof. Kris Sliwa. This interface provides support for automatic link generation and information visualizations based on people, places, and dates. It also supports the recognition of monetary units and other quantized information (e.g., distances and weights). This application is sufficient to meet the basic needs of automatically inter-linking the texts and can be easily extended to meet future needs of the research community. In addition to meeting tangible needs within the *Cervantes Project*, it serves as a test case for the FIF and as a concrete example of how the framework is intended to be used.

This interface provides two primary points of access to the collection. The first is a timeline interface. It provides a simple bar chart showing the distributions of the documents over time. Selecting a bar takes the user to a more detailed view of the time period. Once the chart displays documents as single years, clicking on the bar for a single year brings up a display listing all documents from that year. Figure 11 shows this interface as the mouse passes over the bar that spans the years 1583 – 1592. The second point of entrance is a browsing interface, shown in Figure 12. A browsing interface is provided for both the people and places identified within the collection. Next to the name of each person or place, the number of documents is shown to give the reader a heuristic for determining how interesting it might be to examine the documents for that person or place more closely. Upon selecting a person to view, the reader is then taken to a page that presents the resources available for that person–currently, this includes a list of all documents in which the specified person has appeared and a bar graph of all documents in which that individual has been found as shown in Figure 13.

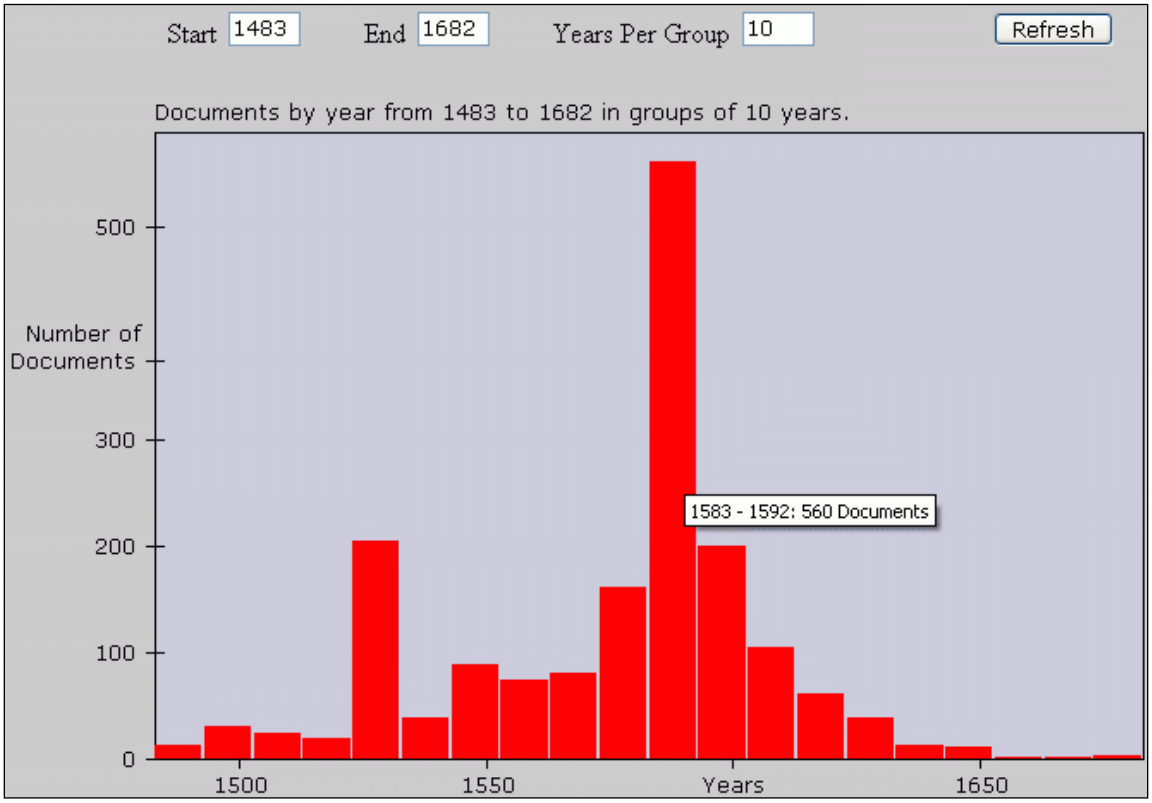

Figure 11: Timeline browser

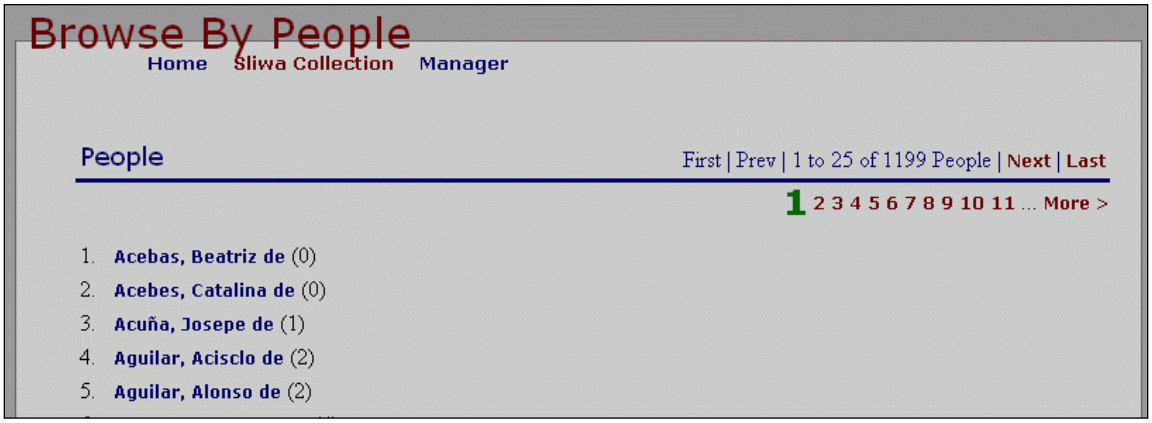

Figure 12: Browsing interface

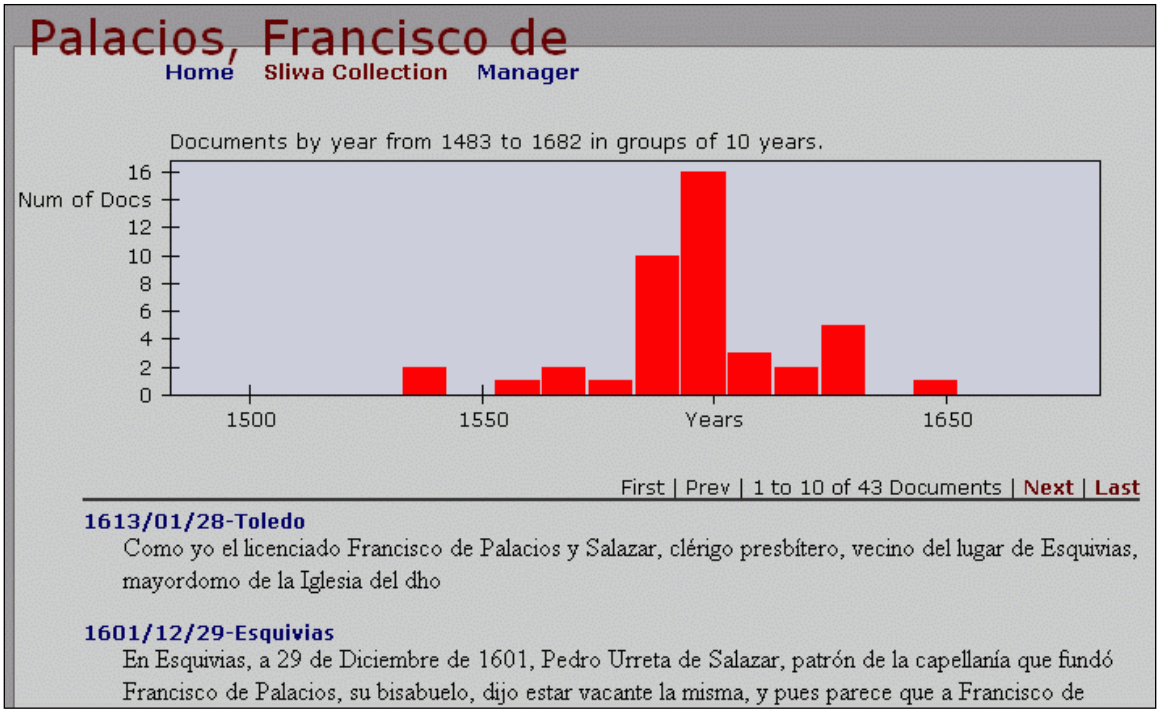

Figure 13: Resources page for a person showing a timeline display of the documents referring to this person and a browsing interface for those documents

Once the user has selected an individual document to view, through either the timeline or browsing interface, that document is presented with two types of features identified and highlighted. The first type is features used to automatically generate navigational linking such as people and places. The second type includes features used to provide generic information such as dates and monetary units.

The browsing and timeline interfaces serve as examples of the wide range of services that can be built onto the underlying framework. Both of these interfaces are built on top of the searching facilities provided by the FIF. The browsing interfaces allow a user to select a feature to be searched for by presenting the contents of a feature list and then displaying the results of a search for that feature. The timeline visualization searches for documents by year, aggregating them into user-defined groups (e.g., each bar in Figure 11 shows the documents in a ten-year period). More complex interfaces are easily imaginable and include filtering search results to perform more sophisticated document queries, timeline visualizations based on event and topic detection, and geospatial

visualizations. New interfaces can be quickly added to the system as described later in this section.

Application Architecture

The architecture of the application, shown in Figure 14, can be divided into three main sections: an implementation of the document model (with supporting tools), an implementation of the feature identification system (with supporting tools) and a JSP and Struts based user interface. Additionally, a manager class provides a façade controller to assist in working with the feature and document packages and to interface with FIF. The web-based interface relies heavily on this manager class to provide users with the ability to navigate the collection based on the identified features, to browse by the people and places found in the collection and to interact with information visualizations.

The Document Package

The document package provides three major services: 1) a parser that processes the original document, converting the text only output of a word file into an XML representation and importing it into the system, 2) an implementation of the `Document`

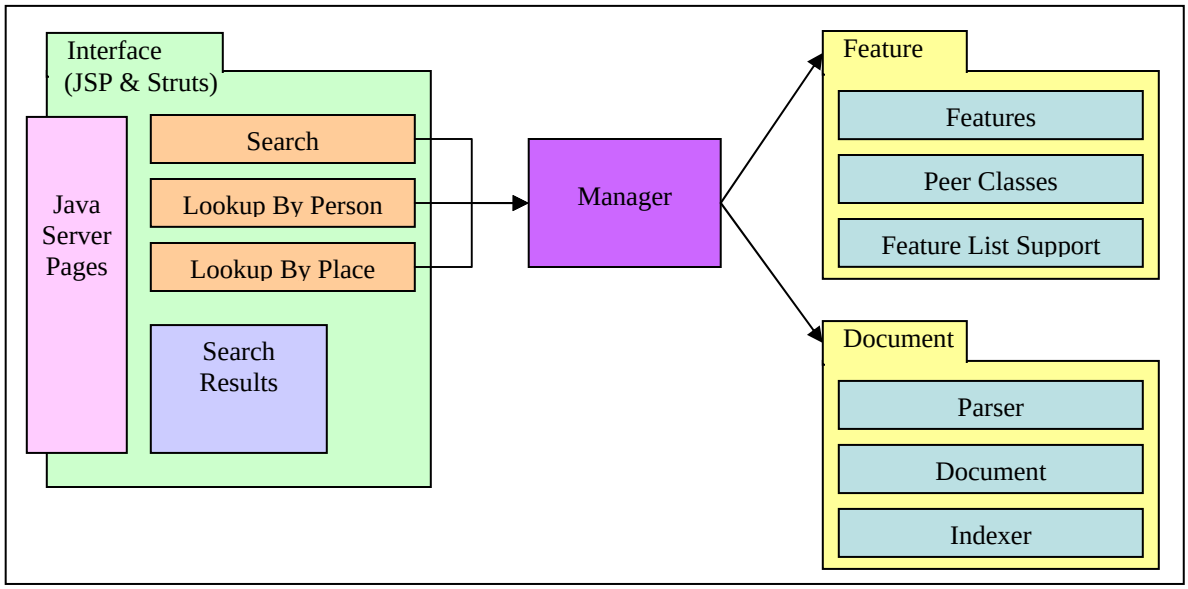

Figure 14: The architecture of the Sliwa application

class tailored to represent these documents, and 3) an implementation of the `IndexFilter` interface that specifies how these documents will be indexed.

The implementation of the `Document` class, `SliwaDocument`, supports the explicitly encoded structure of the document. The textual representation of the document enables relatively simple automatic routines to distinguish the textual content of documents from their titles and citation information. This document provides operations to set the title and citation information of a document and to append paragraphs to the document as they are read from the original source file. This allows the parser to interact with document objects in ways that closely reflect the semantics of the document's structure. The `filter()` operation implemented by the `SliwaDocument` takes advantage of this structure by providing only the document body, not the header or citation information, to be filtered by provided feature lists.

The `Indexer` class implements the `IndexFilter` interface and processes a `SliwaDocument` for indexing. As mandated by the `IndexFilter` interface, it returns Lucene `Document` objects suitable for full text and feature-based indexing. Each of the Lucene documents returned by an indexer contains two un-indexed fields, one containing the title of the document and one containing a summary (the first, suitably long sentence) of the document. This allows the user interface to retrieve basic summary information directly from the Lucene document returned by a search, thereby eliminating the need to restore the full document from the file system until a user chooses to read it. The procedure for establishing the full text index is straightforward– the indexer simple adds a field that contains the text of document body (again, excluding the title and citation information). For the feature-based indices, the Indexer creates fields for identified people, places and dates. Both the people and places are identified using the corresponding feature lists (discussed in more detail below). The titles of the documents explicitly encode the date the document is thought to have been written. Accordingly, the indexer extracts the date of the document from the title, rather than attempting to identify dates within the body of the text.

The Feature Package

The algorithms for identifying features in documents are implemented within the feature package, along with a set of tools for initializing, configuring and using those algorithms. There are four major types of features involved in this package: person names, place names, dates, and monetary units. Each is discussed below along with the tools that support it.

Person and Place Names: Identifying the names of people and places is one of the most clear requirements for supporting the Sliwa collection. Support for this task is implemented by the `PersonFeature` and `PlaceFeature` classes. These classes are intended to serve not simply as algorithms to identify names, but also as data objects that maintain and store information about the people and places in the collection and that make this information available to the rest of the application. Features representing people contain information about a person's first and last name, title, date of birth and death, alternate forms of his name and a biographical description. Features representing places contain information about the context (i.e., which of several places with that name is this one) and a description of the place.

Instances of the people and places features are grouped into `FeatureLists` named 'Names' and 'Places' respectively. Initially, these lists are derived from an index to an earlier version of the book from which these documents were derived. Manager classes are provided to support the process of reading these files and for constructing and subsequently retrieving the `FeatureLists`. While these hand-generated indices provide a good starting point for building the lists, they contain notable omissions and do not fully reflect the current state of the document collection. To allow editorial enhancement of these lists, I have implemented peer classes for both of these features. These peer classes work in conjunction with the web-based feature-editing tool discussed in Displaying and Editing Features to allow an editor to update the data stored in the feature (notably, this includes specifying alternate names for people, such as

“Miguel de Cervantes” for “Miguel de Cervantes Saavedra”) and to add new people and places to the lists.

Both of these classes implement an identification algorithm that matches the name of the person (including alternate forms) or place. They use a fuzzy matching scheme that mitigates most of the effects of spelling variations present in the collection. Once a feature has been identified in a document, a new instance of the BasicFeature class is constructed to represent the identified feature. This new feature contains the exact text sequence that was found in the document and has an id that corresponds to the id of the person or place that was identified. This id will be assigned as an attribute of the segment that is created to mark the identified name and can be used to retrieve the feature that represents the identified person or place.

Dates and Monetary Units: Two feature classes are provided that rely heavily on identifying numbers in the text; they are `DateFeature` and `MonetaryUnitFeature`. The `DateFeature` class is used to identify dates that are spelled out in the text of a document. The `MonetaryUnitFeature` class is a bit of a misnomer. Originally intended to be used to identify units of money (reales, maravedis, and ducados), it can be used to identify any quantifiable units including weights, distances, sizes, volumes, etc. Unlike the name and place features, neither of these classes is intended to specify concrete, retrievable, entities. Instances of the `MonetaryUnitFeature` can be configured to specify what type of unit they should identify. The `DateFeature`, on the other hand, takes no configuration parameters. It simply matches dates in a document.

These two features rely heavily on the `NumberParser` class. This class serves two main functions. First, it constructs a regular expression pattern that can be used to match numbers spelled out in Spanish (e.g., trece mil doscientos y ocho). This pattern supports the irregular spelling found in this collection. Second, given a string that was matched by this pattern, it parses that string to determine its numerical value. In addition to

recognizing numbers that are spelled out in the text, this also recognizes Arabic numerals. This pattern is then used by the date and monetary unit features to construct more complex patterns that match their respective features in the text. Once matched, they call the parser to identify the numerical values in the identified feature.

Once an instance of one these classes has identified a feature in the text, its behavior becomes similar to the person and place name features. It constructs a `BasicFeature` that specifies the exact text matched, and sets attributes that indicate either the day, month, and year of a date or the quantity and unity of a monetary unit.

The Manager

The document and feature packages are connected by a façade controller, the `Manager` class. This provides methods to execute the document parser and to construct the feature lists from the provided indices. Once the documents are parsed from the source data, the manager retrieve a list of the ids for the newly created documents, retrieves them from the FIF document manager, creates a new document collection and adds each document to that collection, first wrapping it with the indexer. While the indexer utilizes the feature lists to identify names and people, the identified features are not marked in the documents themselves. Those documents are stored just as they were extracted from the source files. This will allow future applications to implement different feature identification schemes using the same documents.

The manager class provides two methods for obtaining documents in the Sliwa collection. One returns a `Document` object with the relevant features marked in this document. Attempting to re-run the full feature identification process each time a document was requested would be too time-consuming. To avoid this, the manager uses the feature indices to retrieve the ids of the features that have already been identified in this document (the `getPeopleInDocument(docId)` and `getPlacesInDocument(docId)` operations accomplish this). These features are added to a new temporary feature list along with the features needed to identify

monetary units and dates. The document is then filtered using this temporary feature list, causing these features to be identified in the document. A second operation uses this marked document object along with an XSL style sheet to produce an HTML encoded representation of the document suitable for display by the interface.

The manager also provides a number of operations to execute search requests tailored to the semantic content of this collection. These include searching for documents by year, year and month, or year, month and day. It also includes searching for documents that contain an identified person or place (based on the feature id of the person or place). These search operations return instances of the `HitList` class. Most notably, the `HitList` class provides operations to support the union and intersection of search result sets, thus allowing complex filtering operations to be performed with the results returned by these searches. For example, a list of all document in which Miguel de Cervantes and Ruy Dias appear can be obtained by retrieving all documents for each person and then calculating the intersection of these two sets of documents. Since the documents themselves do not need to be retrieved from the file system, (the Lucene document proxies are used) these operations are relatively quick. In addition to searching for documents, the `Manager` class also provides two operations previously mentioned that allow the application to retrieve the people and places that have been identified in a given document. These relatively few operations provide a powerful mechanism for working with the contents of a large collection.

The User Interface

The user interface described in the introduction to this section is implements as a set of searches over the Sliwa collection. It is composed of a series of modules (Struts `Action` classes) that call the manager class and manipulate documents. These include modules to search the full text of the collection, and lookup documents containing certain people or places. It also contains helper modules, such as the search results module that can be used to manipulate search results. These modules are used by a JSP/HTML based user interface. This design results in a highly modular interface

component that can be quickly understood by future developers and (relatively) easily maintained or extended.

Extending the Application

The strength of this approach to designing the system, both the framework and the application that uses the framework, is the flexibility it allows for extending applications to meet new requirements. In this section I will describe how the system can be extended using the Sliwa application as an example. In particular, I will focus on how new research questions can be addressed and how new interfaces and collections can be added as modular, interoperable components.

Addressing New Research Questions

Adding support for new research questions can be divided into three major tasks:

1) Analysis: How can this question be expressed in terms of features within a document and subsequent operations on those features? This includes determining what portions of a document constitute features, how those features might be arranged into feature lists, and how those features should be processed, indexed and presented to readers after they are identified within the text.

2) Design: How can these features be identified within a particular document collection? This includes designing the architectural components needed to effectively present the needed information to readers and the tools to query the search engine.

3) Implementation: Implement the designed solution.

To illustrate this process, consider how the existing Sliwa collection might be extended to meet two new research questions. First, what types of documents do Juan de Cervantes and Miguel de Cervantes appear in together? Second, what happened in or near Valladolid?

To answer the first question, it is first necessary to determine how document types can be recognized. There are many sophisticated algorithms for document clustering or classification, but, within the Sliwa collection there is a simple, feature based approach that is likely to be sufficient. Key phrases could be identified that indicate that a document falls under a particular category–for example, military records, legal proceedings, commerce, birth certificates, etc. A subject area expert could quickly identify both the types of documents that are of interest (presumably different experts might be interested in different document types) and the phrases that indicate that a particular document is of an instance of one of these types.

Now that the question has been restated in terms of a feature identification task, the next step is to design a solution that will solve this question. In this case, the bulk of the infrastructure is present in the Sliwa collection. A new feature implementation will be needed to recognize phrases, but this can be implemented as a simplification of the place feature. These features will then be grouped into feature lists that represent the document types. This entire process of specifying which phrases should be matched and how those phrases should be grouped into feature lists can be accomplished by subject experts using the feature editor. Since these feature lists will be relatively small, the document types can be identified when documents are retrieved, requiring no indexing. Otherwise, if indexing the document types is desirable, the indexer for the collection will need to be extended to accomplish for this and the collection will need to be re-indexed. The manager component can then be extended to provide operations that will identify the type of a single document, group search results by their document types, or filter search results to include or exclude documents of a particular type. Once this is done, all that remains to answer the original question is to design interface components that allow a reader to retrieve documents which contain both Juan de Cervantes and Miguel de Cervantes and then call the appropriate extensions of the manager to determine which document types appear in the search results.

To begin to address the second question, we first notice that places are already identified within the Sliwa collection. What remains is to determine how the physical locations of places can be specified and how proximity to a particular place will be determined. Geo-referenced digital libraries are relatively common and gazetteers are available to assist in identifying the physical location of a place. This will require the place feature to be extended to include information about the location of a place and the place identification algorithm to provide disambiguation between different places with the same name (or the same place with different names). Once this has been done, the next step is to determine a heuristic for the notion of 'nearness.' This might be simply within a specific radius of the city of Valladolid. Alternatively, this heuristic might consist of a more complex analysis of the documents in the collection to determine a natural clustering of documents roughly centered on Valladolid. Once location information has been added to place features and a heuristic for 'nearness' has been selected and implemented, the user interface needs to be extended to find all places that meet the criteria for being 'near' Valladolid and then retrieve the documents in which those places are mentioned.

Reflection on these two examples we can make a few observations about the process of extending applications. First, applications can be extended in a modular fashion to address specific research needs not originally envisioned when the application was created. Using this framework, tools to answer these new questions can be added onto existing applications in ways that enhances the functionality of both the new and old tools. Second, these extensions are simple and non-destructive. They are simple in the sense that designers and developers can focus on the details that most directly relate to the problems to be solved rather than designing the tools for structuring the collection, storing and retrieving documents, searching the collection, etc. They are non-destructive in that the solutions required to answer these questions can be implemented without modifying existing functionality. Feature markup is not stored with the documents (or, more accurately, while it is possible to store feature markup with the document, this is not necessary and in most cases it should not be done) so these new solutions need not worry about interfering with existing applications or being interfered with by future

application. Finally, using the framework reduces rather than removes a humanities project's dependence on software developers. Software development is an expensive and time consuming task and it is critical for humanities scholars to use their limited resources in this area as efficiently as possible. This framework implements the overall structure of a feature identification system, allowing developers to focus on designing and implementing tools that meet highly specific needs while reusing general purpose tools. With careful planning, many of the more straightforward tasks (such as identifying the types of documents that are of interest to a particular scholar and the key phrases that indicate the type of a documents) can be implemented so that most of the work can be accomplished by humanities scholars with little or no technical training. Taken together, these three factors allows new tools to be designed and developed rapidly in response to the needs of readers and researchers.

Adding Interfaces and Collections

Digital libraries of any interesting size will have both many different document collections and many user communities. Each of these user communities is likely to come to the library with a variety interests and needs. Accordingly, a critical dimension of extensibility for applications of the FIF is their ability to interact with multiple collections and to be presented to readers via multiple interfaces. The current Sliwa application can be divided into three main parts. First is the collection itself, including the documents, custom features, and feature lists. The collection is created, maintained, and manipulated by an application layer that implements the basic functionality needed to support the research needs of the various user communities. Readers then interact with a thin interface layer that translates their actions into commands to the application layer and does some (minimal) processing of the results. This interface layer is implemented as a web-based application. This is diagrammed in Figure 15.

To understand how this might be extended to account for multiple collections and interfaces, consider, by way of example, how the Sliwa application might interact with an implementation of the FIF for more elements of the *Cervantes Project*. A natural

addition to the current tools would be to integrate the biographies we make available on our site with the primary source historical documents in the Sliwa collection. This would This could be accomplished by creating a new collection, along with the appropriate supporting tools. Since it is likely the features can be applied to both collections, the primary extensions needed for the document collection is an implementation of the document model that is sufficient for the much longer biographies. Once this is done, the application and interface layers can be extended to provide support navigating between the biographies and the primary historical documents that relate to them. At this point, one could imagine a wide range of potential enhancements that could be implemented (for example, event detection).

Adding and integrating a new collection, is only one aspect of extensibility. At the other end, it might be desirable to add a new interface to a particular application. One such possibility might be a windows based graphical user interface to this collection, perhaps integrating it with the interactive timeline viewer (ItLv) [36]. Since the web-based interface is implemented as a thin layer that access the main application implementation, a GUI based interface could be added in a similar fashion, accessing the underlying collections using the existing application implementation. Other interfaces could also be applied to the same core application implementation in order to meet the needs of specific user communities.

Finally, imagine that while this work was being conducted by one research group, another group has implemented an FIF collection to study the interaction of Cervantes and music. The web-based interface could then be extended to incorporate the results of

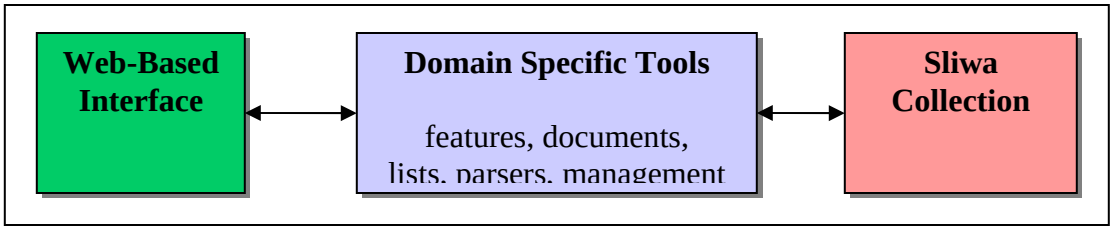

Figure 15: Block diagram of the major components of the Sliwa application

this new tool into the presentation layer of the original application. Thus, the work of multiple, independent development efforts can be quickly connected by extending the interface components used to display the collections. The structure of this hypothetical extension is diagrammed in Figure 16.

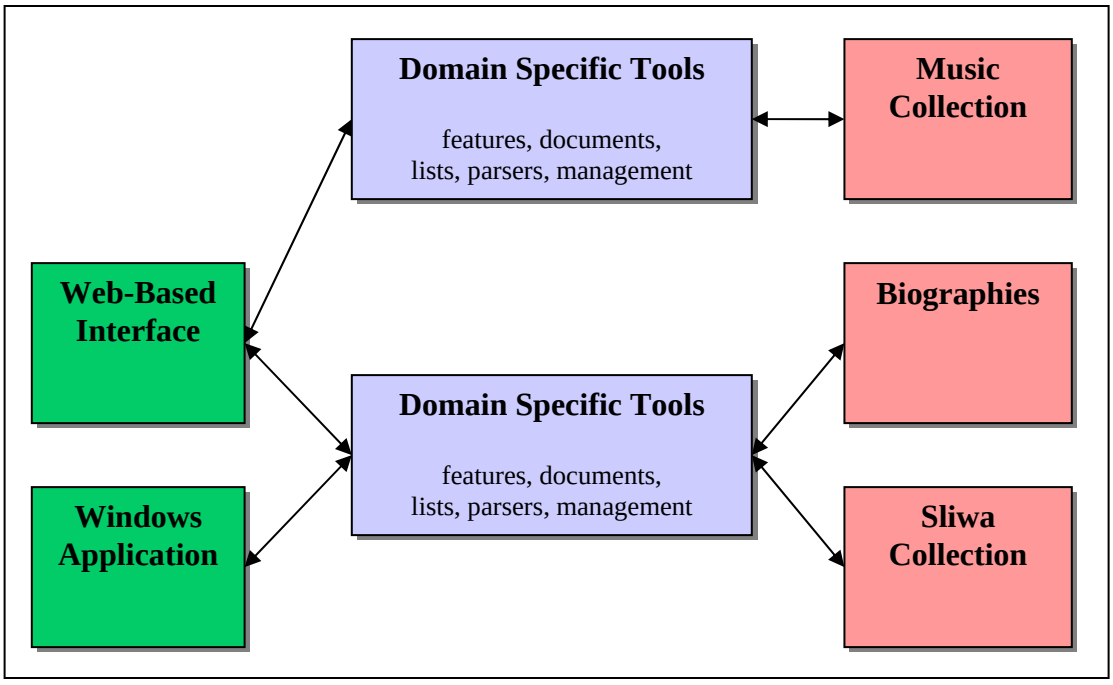

Figure 16: Block diagram of the extensions to the Sliwa application

FUTURE WORK

Since I have taken a horizontal approach to implementing a potentially very large and very complex system, my thesis opens a wide range for future work. I have grouped this work into four major categories: “Framework enhancements” deals with extensions of and refinements to the current framework. “Applications” deals with specific enhancements to the current Sliwa application as well as other potential applications of the framework to projects already in progress within the Texas A&M Engineering Extension Service (TEES) Center for the Study of Digital Libraries (CSDL). “Tools” deals will generic applications and extensions of the framework to provide key resources for use across applications. Finally, “bigger questions,” deals with some of the more open-ended research projects that applications based to some extent on this framework might be able to pursue.

Framework Enhancements

Document chunking and anchors: Currently, documents are treated as a single unit. Custom implementations of the `Document` class may chose to which portions of a document to pass to filters for feature identification and indexing filters may be built to analyze the document structure and adapt accordingly, but from the system’s view the internal structure of the document is opaque. It is well known, however, that this is insufficient. Applications will need to link to arbitrary sections of the document. Full-featured indexing will need to index sub-sections of the document as well as the whole. How a document may be sub-divided into chunks is heavily dependent upon both the type of document that needs to be “chunked” and the applications that the “chunking” is intended to support.

To support these needs, the FIF needs to be extended to support arbitrary document chunking. These chunking mechanisms should support the following objectives. First, chunking operations should be separate from the document representation itself. A single document may need to be chunked in different ways for different purposes. Therefore,

the chunking mechanism needs to be implemented in such a way that it can be applied to an appropriate document type without that document needing to have any knowledge of how it will be chunked. Chunking should be allowed at different levels of non-hierarchical granularity. For example, a book-sized document might be chunked into chapters, sections, and paragraphs. It could also be chunked into pages–which are likely to overlap paragraph and section boundaries. Third, the chunking mechanism needs be integrated into the indexing sub-system to allow indexers to take better advantage of the structure of the documents they are indexing. Fourth and finally, both the chunks and arbitrary document segments need to be persistently referencable. The searching and indexing tools must be able to reference the chunks they are indexing and automatic link generation tools will need to be able to reference arbitrary segments of the documents.

This represents a sizable amount of work and will have a wide-ranging structural impact on the design and implementation of the framework, but initial investigations indicate that it is a readily tractable task.

Second order feature matching: Features identification strategies that identify features directly from the underlying text of a document can be described as first order features–they make no use of the document structure, including previously identified features. Second order feature matching, then, describes feature matching algorithms that do make use of information about previously identified features or other elements of the document structure. Currently, only first order feature mapping is supported.

Allowing second order feature mapping would offer two primary benefits. First, feature identification algorithms could use information from previously identified features to identify features more reliably and describe the feature's details more fully. For example, one possible feature might specify that the phrase 'como yo' followed by a <person> segment indicates that the identified person is the creator of the document. Second, more complex features could be built from simpler ones. For example, the current approach to identifying numbers in the text utilizes a regular expression that is over 18,000 characters long. If second order feature matching were available, individual

number words could be recognized with a series of much simpler patterns (most of less than 50 characters). Sequences of these individual number words could then be identified in a second pass to identify composite numbers such as twenty one thousand.

Multiple features at a single location: Currently, only one, non-overlapping feature can be identified at any particular point in the text (though features could, potentially be nested). In reality, multiple features can appear at a single position for a variety of reasons. Multiple different features might match a single span of text, requiring subsequent disambiguation–either by automatic means or by human readers. A single span of text might legitimately match multiple features; a word might have a corresponding entry in a dictionary, an encyclopedia and be a key term that points to other documents in a collection. Each of these cases could result in features with identical or overlapping spans of text. Currently, given the strict hierarchical nature of the document model this is not supported. The current feature model supports a one to one mapping between spans of text and feature. This is a convenient oversimplification, that while sufficient for many purposes, mask the true complexity of feature identification. Support for one to many (requiring disambiguation) and many to one (requiring more complex displays and possibly non-hierarchical representations) mappings is needed.

Better “hooking” mechanisms: One major application of a feature identification system is to allow “hooks” between related portions of documents. Currently these hooks are made available to applications that implement the framework, primarily by using the identified key features to trigger custom developed searches. In the interest of promoting interoperability, a key challenge is developing tools that make these hooks available systems that are not based on the FIF. Similar work is currently being conducted as part of the National Science Digital Library [18].

Searching feature lists: Feature lists can be exceedingly large–the rather modest list of names in the Sliwa collection contains more than 1300 entries, too large to be easily navigated by via a browsing interfaces. Accordingly, a mechanism for searching is

needed. While this is relatively straightforward, a solution similar to that employed by the index filter or the web peer adapter is needed to handle the fact that the details of feature implementations are not known in advance.

Applications

Sliwa

Enhancements to a particular collection or interface are always possible. A few of the more interesting enhancements that could be added to the Sliwa collection are described below.

HMM-based feature identification algorithms: The algorithms currently used to identify names and places within documents are based on relatively simple pattern matching, based on an index of known names in the collection that provides some fuzzy matching capabilities to handle spelling irregularities. Current state of the art named-entity recognition algorithms are available and could likely achieve much higher rates of precision and recall. These algorithms could also support the identification of new names, not currently found in the indices. Hidden Markov models or semi-Markov models have yielded exceptional results and provide a well-studied place to start refining the implementation of the system.

Key-phrase identification: Key phrase identification provides a relatively simple approach with the potential to yield major enhancements. Key phrases targeted to identify documents of a certain type can be readily identified within the text. A feature implementation that would search for these phrases can be easily implemented and made available to via the web-based feature-editing tool. Subject area experts could then use this tool to construct lists of phrases that, where documents that match a phrase in a particular list are likely to be of the same general type. For example, given lists that represent military information, financial records, personal documents, and legal proceedings, a reader might search for all documents that mention Miguel de Cervantes

and Cristóbal de Alcántara and quickly see in which types of documents these two individuals appear together.

Better editorial tools: There is much room to explore the interaction of editors (either official or editorial feedback provided by readers) and the automatic routines supported by the collection. One possible question is how to provide editors with tools to identify people not currently recognized by the system? Another direction would be to examine what affordances could be provided for editing the collection, creating hand-crafted linkages between document, creating trails of documents.

Documented biographies: One key problem with biographies is that readers of the biography often have little access to the primary sources used to inform the biography. One potential application of this tool for the Sliwa collection is to use the identified features to provide hooks that allow readers to navigate between narrative biographies and the documents in the collection. This offers the twin advantages of allowing readers of a biography to "dig deeper" into the primary source material and situating the primary source material within the perspective of various biographers.

Other Applications

One of the major claims I have made is that this framework is suitable to support a wide range of collections and research agendas. A natural direction for future work is applying the FIF in other context to create dense inter-document linking and to explore other domains in which some of the general features of the document would be helpful. A few potential projects are listed below.

Picasso: The *Picasso Project*, in addition to containing 7000 images, also contains extensive textual descriptions of Picasso's life and painting. The FIF could be applied to this collection, again emphasizing the identification of key people and places within these narratives, but also providing support for terms of interest in an art-history context. One potential application is using the framework to identify segments that refer to a painting and then replace the "painting segment" with an appropriate representation

when the document is requested. That representation could then be tailored to the environment in which the document would be presented. For example, it could be displayed as a link to the painting, a representation of the image, or a description of the image in a popup window.

Expedition records: We have just begun to work with a collection of diaries and other records that document early expeditions into Texas by Spanish conquistadors [24]. In this collection, key tasks would include identifying geographical features, temporal sequences and events recorded in the records. Once identified, these features could be used to assist in determining the differences between multiple accounts of the same expedition.

More of the *Cervantes Project*: A natural application would be to use the FIF as a tool to provide inter-linkages between more of the collections within the *Cervantes Project*. This could provide a key integrative strategy for the various resources maintained by the project. Work is currently underway to use the framework to help integrate documents from our music collection with the broader corpus of the project. Another important project that could be undertaken in the near future would be integrating the biographies we have access to with the historical documents. This would require investigating how to identify topics and events, both within a single, large document and within the many, smaller documents contained in the Sliwa collection.

Tools

The broader context in which the framework is intended to be used includes many elements that for which general tools could be developed that would be applicable in a wide variety of settings. One such tool would provide support for “headword” documents. Documents such as dictionaries, encyclopedias, glossaries, indices, and thesauri are structured around groupings identified by a headword that is then described in more detail. These documents can serve as key resources for enhancing digital archives. A tool that could be configured to recognize documents of this format and automatically build feature lists based on the headwords in a document would have

applications in a wide range of projects. Since the sections in these documents are providing detailed information specifically about the headword, this tool would be able not only to develop valuable feature lists, but also to identify information specifically intended to describe the features that it finds.

A second key tool is an implementation of the document model based on the TEI encoding standards [49], along with tools to support a variety of metadata standards more directly. The bulk of this work could be implemented with the classes currently available simply by configuring the default segment factory. This configuration of the structure of the content of the document model could then be further enhanced by implementing and extension of the `Document` class to provide more support for the structural elements of the TEI directly via its API.

Bigger Questions

The future work listed above is largely implementation work. While the challenges and amount of work presented by these projects are significant and often complex, the solutions are relatively straightforward extensions of the current FIF and applications of it. They do, however, begin to touch on and offer directions to some broader research questions. In particular, work along these lines will help give shape to understanding how we can better identify the internal structure of documents and then use our improved knowledge of that structure to facilitate research and enhance collections. Another major line of research to pursue is the role of editors in enhancing collections. How can we improve the effectiveness and efficiency of corpus editors in digital library projects.

CONCLUSIONS

In this thesis, I have described an approach to implementing a feature identification system to support digital collections that provides a general framework for applications while allowing decisions about the details of document representation and features identification to be deferred to domain specific implementations of that framework. These deferred decisions include details of the semantics and syntax of markup, the types of metadata to be attached to documents, the types of features to be identified, the feature identification algorithms to be applied, and which features should be indexed. This approach provides strong support for the general aspects of developing a feature identification system allowing future work to focus on the details of applying that system to the specific needs of individual collections and user communities.

The framework I have presented is extensible at five main points. Most important is the feature identification. Custom implementations of the feature component will provide the primary mechanism for applications using the FIF to control both what features are identified in the documents and the algorithms to be employed in identifying them. This allows arbitrary custom implementations of the feature component, providing that these implementations adhere to a simple API. The second major extension point is the indexing system. This component determines which features are indexed for use in application and accordingly represents a major portion of the functionality of the system. Third, the OHCO based document content model allows generic feature identification algorithms to be provided across a open-ended range of underlying document types. The basic XML-centric segment component provided by the framework can be extended or replaced by implementations for alternative document formats such as RTF, PDF or Postscript. Implementations for non-textual documents (e.g. audio or images) can also be envisioned. Fourth, this content model is contained within a document component that can be extended to provide operations that more closely match the domain in which they will be used, to use custom data storage system or to tailor what portions of the document are presented to the feature identification and indexing portions of the system.

TEI conformant documents, for example, can be implemented so that only the body content (and not the header information) is passed to the feature identification system. The indexing sub-system can then be tailored to take advantage of these customized document components in order to leverage more detailed structural information. Fifth, the syntactical structure of the document content model can be explicitly represented and enforced by customized segment factories. This allows the syntactic (and by extension semantic) constrains of the document model to be enforced in a context where custom feature implementations may try to identify features that are not allowed by the content model. The FIF includes a basic implementation of a segment factory that is suitable for representing XML documents and provides a web based interface for specifying the structure of those documents.

In order to meet a tangible need within the *Cervantes Project* and to demonstrate the general applicability of the FIF, I have built a web-based interface for the Sliwa collection of historical documents pertaining to Cervantes and his family members, based on the FIF. This tool supports the identification of key features (person names, places, dates, monetary units and numbers), automatic hyperlink generation, and timeline visualizations based on these features. These documents form a critical resource for Cervantes scholars, providing access to primary source material to help them research topics including where he lived, his relationship with his father, sisters, and daughter, his life as a soldier, his legal problems, and how much Cervantes received for his works.

The framework presented here, along with its application to this collection, provides scholars with sophisticated tools that can use the "unstructured" information contained in these documents to support the visualization, navigation, and advanced searching strategies they need to effectively pursue answers to these questions. In this context, the feature identification system provides a key strategy for establishing connections between resources in the collection since the historical records do not neatly fit within the narrative and thematic structures of the *Quixote* [2]. Looking beyond the scope of the

Cervantes Project the strategy used here shows tremendous promise for facilitating the types of collection enhancement needed by humanities archives. Efforts at enhancing humanities collections can be broken into three major groups:

1. Huge collections such as the *Making of America* [50] [10], Gutenberg [41], and *Christian Classics Ethereal Library* [40] that have minimal tagging, annotation or commentary. These projects perform a crucial service by digitizing tremendous amounts of information.
2. Smaller projects in which editors carefully work with each page and line providing markup and metadata of extremely high quality and detail, mostly by hand. These efforts closely parallel traditional approaches to editorial work in a print environment. Projects in this group include the William Blake Archive [21], the Canterbury Tales project [43] the Rossetti Archive [34], and, currently, the *Cervantes Project*.
3. Middle ground projects which aim to develop collections with extensive tagging and markup, yet which are too large for hand editing to be practical. Such projects require new editorial roles that focus on customizing and skillfully applying automated techniques [15]. The *Perseus Project* [14] exemplifies this group.

My thesis work helps to bridge the gap between large, relatively unstructured collections and smaller, hand-edited collections, allowing editors to rapidly develop and employ customized tools to automatically enhance the collection while focusing the resources available for hand-editing on those elements of the collection that cannot be processed automatically or are important enough to warrant specific attention.

In implementing the system on which my thesis work is based, I have taken a horizontal approach–pursuing breadth across the system as a whole rather than focusing on the details of individual components. This approach offers three main benefits. First, it meets a tangible need within the *Cervantes Project* to make the documents in the Sliwa

collection available to the scholarly community in a form that readily supports the types of research they will need. Second, it provides a proof of concept that demonstrates the major components of this approach to identifying and using the internal structure of documents in a cultural archive. Third, it serves as a solid base that may be extended by future research efforts.

REFERENCES

[1] Apache Software Foundation (2005) Lucene Web Site http://lucene.apache.org/ [accessed 9 Sept 2005]

[2] Audenaert N, Furuta R, Urbina E, Deng J, Monroy C, Sáenz R, Careaga E (2005). Integrating collections at the Cervantes project. In: Proc. 5th ACM/IEEE-CS joint conference on digital libraries, Denver, CO, pp 287-288

[3] Audenaert N, Furuta R, Urbina E, Deng J, Monroy C, Sáenz R, Careaga E (2005) Integrating diverse research in a digital library focused on a single author. In: Proc. 9th European conference on research and advanced technology for digital libraries, Vienna, Austria, pp 151-161

[4] Bainbridge D, Thompson J, Witten IH (2003). Assembling and enriching digital library collections. In: Proc. 3th ACM/IEEE-CS joint conference on digital libraries, Houston, TX, pp 323-334

[5] Bikel DM, Schwartz R, Weischedel RM (1999) An algorithm that learns what's in a name. Machine Learning, 34(1-3):211-231

[6] Burnett I, Van de Walle R, Hill K, Bormans J, Pereira F (2003) MPEG-21: goals and achievements. IEEE Multimedia, 10(4):60-70

[7] Callan J, Mitamura T (2002) Knowledge-based extraction of named entities. In: Proc. 11th international conference on information and knowledge management, McLean, VA, pp 532-537

[8] Chinchor NA (1998) Overview of MUC-7/MET-2. In: Proc. 7th message understanding conference (MUC-7), Fairfax, VA. http://www.itl.nist.gov/iaui/894.02/related_projects/muc/proceedings/muc_7_toc.html.

[9] Cohen WW, Sarawagi S (2004) Exploiting dictionaries in named entity extraction: combining semi-Markov extraction processes and data integration methods, In: Proc. 2004 ACM SIGKDD international conference on knowledge discovery and data mining, Seattle, WA, pp 89-98

[10] Cornell University (2005) Making of America http://moa.cit.cornell.edu/moa/ accessed 8 Sept 2005]

[11] Crane G (1998) The Perseus Project and beyond. D-Lib Magazine

[12] Crane G (2000) Designing documents to enhance the performance of digital libraries: Time, space, people and a digital library of London. D-Lib Magazine 6(7/8)

[13] Crane, G (2004) Georeferencing in historical collections. D-Lib Magazine, 10(5).

[14] Crane G, ed. (2005) Perseus Project, Tufts University. http://www.perseus.tufts.edu/. [accessed 9 Sept 2005]

[15] Crane G, Rydberg-Cox JA (2000) New technology and new roles: the need for "corpus editors." In Proc. 5th ACM conference on digital libraries, San Antonio, TX, pp 252-253

[16] Crane G, Smith DA, Wulfman CE (2001) Building a hypertextual digital library in the humanities: a case study on London. In: Proc. first ACM/IEEE-CS joint conference on digital libraries, Roanoke, VA, pp 426-434

[17] Crane G, Wulfman C (2003) Towards a cultural heritage digital library. In: Proc. 3rd ACM/IEEE-CS joint conference on digital libraries, Houston, TX, pp 75-86

[18] Colati G, Crane G, Choudhury S, Szalay A, principal investigators (2005) The Services for a Customizable Authority Linking Environment (SCALE) project http://dca.tufts.edu/scale/. Accessed on Oct 19, 2005.

[19] Dekkers M, Baker T, directors (2005) The Dublic Core Metadata Initiative. http://dublincore.org/ [accessed 9 Sept 2005]

[20] DeRose SJ, Durand DG, Mylonas E, Renear AH (1990) What is text, really? J of Computing in Higher Education, 1(2):3-26

[21] Eaves M, Essick R, Viscomi J, eds. (2005) The William Blake Archive, The Institute for Advanced Technology in the Humanities. http://www.blakearchive.org/. Accessed on Sept 9, 2005.

[22] The J. Paul Getty Trust (2005) Getty Thesaurus of Geographic Names Online http://www.getty.edu/research/conducting_research/vocabularies/tgn/ [accessed 9 Sept 2005]

[23] Hatcher E, Gospodnetic O. (2004) Lucene in Action. Manning, Greenwich, CT

[24] Imhoff B, ed. (2002) The diary of Juan Dominguez de Mendoza's expedition into Texas (1683-1684): A critical edition of the Spanish text with facsimile reproductions. William P. Clements Center for Southwest Studies, Southern Methodist University, Dallas, TX

[25] Finneran RJ, ed. (1996) The Literary Text in the Digital Age. University of Michigan Press, Ann Arbor, MI

[26] Furuta R., Kalasapur S, Kochumman R, Urbina E, Vivancos-Pérez R (2001) The Cervantes Project: steps to a customizable and interlinked on-line electronic variorum edition supporting scholarship. In: Proc. 5th European conference on research and advanced technology for digital libraries, Darmstadt, Germany, pp 71-82

[27] Krupka GR, Hausman K (1998) Isoquest, Inc: Description of the NetOwl(TM) extractor system as used for MUC-7. In: Proc. 7th message understanding conference (MUC-7), Fairfax, VA http://www.itl.nist.gov/iaui/894.02/related_projects/muc/proceedings/muc_7_toc.html.

[28] Krupka GR (1995) Description of the SRA system as used for MUC-6. In: Proc 6th Message Understanding Conference (MUC-6), Columbia, MD, pp 221-235

[29] Kochumman R, Monroy C, Furuta R, Goenka A, Urbina E, Melgoza E (2002) Towards an electronic variorum edition of Cervantes' Don Quixote: visualizations that support preparation. In: Proc. 2nd ACM/IEEE-CS joint conference on digital libraries, Portland, OR, pp 199-200

[30] The Library of Congress (2005) MARC Standards. http://www.loc.gov/marc/ [accessed 9 Sept 2005]

[31] The Library of Congress (2005) Metadata Encoding and Transmission Standard http://www.loc.gov/standards/mets/. [accessed Sept 9, 2005]

[32] MPEG-21, information technology, multimedia framework (2003) Part 2: digital item declaration. ISO/IEC 21000-2:2003

[33] Mallen E, ed. (2005) The Picasso Project, Hispanic Studies Department, Texas A&M University. http://www.tamu.edu/mocl/picasso/ [accessed 7 Feb 2005]

[34] McGann J, ed. (2003) The Rossetti Archive, The Institute for Advanced Technologies in the Humanities, University of Virginia. http://www.rossettiarchive.org/ [accessed 7 Feb 2005]

[35] Mikheev A, Moens M, Grover C (1999) Named entity recognition without gazetteers, In Proc. of the 9th conference on European chapter of the association for computational linguistics, Bergen, Norway, pp 1-8

[36] Monroy C, Kochumman R, Furuta R, Urbina E, Melgoza E, Goenka A (2002) Visualization of variants in textual collations to analyze the evolution of literary works in the Cervantes Project. In: Proc. 6th European conference on research and advanced technology for digital libraries, London, UK, pp 638-653

[37] Pasça M (2004) Acquisition of categorized named entities for web search. In: Proc. Conference on Information and Knowledge Management, Washington DC, pp 137-145

[38] Pastor JJ (2005) Música y literatura: la senda retórica. Hacia una nueva consideración de la música en Cervantes. Doctoral dissertation. Universidad de Castilla-La Mancha.

[39] Paynter GW (2005) Developing practical automatic metadata assignment and evaluation tools for internet resources. In Proc. 5th ACM/IEEE-CS joint conference on digital libraries, Denver, CO, pp 291-300

[40] Plantinga H, coord. (2005) Christian Classics Ethereal Library, Calvin College, Grand Rapids, MI. http://www.ccel.org/ [accessed 8 September 2005]

[41] Project Gutenberg Literary Archive Foundation (2005) Project Gutenberg http://www.gutenberg.org/. [accessed 9 Sept 2005]

[42] Ranear A, Mylonas E, Durand D (1996) Refining our notion of what text really is: the problem of overlapping hierarchies. In: Research in Humanities Computing. Nancy Ide and Susan Hockey, eds. Oxford University Press, p. 265

[43] Robinson P, ed. (2002) The Canterbury Tales Project, De Montfort University, Leicester, England. http://www.cta.dmu.ac.uk/projects/ctp/. [accessed 25 May 2002]

[44] Shillingsburg PL (1996) Scholarly Editing in the Computer Age: Theory and Practice. University of Michigan Press, Ann Arbor

[45] Sliwa K (2000) Documentos Cervantinos: Nueva recopilación; lista e índices. Peter Lang, New York

[46] Stevenson M, Gaizauskas R. (2000) Using corpus-derived name lists for named entity recognition. In: Proc 6th conference on applied natural language processing. Seattle, WA, pp 290-295

[47] Sundheim B (1995) Overview of results of the MUC-6 evaluation. In: Proc. 6th message understanding conference (MUC-6), Columbia, MD, pp 13-31

[48] Speerberg-McQueen CM, Burnard L, eds. (1994) Guidelines for electronic text encoding and interchange. Text Encoding Initiative, Chicago and Oxford.

[49] The TEI Consortium http://www.tei-c.org/. Accessed on Sept 9, 2005.

[50] University of Michigan (2005) Making of America. http://www.hti.umich.edu/m/moagrp/ [accessed 8 Sept 2005]

[51] Urbina E, ed. (2005) The Cervantes Project, Center for the Study of Digital Libraries, Texas A&M University. http://csdl.tamu.edu/cervantes. [accessed 7 Feb 2005]

[52] Urbina E, Furuta R, Smith SE, Audenaert N, Deng J, Monroy C, (2006) Visual knowledge: textual iconography of the Quixote, a hypertextual archive. In: Literary and Linguistic Computing (forthcoming)

VITA

Name: ..... Michael Neal Audenaert

Address:..... Department of Computer Science, Mail Stop 3112, Texas A&M University, College Station, TX, 77840

Email Address: ..... neal@csdl.tamu.edu

Education:..... B.S. Computer Science, Texas A&M University, 2001

M.S. Computer Science, Texas A&M University, 2005