ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

Ian Ruthven (Ed)

Miro'95

Proceedings of the Final Workshop on Multimedia Information Retrieval
(Miro '95)
Glasgow, Scotland
18-20 September 1995

Paper:

Use and Reuse of Indexing and Retrieval Functionality in a Multimedia IR Framework

Gabriele Sonnenberger, Tore A. Bratvold and Hans-Peter Frei

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

Use and Reuse of Indexing and Retrieval Functionality in a Multimedia IR Framework

Gabriele Sonnenberger, Tore A. Bratvold, and Hans-Peter Frei

Union Bank of Switzerland, UBILAB
cBahnhofstrasse 45, CH-8021 Zurich, Switzerland
{sonnenberger, bratvold, frei}@ubilab.ubs.ch

Abstract

We are working towards a reusable IR framework designed to facilitate the development of multimedia IR applications and to support the experimental evaluation of indexing and retrieval techniques. We recently have been reviewing and refining the design and implementation of the framework in order to improve its ease of use and reusability. Our particular goals have been that

- the framework should make transparent which IR functionality is available for information units of a certain type;
- particular functionality should be selectable as default either for a whole application or an individual IR database and/or user;
- users of the framework should be able to invoke specific functionality at run-time.

In this paper, we discuss in detail the (re-)use of IR functionality in our framework. First, we sketch how indexing and retrieval are organized in the framework; subsequently, we focus on the above issues.

1 Introduction

Information retrieval techniques for non-textual media such as speech [3,5] or graphs [1] are at an early stage of development and mostly less established than techniques for text. Consequently, there is a special need for evaluating and comparing the various approaches by experimental applications. Also, broadening the scope of IR and aiming to cover several media makes the development of systems more complicated and laborious.

The goal of the work described in this paper is to ease the development of multimedia IR applications and to support the experimental evaluation of indexing and retrieval techniques by providing reusable software in the form of an IR framework. An overview of FIRE, as we call our framework, is given in [7], where also the most important design criteria for developing reusable software in the IR domain are discussed. Recently, we have been reviewing and refining the design and implementation of FIRE in order to improve its ease of use and reusability. Our particular goals have been that

- the framework should make transparent which IR functionality, e.g. indexing techniques or matching methods, is available for information units of a certain type;
- particular functionality should be selectable as default either for a whole application or an individual IR database and/or user;
-

users of the framework should be able to invoke specific functionality at run-time and apply it to individual objects.

The focus of this paper is on the use and reuse of indexing and retrieval functionality provided by FIRE with special emphasis on the above issues. The paper is organized as follows: Section 2 describes briefly the major objectives and characteristics of FIRE as an IR framework. Subsequently, we sketch how indexing and retrieval are organized in FIRE. Sections 4 to 6 are the main parts of the paper. First, we outline how the framework offers information about available functionality and its usage. Next, we describe how a particular functionality can be selected as default. Then, we show how specific IR functionality can be chosen at run-time and applied to individual objects. The paper concludes with some implementation details and remarks on future work.

2 Objectives and Characteristics of FIRE

FIRE is designed to facilitate the development of IR applications and to support the experimental evaluation of indexing and retrieval techniques. In addition, FIRE is supposed to provide basic functionality for several media and should end up as a flexible and extensible framework that can be used for developing a wide range of IR applications. FIRE is an acronym for ‘Framework for Information Retrieval Applications’. It is developed in cooperation with the IR group at ‘The Robert Gordon University’, U.K.

A major problem in designing and implementing an IR framework is that we do not know in advance what kind of information units shall be managed in a specific application; nor do we know how they are going to be represented! Further, a variety of IR models and methods for indexing and retrieving information have been developed so far and, with the increasing importance of multimedia information, even more are to be expected. A useful IR framework should provide the option of applying different IR models and methods for indexing and retrieval.

For being able to use the functionality provided by a framework, transparency is an essential prerequisite. Application developers as well as end-users should be able to know which functionality is available and how to make use of it. Such information should be provided dynamically, since the framework’s functionality may change over time. FIRE’s approach for providing information about available functionality is to allow the user to browse through the class hierarchy where classes and objects as well as important methods are associated with meta-information.

A framework with rich functionality is of limited value if it is complex to use, or if its use requires understanding of many internal details. A framework should be easy to use especially if standard functionality is sufficient for solving a particular task. In order to achieve this, FIRE comes with default settings for operations for which several alternative methods are possible. These defaults make it possible to develop an application with little programming effort. The application developer as well as the end-user may modify the default settings in order to meet specific requirements. This can be done at run-time without any code manipulation.

Frameworks are developed to be reused in many different applications. In contrast to class libraries, the goal of a framework is not only to provide reusable classes and methods but also to design the overall structure of related applications. Thus, we have to design carefully the basic components of an IR application. These are especially documents, data elements from which these documents are composed, and indexes. A further important task is to organize indexing and retrieval in such a way that different models and methods can be applied by using the same interface. FIRE supports this by introducing indexing and retrieval modalities. These modalities allow the user to specify in a declarative way at run-time how indexing and retrieval have to be performed, e.g. indexing method to be used or retrieval model to be applied.

3 Indexing and Retrieval in FIRE

In FIRE, indexing and retrieval are performed in essentially the same way as in most IR systems. The main difference is that in FIRE these processes are more flexible by introducing indexing and retrieval modalities. In the following, we give a brief overview how indexing and retrieval are performed in FIRE and show how the modalities control these processes.

3.1 Indexing

FIRE represents documents by a set of features (or attributes). Note that we use the term document in a broad sense: documents may consist of structured and unstructured parts and may be composed of different media. Formally, the class *ReprInfoUnit* is responsible for modeling documents. Concrete subclasses of *ReprInfoUnit* define how documents of a certain type are represented in a particular application. When dealing with text documents for instance, there may be features like *Title*, *Authors*, *TextBody* and *PublicationDate*. Such features are selected from a collection of data types represented by the class *InfoObjectElement*. The framework defines common data types, like string, text, image, person name, bibliographic reference, and date. If needed, the application developer may extend this set of data types by adding new subclasses to *InfoObjectElement*.

Indexing a document involves three subprocesses:

- control of the indexing process,
- derivation of indexing features, and
- integration of indexing features into an index.

Figure 1 presents a functional model of the basic indexing process. (For specifying functional models and describing classes and objects, we use the notation developed by Rumbaugh et al. [6].) An additional task to be

performed is updating the index, e.g. sorting the entries and calculating indexing weights. This is usually done after indexing a whole set of documents and therefore not part of Figure 1.

Figure 1: Functional model of the indexing process. Note that steps 3 to 6 are repeated for each document feature to be indexed.

The subprocess for controlling the indexing of a document receives as input the document to be indexed (an instance of *ReprInfoUnit* or one of its subclasses). In addition, this process has access to the indexing modalities associated with the document. Indexing modalities specify how the indexing is to be performed, e.g. which features of the document are to be indexed and to which index(es) the results are to be passed. Indexing modalities are discussed in more detail in Section 6.

Indexing features are specified separately for each document feature to be indexed. The control process derives from the indexing modalities the method to be applied, invokes the method and passes the parameters to be used. In addition, it specifies the basic values of the source of the indexing features, i.e. the name of the document and the name of the document feature. The result, a set of indexing features, is passed back to the control process rather than passing it directly to the target index(es). This organization form has the advantage that indexing features can be derived without necessarily storing them in an index (this option is used for instance when preparing a query document for retrieving information). The control process sends the indexing features to the index(es) specified in the indexing modalities. The receiving index integrates the features into the index either by creating new index entries or by adding postings to existing entries.

As outlined above, the derivation of indexing features is controlled by indexing modalities. The integration of indexing features into an index, however, is not determined by indexing modalities, since this task must be performed uniformly for all indexing features of an index. Also, updating an index is fully controlled by the index itself.

3.2 Retrieval

The retrieval process is also divided into several subprocesses. These are:

- control of the retrieval process,
- derivation of indexing features from the query document,
- evaluation of single query conditions, and
- combination of partial retrieval results.

Figure 2 gives a functional model of the retrieval process to illustrate the interaction between the subprocesses.

Figure 2: Functional model of the retrieval process

FIRE favors retrieval interfaces where the user creates a document and fills it in partially to specify the information need, c.f. [7]. The retrieval process is initiated by passing the query document to the control process. Thereafter the query document is indexed. This is done in essentially the same way as for documents to be stored in an IR database. The result of this step is a set of indexing features, which form part of the query conditions. A query condition consists of an indexing feature and specifies how the condition is to be evaluated, i.e. which indexes are to be looked-up, which matching method is to be used, etc. These details are specified by the retrieval modalities, which are associated with the query document. The result of evaluating a query condition is a structure consisting of the condition and a set of features fulfilling the condition. Additionally, a retrieval weight is attached to the indexing features, which is a combination of a matching degree and an indexing weight. Figure 3 shows the evaluation of a single query condition in more detail.

Figure 3: Evaluation of a query condition with a key-based index look-up

The last step of the retrieval process is the combination of the partial retrieval results achieved by evaluating single conditions. The subprocess responsible for this task receives a(n internal) query which is either an unstructured or a structured query. Figure 4 shows the modeling of a query. An unstructured query consists of a set of basic retrieval results for single conditions. A structured query is composed of an operator, e.g. an *and* or a *not* operator, and one or more arguments. Such an argument may be a query itself (structured or unstructured) or a basic retrieval result. Each query is associated with the name of a method for calculating RSVs. This way, different retrieval models, e.g. the probabilistic or the vector space model, can be realized quite conveniently. The user interface of the current FIRE implementation does not yet support the formulation of structured queries. However, the user will be able to express relations by defining span-to-span links between (elements of) features of the query document in future versions of FIRE.

Figure 4: Modeling of a query

The retrieval process depends on retrieval modalities. They control:

- which features of a query document are evaluated,
- how the query document is indexed,
- which index is looked-up,
- which method is applied for matching an indexing feature of a query with the indexing features managed by an index of an IR database,

- how retrieval weights are calculated from matching degrees and indexing weights, and
- how partial retrieval results are combined and RSVs are computed.

Retrieval modalities are discussed in more detail in Section 6.

4 Getting Informed about Available Functionality and its Usage

A first prototype of FIRE has been implemented providing basic IR functionality for German and English text as well as English speech. Future versions of FIRE will support additional media. FIRE's IR functionality includes methods for

- deriving indexing features from information units,
- determining the similarity between two items (approximate match),
- computing relevance weights from indexing weights and matching degrees, and
- computing RSVs from partial retrieval results.

In many cases, more than one method may be suitable to solve a particular task. Consider for instance the matching of two strings. In order to determine their similarity, we can compute the phonetic similarity between them, determine the number of common n-grams, apply word formation rules to check whether they are derived from the same stem, etc. Operations which can be performed in different ways are not 'hard-coded' in the system, rather the application developer as well as the end-user can select the method most appropriate for the given task.

Important for a successful usage of the functionality at hand is that the system makes transparent

1. which methods are available in each case, and
2. how a given method is used, i.e. which parameters have to be set, what is the type of the parameters, etc.

In the initial design of FIRE, c.f. [7], methods were attached to the classes on whose instances an operation is performed. Methods for deriving indexing features for example were defined for the concrete subclasses of *InfoObjectElement*. Operations which may be performed in different ways are now modeled as classes. Figure 5 shows as example the class *Indexer* which is responsible for deriving indexing features from information units. This organization form has several advantages and is a design pattern that has proven to be successful, c.f. [2].

Figure 5: Class *Indexer* and a few subclasses

The abstract class *Indexer* defines two attributes and an abstract method. The attributes serve to specify the type of information units a concrete *Indexer* subclass is designed for (*TypeInformationObjectElement*) and the type of the indexing features derived by the subclass (*TypeIndexingFeature*). The method *deriveIndexingFeatures* provides a uniform interface for deriving indexing features, since its two parameters are needed by all *Indexer* subclasses. In case a specific *Indexer* requires additional parameters, e.g. for specifying a stopword list, these parameters are defined as attributes of the *Indexer* subclass.

ET++ [8-9], on which the implementation of FIRE is based, includes basic support for metainformation. Metainformation is provided semi-automatically and requires little effort of the programmer (call of a macro). This is important, since otherwise there is a high risk that metainformation does not reflect the actual modeling. ET++

metainformation allows one to determine the class name of an object/class and the name of its superclass. Furthermore, information is available about the names, types and values of the attributes defined for a class. ET++, however, does not provide adequate support for metainformation about operations and methods. By modeling operations as classes, metainformation about operations and methods is conveniently available. We can browse through FIRE's class hierarchy to determine for example which *Indexers* are defined for a particular type of information units. We also know the type of the indexing features a particular *Indexer* is deriving and which additional parameters are defined. Since parameters are modeled as attributes of classes, we can access their names, types and values. Such information is needed for a proper usage of the functionality provided by a system. In FIRE, metainformation is also used for consistency checks, e.g. to test whether an *Indexer* selected is in fact applicable to a given information unit. A further advantage is that functionality, e.g. a new *Indexer*, can be added by the application developer without modifying the framework!

5 Setting Defaults

The acceptance of a framework depends highly on the ease of its use. A rich functionality is attractive, however it may burden the application developer, if too many decisions have to be made before running an application. To avoid such problems, a framework should define defaults for operations where several possibilities exist. In addition, it should be possible to

- define specific defaults for a particular IR database and/or user,
- exchange default settings among IR databases or users, and
- perform consistency checks, e.g. in order to avoid that a method is set as default that is not supported by an application.

In FIRE, defaults are represented explicitly and the implementation details for managing defaults are mostly hidden from the user. For each operation which can be performed in different ways, e.g. approximate matching or derivation of indexing features, a class for specifying defaults is defined. Instances of these classes specify which methods and parameters are to be used as default for performing an operation on an object of a certain type. Figure 6 shows as example the class *DefaultIndexer* and a concrete instance of this class, called *(DefaultIndexer)*. The instance object specifies that *IOE-Text* objects have to be indexed by the method *EnglishText-RuleBasedStemming* and the results have to be sent to the index *IndexInvFile-EnglText*.

Figure 6: Class *DefaultIndexer* and an instance object of it

An additional class, called *DefaultsManager*, is defined for collecting default settings. The class *DefaultsManager*, see Figure 7, is responsible for the defaults defined for an application. FIRE provides default settings, which may be changed by the application developer. Defaults are set via the method *setDefault*, whereas the method *getDefault* allows one to access default settings. The method *checkDefaults* performs consistency checks; it tests whether defaults are defined as required and whether the default methods are defined within the application.

 Figure 7: Class *DefaultsManager*

The subclasses of *DefaultsManager* serve to handle the defaults defined for a particular IR database (*DefaultsManagerIR-Database*) and/or user (*DefaultsManagerUser*). Supported by the inheritance mechanism of object-oriented approaches, only those defaults need to be defined which differ from the default settings of an application. In case defaults are defined for an IR database, they overwrite the corresponding default settings of the application. The same principle holds for the defaults defined for a particular user with regard to the default settings of an IR database. User- and database-specific defaults may be modified by an authorized user at run-time.

The classes for managing defaults are defined by the framework, thus instances of these classes are interpretable by any FIRE application. Thus, default settings can not only be exchanged between IR databases and users of the same application but also between different applications.

6 Selecting a Particular Functionality

In FIRE, indexing and retrieval are controlled by modalities, c.f. Section 3. These modalities specify in a declarative way which methods and parameters are to be used in a given situation. Indexing and retrieval modalities can be set at run-time without any programming effort. The ease of selecting particular IR functionality promotes strongly the reusability of the framework.

Each document to be indexed and each query document to be evaluated is associated with an *IndexingModalities* respectively a *RetrievalModalities* object. Figure 8 shows the definition of the class *IndexingModalities* and Figure 9 the definition of *RetrievalModalities*.

 Figure 8: Class *IndexingModalities*

Indexing modalities include an entry for each feature of a document to be indexed. Document features are identified by their names (attribute *NameRIU-Feature*). An entry for a feature, called *IM-Unit*, specifies the method to be used for deriving indexing features and the index(es) in which the results are to be stored. For a feature, several entries may be defined, e.g. for testing alternative indexing techniques.

Retrieval modalities, see Figure 9, are quite similarly organized. For each feature of a document being part of a user's query, an entry called *QM-Unit* is included. Such an entry specifies how a query condition is to be evaluated: index to be looked-up, matching method to be used, etc. Entries are grouped together in an object called *QueryModalities*. *QueryModalities* are associated with *IndexingModalities* and with a method for calculating RSVs. *QueryModalities* include this way all details needed to evaluate a user's query. *RetrievalModalities* may consist of several alternative *QueryModalities*. In this case, the user's query is evaluated in different ways, e.g. by applying different methods for calculating RSVs..

Indexing and retrieval modalities are derived automatically from the default settings valid for the feature values of a document. They can be modified by the user if the user wants a different functionality, e.g. keep indexing results additionally in a private index. In order to ease the use of user-defined modalities, the user may create a master document with individual modalities and use it for the further indexing and retrieval of information.

Figure 9: Class RetrievalModalities

Conclusions

In this paper, we have shown that the (re-)use of indexing and retrieval functionality provided by an IR framework is strongly supported by making transparent which functionality exists and how it is used as well as by allowing the user to select a particular functionality as default and to choose a specific functionality at run-time by setting modalities. We also have shown how indexing and retrieval are organized and important concepts are modeled in FIRE to support the (re-)use of the framework. In the future, we will extend the work reported in this paper to support the graphical presentation of information. A flexible presentation of information requires quite similar mechanisms as have been described here. We are quite confident that our approach can be extended to fulfil the requirements of powerful and flexible user interfaces.

A first FIRE prototype has been implemented providing basic indexing and retrieval functionality for English and German text documents coded in the HTML format as well as English speech documents. A graphical user interface [4] is developed by our project partner at the Robert Gordon University. A World Wide Web interface is also supported, allowing FIRE applications to be accessed over Internet. The implementation of FIRE is based on ETOS a seamless integration of ET++, cf. [8-9], and ObjectStore. The next major implementation steps will be to provide indexing and retrieval functionality for information in tables, to add advanced text retrieval facilities, and to make use of database functionality for IR purposes.

References

- [1]. Agrawal R, Psaila G, Wimmers E L, Zait M. Querying Shapes of Histories. In: Proc. of the 21st Int. Conference on Very Large Data Bases, pp 502-514, 1995
- [2]. Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995
- [3]. Glavitsch U, Schäuble P. A System for Retrieving Speech Documents. In: Proc. of the 15th Annual Int. ACM SIGIR Conference on R & D in Information Retrieval (SIGIR-92), pp 168-176, 1992
- [4]. Hendry D G, Harper D J. Coordinating Information Seeking on Interactive Displays. In: Proc. of the 2nd Int. Conference on Electronic Library and Visual Information Research, De Montfort University, Milton Keynes, UK, 1995
- [5]. James D A. The Application of Classical Information Retrieval Techniques to Spoken Documents. PhD thesis, University of Cambridge, 1995
- [6]. Rumbaugh J, Blaha M, Premerlani W, Eddy F, Lorensen W. Object-Oriented Modeling and Design. Englewood Cliffs/N.J.: Prentice Hall, 1991
- [7]. Sonnenberger G, Frei H-P. Design of a Reusable IR Framework. In: Proc. of the 18th Annual Int. ACM SIGIR Conference on R & D in Information Retrieval (SIGIR '95), pp 49-57
- [8]. Weinand A, Gamma E, Marty R. ET++ – An Object-Oriented Application Framework in C++. In: Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings, pp 46-57, 1988
- [9]. Weinand A, Gamma E. ET++ – a Portable, Homogenous Class Library and Application Framework. In W.R. Bischofberger and H.-P. Frei (eds.): Proc. of the UBILAB Conference '94, Zurich, Switzerland, pp 66-92, 1994