Integrated Information Management: An Interactive, Extensible Architecture for Information Retrieval

Eric Nyberg
Language Technologies Institute
Carnegie Mellon University
Pittsburgh, PA 15213
ehn@cs.cmu.edu

Hal Daume
Language Technologies Institute
Carnegie Mellon University
Pittsburgh, PA 15213
hcd@cs.cmu.edu

1. INTRODUCTION

Most current IR research is focused on specific technologies, such as filtering, classification, entity extraction, question answering, etc. There is relatively little research on merging multiple technologies into sophisticated applications, due in part to the high cost of integrating independently-developed text processing modules.

In this paper, we present the Integrated Information Management (IIM) architecture for component-based development of IR applications[1]. The IIM architecture is general enough to model different types of IR tasks, beyond indexing and retrieval. Rather than providing a single framework or toolkit, our goal is to create a higher-level framework which is used to build a variety of different class libraries or toolkits for different problems. Another goal is to promote the educational use of IR software, from an "exploratory programming" perspective. For this reason, it is also important to provide a graphical interface for effective task visualization and real-time control.

Prior architecture-related work has focused on toolkits or class libraries for specific types of IR or NLP problems. Examples include the SMART system for indexing and retrieval [17], the FIRE [18] and InfoGrid [15] class models for information retrieval applications, and the ATTICS [11] system for text categorization and machine learning. Some prior work has also focused on the user interface, notably FireWorks [9] and SketchTrieve [9][2]. Other systems such as GATE [4] and Corelli [20] have centered on specific approaches to NLP applications.

The Tipster II architecture working group summarized the requirements for an ideal IR architecture [6], which include:

- *Standardization*. Specify a standard set of functions and interfaces for information services.
- *Rapid Deployment*. Speed up the initial development of new applications.
- *Maintainability*. Use standardized modules to support plug-and-play updates.
- *Flexibility*. Enhance performance by allowing novel combinations of existing components.
- *Evaluation*. Isolate and test specific modules side-by-side in the same application.

One of the visions of the Tipster II team was a "marketplace of modules", supporting mix-and-match of components developed at different locations. The goals of rapid deployment and flexibility require an excellent user interface, with support for drag-and-drop task modeling, real-time task visualization and control, and uniform component instrumentation for cross-evaluation. The modules themselves should be small, downloadable files which run on a variety of hardware and software platforms. This vision is in fact a specialized form of component-based software engineering (CBSE) [14], where the re-use environment includes libraries of reusable IR components, and the integration process includes real-time configuration, control, and tuning.

Section 2 summarizes the architectural design of IIM. Section 3 provides more detail regarding the system's current implementation in Java. In Section 5 we describe three different task libraries that have been constructed using IIM's generic modules. Current instrumentation, measurement, and results are presented in Section 6. We conclude in Section 7 with some relevant comparisons of IIM to related prior work.

2. ARCHITECTURAL DESIGN

IIM uses a flow-based (pipe and filter [16]) processing model. Information processing steps are represented as nodes in a graph. Each edge in the graph represents a flow connection between a parent node and a child node; the documents produced by the parent node are passed to each child node. In IIM, the flow graph is referred to as a *node chain*. A sample node chain is shown in Figure 1. The IIM class model includes six basic node types, which can be used to model a variety of IR problems:

1. *Source*. Generates a document stream (from a static collection, web search, etc.) and passes documents one at a time to its child node(s).
2. *Filter*. Passes only documents which match the filter to its child node(s).
3. *Annotator*. Adds additional information to the document regarding a particular region in the document body.

[1]This work is supported by National Science Foundation (KDI) grant number 9873009.
[2]For further discussion on how these systems compare with the present work, see Section 7.

.

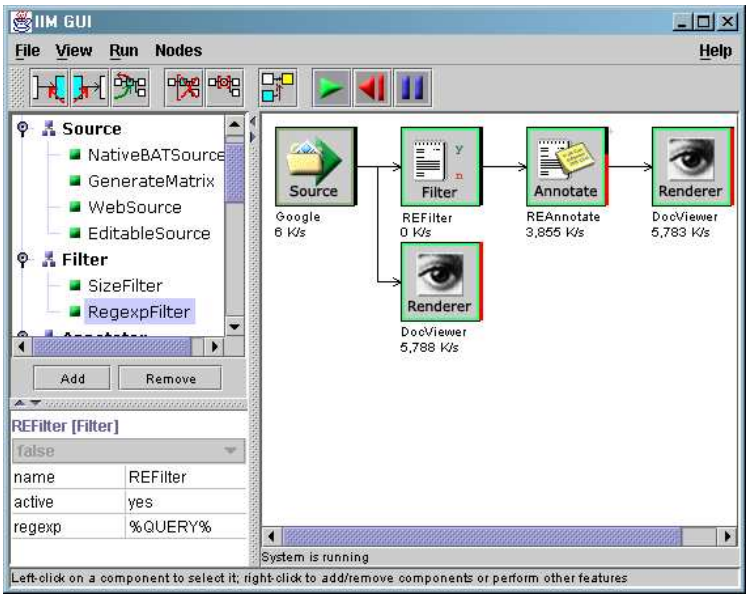

Figure 1: IIM User Interface

4. *Sink*. Creates and passes either a single document or a collection to its child node(s), after pooling the input documents it receives.

5. *Transformer*. Creates and passes on a single new document, presumably the result of processing its input document.

6. *Renderer*. Produces output for documents received (to disk, to screen, etc.).

The IIM class model is embedded in a Model-View-Controller architecture [5], which allows the system to be run with or without the graphical interface. Pre-stored node chains can be executed directly from the shell, or as a background process, completely bypassing all user interaction when optimal performance is required. The Controller subsystem and interface event dispatching subsystem must run as separate threads to support dynamic update of parameters in a running system. The View (user interface) should support: a) plug-and-play creation of new node chains; b) support for saving, loading and importing new node chains; c) dynamic visualization of a task's status; and d) direct manipulation of a node's parameters at any time.

In addition to the nodes themselves, IIM supports two other important abstractions for IR task flows:

- *Macro Nodes*. Certain sequences of nodes are useful in more than one application, so it is convenient to store them together as a single reusable unit, or *macro node*. IIM allows the user to export a portion of a node chain as a macro node to be loaded into the Node Library and inserted into a new chain as a single node. The user may specify which of the properties of the original nodes are visible in the exported macro node (see Figure 3).

- *Controllers*. Some IR tasks require iteration through multiple runs; the system's behavior on each successive trial is modified based on feedback from a previous run. For example, a system might wish to ask for more documents or perform query expansion if the original query returns an insufficient number of relevant documents. IIM includes a *Controller* interface, which specifies methods for sending feedback from one node to another. The user can implement a variety of controllers, depending on the needs of the particular application.

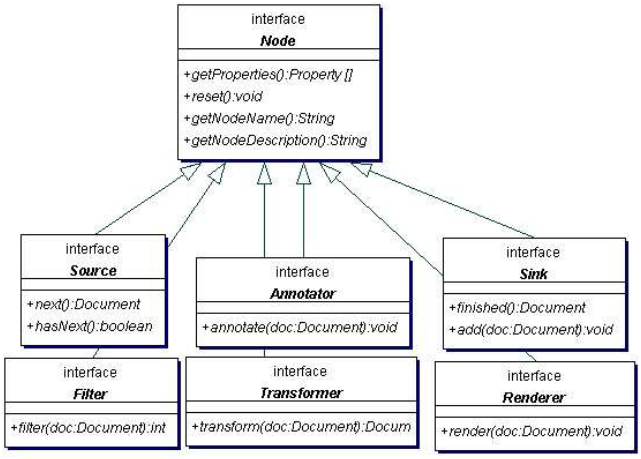

Figure 2: Node Interface and Subtypes.

3. JAVA IMPLEMENTATION

In the IIM Java implementation, nodes are specified by the abstract interface Node and its six abstract subinterfaces: *Source*, *Filter*, *Annotator*, *Transformer*, *Sink* and *Renderer* (see Figure 2). Any user-defined Java class which implements one of the *Node* subinterfaces can be loaded into IIM and used in a node chain. The visualization of a node is represented by a separate Java class, Box, which handles all of the details related to drawing the node and various visual cues in the node chain display.

The graphical user interface (Figure 1) is implemented as a set of Java Swing components:

- *Node Chain Display*. The canvas to the right displays the current node chain, as described in the previous section. While

Figure 3: Exporting A Macro Node.

the node chain is running, IIM provides two types of visual feedback regarding task progress. To indicate the percentage of overall run-time that the node is active, the border color of each node varies from bright green (low) to bright red (high). To indicate the amount of output per node per unit of time spent (throughput), the system indicates bytes per second as a text label under each node. A rectangular meter at the right of each node provides a graphic visualization of relative throughput; the node with the highest throughput will have a solid red meter, while other nodes will have a meter level which shows their throughput as a percentage of maximum throughput.

- *Node Library*. The tree view to the upper left displays the library of nodes currently available on the user's machine for building and extending node chains. New nodes or node directories can be downloaded from the web and added while the system is running. The component loader examines each loaded class using Java's reflection capabilities, and places it in the appropriate place(s) in the component tree according to which of the *Node* subinterfaces it implements.

- *Node Property Editor*. The Property Editor (table view) to the lower left in Figure 1 displays the properties of a selected node, which the user can update by clicking on it and entering a new value.

- *Node Chain Editor*. IIM supports dynamic, interactive manipulation of node chains. The left side of the toolbar at the top of the IIM Window contains a set of chain editing buttons. These allow the user to create, modify and tune new node chains built from pre-existing components.

- *Transport Bar*. IIM uses a tape transport metaphor to model the operation of the node chain on a given data source. The "Play", "Pause" and "Rewind" buttons in the toolbar (right side) allow the user to pause the system in mid-task to adjust component parameters, or to start a task over after the node chain has been modified.

The run-time Controller subsystem is implemented as a Java class called *ChainRunner*, which can be invoked with or without a graphical interface component. *ChainRunner* is implemented as a *Thread* object separate from the Java Swing event dispatching thread, so that user actions can be processed concurrently with the ongoing operation of a node chain on a particular task.

4. IIM COMPONENTS

The current IIM system includes a variety of nodes which implement the different IIM component interfaces. These nodes are described in this section.

4.1 Source Nodes

- *EditableSource*. Prompts the user to interactively enter sample documents (used primarily for testing, or entering queries).

- *WebSource*. Generic support for access to web search engines (e.g., Google). Includes multithreading support for simultaneous retrieval of multiple result documents.

- *NativeBATSource*. Generic support for access to document collections stored on local disk. Implemented in C, with a Java wrapper that utilized the Java Native Interface (JNI).

4.2 Filter Nodes

- *SizeFilter*. Only passes documents which are above a user-defined size threshold.

- *RegexpFilter*. Only passes documents which match a user-defined regular expression; incorporates the GNU regexp package.

4.3 Annotator Nodes

- *NameAnnotator*. Locates named entities (currently, person names) in the body of the document, and adds appropriate annotations to the document.

- *IVEAnnotator*. For each named entity (person) annotation, checks a networked database for supplemental information about that individual. An interface to a database of information about individuals, publications, and organizations, created as part of the Information Validation and Evaluation project at CMU [12]. Implemented using Java Database Connectivity (JDBC).

- *BrillAnnotator*. Accepts a user-defined annotation (e.g., PASSAGE) and adds a new annotation created by calling the Brill Tagger [1] on the associated text. Implemented via a TCP/IP socket protocol which accesses a remote instance of the tagger running as a network service.

- *ChartAnnotator*. Accepts a user-defined annotation, and adds new annotations based on the results of bottom-up chart parsing with a user-defined grammar. The user can select which linguistic categories (e.g., NP VP, etc.) are to be annotated.

- *RegexpAnnotator*. Annotates passages which match a user-defined regular expression.

4.4 Transformer Nodes

- *BrillTransformer*. Similar to the *BrillAnnotator* (see above), but operates directly on the document body (does not create separate annotations).

- *Inquery*. Accepts a query (represented as an input document) and retrieves a set of documents from the Inquery search engine [2]. Accesses an Inquery server running as a networked service, using TCP/IP sockets.

- *WordNet*. Accepts a document, and annotates each word with a hypernym retrieved from WordNet [19]. Accesses a WordNet server running as a networked service, using TCP/IP sockets.

4.5 Sink Nodes

- *Ranker*. Collects documents and sorts them according to a user-defined comparator. The current implementation supports sorting by document size or by annotation count.

- *CooccuranceSink*. Builds a matrix of named entity associations within a given text window; uses NAME annotations created by the *NameAnnotator* (see above). The output of this node is a special subclass of *Document*, called *MatrixDocument*, which stores the association matrix created from the document collection.

- *QAnswer*. Collects a variety of annotations from documents relevant to a particular query (e.g., "What is Jupiter?"), and uses them to synthesize an answer.

4.6 Renderer Nodes

- *StreamRenderer*. Outputs any documents it receives to a user-specified file stream (or to standard output, by default).

- *DocumentViewer*. Pops up a document display window, which allows the user to browse documents as they are accepted by this node.

- *MatrixRenderer*. A two-dimensional visualization of the association matrix created by the *CoocurrenceSink* (see above). Accepts instances of *MatrixDocument*.

5. IIM APPLICATIONS

The initial set of component nodes has been used as the basis for three experimental applications:

- *Filtering and Annotation*. An interactive node chain that allows the user to annotate and collect documents matching any regular expression; the resulting collection can then be viewed interactively (with highlighted annotations) in a pop-up viewer window.

- *Named Entity Association*. A node chain which performs named-entity annotation using a phi-square measure[3], producin a *MatrixDocument* object (a user-defined *Document* subclass, which represents the association matrix). Note that the addition of a specialized *Document* subclass does not require recompilation of IIM (although the user must take care that specialized document objects are properly handled by user-defined nodes).

- *Question Answering*. A node chain which answers "What is" questions by querying the web for relevant documents, finding relevant passages [8, 10], and synthesizing answers from the results of various regular expression matches[3].

6. PERFORMANCE

In order to support accurate side-by-side evaluation of different modules, IIM implements two kinds of instrumentation for run-time performance data:

- *Per-Node Run Time*. The *ChainRunner* and *Box* classes automatically maintain run-time statistics for every node in a chain (including user-defined nodes). These statistics are printed at the end of every run.

- *Node-Specific Statistics*. For user-defined nodes, it may be useful to report task-specific statistics (e.g., for an *Annotator*, the total number of annotations, the average annotation size, etc.). IIM provides a class called *Options*, which contains a set of optional interfaces that can be implemented to customize a node's behavior. Any node that wishes to report task-specific statistical data can implement the *ReportsStatistics* interface, which is called by the *ChainRunner* when the chain finishes.

An example of the statistical data produced by the system is shown in Figure 4. The system is careful to keep track of time spent "inside" the nodes, as well as the overall clock time taken for the task. This allows the user to determine how much overhead is added by the IIM system itself.

The throughput speed of the prototype system is acceptably fast, averaging better than 50M of text per minute on a sample filtering task (530M of web documents), running on a typical Pentium III PC with 128M RAM. IIM requires about 10M of memory (including the Java run-time environment) for the core system and user interface, with additional memory requirements depending on the size of the document stream and the sophistication of the node chain[4]. Although the core system is implemented in Java, we have also implemented nodes in C++, using appropriate wrapper classes and the Java Native Interface (JNI). This technique allows us to implement critical, resource-intensive nodes using native code, without sacrificing the benefits of the Java-based core system.

7. DISCUSSION

The preliminary results of the IIM prototype are promising. IIM's drag-and-drop component library makes it possible to build and tune a new application in a matter of minutes, greatly reducing the amount of effort required to integrate and reuse existing modules.

[3]We are currently expanding this application to include part of speech tagging and syntactic parsing, both of which are straightforwardly modeled as examples of the *Annotator* interface.

[4]Node chains which create a high volume of annotations per document use more memory, as do node chains which create new collections, transform documents, etc.

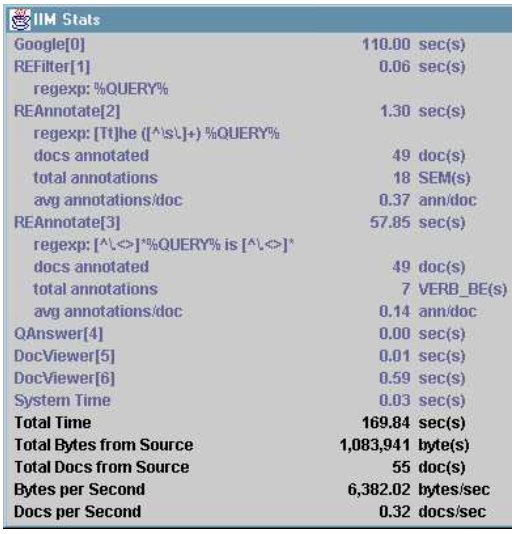

Figure 4: Statistics for a Node Chain.

In the future, we hope this high degree of flexibility will encourage greater experimentation and the creation of new aggregate systems from novel combinations of components, leading to a true "marketplace of modules".

Building extensible architectures as "class library plus application framework" is not a new idea, and has been discussed before with respect to information retrieval systems [7, 18, 9]. One might claim that any new IR architecture should adopt a similar design pattern, given the proven benefits of separating the modules from the application framework (flexibility, extensibility, high degree of reuse, easy integration, etc.). To some extent, IIM consolidates, refines and/or reimplements ideas previously published in the literature. Specifically, the following characteristics of the IIM architecture can be directly compared with prior work:

- The IIM classes *Renderer*, *Document*, *MultiDocument*, and annotations on *Document* can be considered alternative implementations of the InfoGrid classes *Visualizer*, *Document*, *DocumentSet* and *DocumentPart* [15]. However, in IIM annotations are "lightweight", meaning that they do not require the instantiation of a separate user object, but can be modeled as simple String instances in Java when a high degree of annotation requires optimal space efficiency.
- The use of color to indicate status of a node is also used in the SketchTrieve system [18].
- IIM's visualization of the document flow as a "node chain" can be compared to the "wire and dock" approach used in other IR interfaces [9, 4, 13].
- The use of a Property Editor to customize component behavior is an alternative approach to the IrDialogs provided by the FireWorks toolkit [9] for display and update of a component's state.

Nevertheless, IIM is at once simpler and more general than systems such as InfoGrid [15] and FIRE [18]. One could claim that IIM supports a higher degree of *informality* [9] than FIRE, since it enforces no type-checking on node connectivity. Since all tasks are modeled abstractly as document flows, nodes need only implement one of the *Node* sub-interfaces, and each node chain must begin with a *Source*. Another point of comparison is the task-specific detail present in the FIRE class hierarchy. In IIM, task-specific objects are left up to the developer (for example, representing particulars of access control on information sources, or details of indexing and retrieval, such as *Index*, *Query*, etc.).

Hendry and Harper [9] have used the *degree of user control* as a dimension of comparison for IR architectures. At one extreme are systems which allow dynamic view and access to the run-time state of components, while at the other lie systems which hide implementation detail and perform some functions automatically, for improved performance. In their comparison of SketchTrieve and InfoGrid, Hendry and Harper note that "a software architecture should provide abstractions for implementing both these". In IIM, the use of macro nodes can hide component details from the end user, especially when the component's parameter values have been tuned in advance for optimal performance.

8. ONGOING RESEARCH

While the initial results reported here show promise, we are still evaluating the usability of IIM in terms of trainability (how fast does a novice learn the system), reusability (how easily a novice can build new applications from existing node libraries) and ease of integration (effort required to integrate external components and systems). The current version of IIM lacks the explicit document management component found in systems like GATE [4] and Corelli [20]; we are in the process of adding this functionality for the official release of IIM.

The IIM system (source code, class documentation, and node libraries) will be made available via the web as one of our final project milestones later in 2001. Anyone interested in using the system or participating in ongoing research and development is invited to visit the IIM web site and join the IIM mailing list:

$$http://hakata.mt.cs.cmu.edu/IIM$$

9. ACKNOWLEDGEMENTS

The authors would like to thank Jamie Callan for his guidance on the architecture design, and Krzysztof Czuba for providing networked instances of the Brill Tagger, Inquery, and WordNet.

10. REFERENCES

[1] Brill, Eric (1992). "A simple rule-based part of speech tagger", *Proceedings of the Third Conference on Applied Natural Language Processing*.

[2] Callan, J. P., W. B. Croft, and S. M. Harding (1992). "The INQUERY Retrieval System", *Proceedings of the 3rd International Conference on Database and Expert Systems*.

[3] Conrad, J., and M. H. Utt (1994). "A System for Discovering Relationships by Feature Extraction from Text Databases", *SIGIR '94*.

[4] Gaizauskas, R. Cunningham, H. Wilks, Y. Rodgers, P. and Humphreys, K. GATE – an environment to support research and development in natural language engineering. Proceedings of the 8th IEEE International Conference on Tools with Artificial Intelligence (ICTAI96) , Toulouse, France, pp 58-66, 1996.

[5] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

[6] Grishman, R. (1996). "Building an Architecture: A CAWG Saga", in *Advances in Text Processing: Tipster Program Phase II*, sponsored by DARPA ITC.

[7] Harper, D.J. and A.D.M. Walker (1992). “ECLAIR: An extensible Class Library for Information Retrieval”, *Computer Journal*, 35(3):256–267.

[8] Hearst, M. “Automatic acquisition of hyponyms from large text corpora.” *COLING ’92*.

[9] Hendry, D. G., and Harper, D. J. (1996). “An architecture for implementing extensible information-seeking environments”, *SIGIR ’96*.

[10] Joho, H. and M. Sanderson, “Retrieving descriptive phrases from large amounts of free text”, *CIKM 2000*.

[11] Lewis, D., D. Stern and A. Singhal (1999). “ATTICS: A Software Platform for Online Text Classification”, *SIGIR ’99*.

[12] Mitamura, T. (2001). “Language Resources for Determining Authority”, unpublished manuscript.

[13] Neuendorffer, T. (2000). “Analyst’s Workbench: A CAD-like GUI for Textual Search and Filter Creation”, HCII Seminar Series, Carnegie Mellon University, November 29.

[14] Pressman, R. (2000). *Software Engineering: A Practitioner’s Approach*, 5th edition, McGraw-Hill.

[15] Rao, R., S.K. Card, H.D. Jellinek, J.D. MacKinlay and G. Robertson: The Information Grid: A Framework for Information Retrieval and Retrieval–Centred Applications. *UIST ’92*.

[16] Shaw, M. and D. Garlan (1996). *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall.

[17] Salton, G. (1971). *The SMART Retrieval System - Experiments in Automatic Document Processing*, Prentice-Hall.

[18] Sonnenberger, G. and H. Frei (1995). “Design of a reusable IR framework”, *SIGIR ’95*.

[19] Fellbaum, C. (ed) (1998). WordNet: An electronic lexical database. Cambridge, MA: MIT Press.

[20] Zajac, R. (1997). “An Open Distributed Architecture for Reuse and Integration of Heterogenous NLP Components”, In *Proceedings of the 5th conference on Applied Natural Language Processing (ANLP-97)*.