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:

Implementing a Multimedia IR Framework

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

Published in collaboration with the
British Computer Society

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

Implementing a Multimedia IR Framework

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

UBILAB, Union Bank of Switzerland,
P.O.Box, CH-8021 Zurich, Switzerland,
Tel.: +41 1 234 5809, Fax: +41 1 236 4671
E-mail: {bratvold,sonnenberger,frei}@ubilab.ubs.ch

Abstract

A well-designed information retrieval (IR) framework should provide powerful abstractions for common IR concepts and techniques. Central to the success of such a framework is whether these abstractions can be implemented to satisfy users' demands of functionality, performance, flexibility, and ease of use.

This paper discusses implementation issues for a multimedia IR framework in the context of the recently proposed FIRE design. We present the software architecture of the framework, and outline the cornerstone components of its implementation. Issues of reusability, ease of use, and performance are discussed with references to solutions adopted in FIRE.

1 Introduction

Multimedia information retrieval is a wide and diverse field, with information units of different media requiring different approaches to representation, indexing, retrieval, and user interaction. This makes the development of multimedia IR applications a demanding undertaking. Having a framework of reusable components for developing such applications therefore seems appealing. However, the IR community uses a rich variety of algorithms and techniques, and it is a complex and delicate task to integrate these smoothly into a unified and consistent framework.

Typically, a framework determines the overall structure of an application, its thread of control, and the collaboration of its constituent parts [1]. While a well-designed framework can be a great source of code reuse and a powerful aid for application development, it also imposes restrictions. A too general framework leaves much work to the application developer, while a too specific framework has limited applicability. A major problem in developing a framework is therefore to find good abstractions matching the problem domain. A framework for developing multimedia IR applications should provide powerful abstractions for common IR concepts and techniques. Due to the richness of techniques applied in the field of IR, a relatively high level of abstraction seems necessary. The FIRE¹ design described in [2] is an attempt at capturing such abstractions in an object-oriented class hierarchy. Object-orientation is well suited for generalizing over different media and functionality, and at the same time underpins a programming paradigm promising a high degree of reuse.

This paper discusses issues relevant to the realization of a multimedia IR framework with reference to the implementation of FIRE. As often is the case in object-oriented software development, design and implementation are not strictly sequential activities, but overlap in iterative cycles of refinement. We focus on three topics that we have found to be important in developing a multimedia IR framework, and point to how these topics are addressed in the FIRE implementation:

- how to provide framework components that can be applied flexibly in different contexts, thus encouraging and supporting reuse,
- how to make the resulting framework easy to use despite its encapsulation of potentially complex functionality, and
- how to evaluate the significance of performance overheads resulting from the use of a framework as opposed to a custom built application.

The following section presents an overview of the design and implementation of FIRE, outlining the principal components and their relations. We then discuss issues of implementing a multimedia IR framework (section 3), focusing on flexibility and reuse, ease of use and performance. Finally, we discuss briefly FIRE's characteristics as a framework with respect to related work (section 4).

¹ FIRE is an acronym for **F**ramework for **I**nformation **R**etrieval applications.

2 An Overview of FIRE

This section presents a brief overview of the design and implementation of FIRE. For further details on the design, see [2]. In the following elaboration, the reader may find figure 1 to be a useful point of reference.

2.1 Data components

A document in FIRE is modeled by defining a subclass of the abstract class *ReprInfoUnit*. We will use the terms document and *ReprInfoUnit* interchangeably to mean a *ReprInfoUnit* subclass. A *ReprInfoUnit* is composed of multimedia document components modeled as subclasses of the abstract class *InfoObjectElement*. Different media formats are represented as subclasses of *InfoObjectElement*, e.g. *IOE_Text*, *IOE_Date*, *IOE_Speech*, *IOE_Image*, and *IOE_Video*.

The abstract class *Index* is used for representing indexes. Subclasses of *Index* model particular index types. An index consists of a collection of *IndexingFeature* objects together with access structures composed of key values and postings. A posting refers to a *ReprInfoUnit* or an *InfoObjectElement* through an object of the class *IO_Address*. Specialized subclasses of *IO_Address* can be used for referring to documents and document components at an arbitrary level of detail, for example a word in a piece of text.

Classes in FIRE are part of a single rooted class hierarchy. This has certain advantages, for example it allows access control to be implemented uniformly for all classes.

2.2 Functionality

As well as representing data, classes in FIRE also encapsulate methods for performing IR specific operations. Abstract classes like *InfoObjectElement* and *Index* define interfaces of methods that can be overridden in concrete subclasses.

Figure 1: Documents, document components and indexes

Indexing of a *ReprInfoUnit* is a two stage process. First indexing features are derived from the document to be indexed, initiated by invoking a method *Index()* at the *ReprInfoUnit* object. This method in turn invokes other methods for deriving indexing features from each of the *InfoObjectElement* of which the document is composed. Specialized functionality for deriving indexing features is provided for each *InfoObjectElement* subclass according to the media it represents. For instance, for *IOE_Text*, word stems can be derived, while for *IOE_Speech*, n-grams of phonetic units can be derived. A class *Indexer* is used for specifying how indexing features are derived, e.g. with respect to tokenization and normalization. Secondly, each derived indexing feature is inserted into an index. An *Index* object is responsible for processing the derived features to produce posting lists sorted by key values. These are later to be used for efficient exact or approximate matching. Figure 1 illustrates how documents (*ReprInfoUnit* objects) are composed of *InfoObjectElement* objects, and how derived indexing features are inserted into *Index* objects. *IndexingModalities* is a meta-information class that can be used for controlling how indexing is to take place with respect to which document features are indexed, which *Indexer* is applied, and in which indexes the derived indexing features are inserted.

Using a retrieval by template approach, the course of the retrieval process is similar in structure to that of indexing. Indexing features are derived from the query document as for any other document. The derived features are then passed to an index, where they are matched against key values according to some matching criterion. The individual matching scores are weighted and combined, and the result is a list of documents ranked according to a

selected criterion for calculating retrieval status values. Again, there are modalities controlling the process, RetrievalModalities, that can be set to select the functionality to be applied during retrieval.

2.3 Implementation

The previous sections outline the FIRE design at a fairly abstract level. To produce a practical implementation of the framework, a number of issues have to be worked out in more detail. Such issues include how to

- represent functionality and concepts as flexible and reusable building blocks;
- make the framework extensible;
- limit the risks of erroneous use of framework components;
- enforce consistency when parameters are set or modified;
- make available functionality transparent to the application developer and to the end-user;
- support the user interface of the resulting application.

The framework implementation will have to address all of these issues.

FIRE is being implemented using the ETOS persistent object-oriented application framework [3]. ETOS is an integration of Object Design's ObjectStore ODBMS [4] and ET++ [5], a C++ based application framework developed at UBILAB and at the University of Zurich. A principal advantage of object-oriented design is that design concepts map conveniently down to an object-oriented implementation platform. This design-implementation combination is particularly powerful when objects are stored persistently using an ODBMS. Although hard to avoid completely, it is our clear intention to restrict the influence of the particular development platform on the resulting implementation. In particular, we see ObjectStore mainly as an example of an ODBMS, and we aim to restrict dependencies on ObjectStore to those parts of FIRE dealing directly with low level issues of object creation and storage. Thus, at higher levels of the FIRE implementation, the underlying ODBMS is not visible, allowing us to focus fully on issues of IR and frameworks rather than ODBMSs.

The core functionality of FIRE deals with indexing, retrieval and the support of a graphical user interface. This involves the following entities: the abstract classes *ReprInfoUnit*, *InfoObjectElement*, and *Index*, and associated modalities/meta-information. These core concepts of the framework only know and deal with abstract entities, i.e. they do not know the difference between, or even the existence of, concrete classes such as *IOE_Date* and *IOE_Speech*, but rely on these subclasses conforming to the interface of their abstract superclass *InfoObjectElement*. In this respect, the abstractions provided by the design are powerful, but they also impose restrictions on how concrete subclasses can be manipulated. Implementing FIRE gives valuable experience about which abstractions are effective, which leads to improved designs in succeeding iterations of the design-implementation cycle

Figure 2: Software architecture of an IR application using FIRE

Figure 2 illustrates the architecture of an IR application developed with FIRE. The application code extends and specializes FIRE, filling in application dependent gaps and setting parameter values. It should be noted that the relative sizes of the elements in the figure do not necessarily relate to the size of the corresponding software components. In particular, the code needed for building standard applications will tend to be relatively small compared to FIRE itself.

The implementation of FIRE is in progress at UBILAB and at the Robert Gordon University. At UBILAB we are working on the retrieval engine and core functionality, while our project partner takes care of the graphical user interface, see [6] for details. In the current prototype implementation we support storage and retrieval of text (in ASCII and HTML formats) through classes *IOE_Text* and *IOE_String*. Normalization methods for text have been implemented for both the English and the German language. In addition, we support storage and retrieval of spoken

English through an *IOE_Speech* class, while work on spoken German is in progress. A number of other basic *InfoObjectElement* subclasses are implemented, e.g. *IOE_Integer*, *IOE_Date*, and *IOE_PersonName*, together with a range of corresponding methods for matching. A few experimental applications have been implemented, e.g. for retrieving textual abstracts of scientific papers and for retrieving spoken English news broadcasts. At present we support a vt100/ASCII user interface, a windows-based GUI, and a WWW/Netscape interface. However, FIRE is still at an early stage of development, and much work remains. In particular, we aim to focus on generalizing the framework design further, and to make use of database indexes for IR purposes. Concerning further multimedia support, we have concrete plans for supporting the retrieval of information from tables, which is particularly relevant to the financial sector. This may generalize further into retrieval from charts and graphs.

The remainder of the paper is devoted to discussing and elaborating on some of the issues brought up in this section, using solutions adopted in the FIRE implementation as examples.

3 Framework Implementation

3.1 Flexibility and Reuse

Central to the success of a framework is the degree to which its components are reusable in different applications. Reusability depends largely on the flexibility by which components can be connected, enhanced and applied in different contexts. In the implementation of FIRE, the concrete subclasses of *InfoObjectElement* and *Index* provide building blocks which to a large extent are independent and self-contained. These building blocks have a uniform appearance conforming to the interface of their respective abstract superclasses, while providing their own specialized code for deriving indexing features, for matching, and for their presentation in a graphical user interface. This uniformity allows IR components to be reused flexibly in different applications. For instance, an application dealing with speech retrieval may use the *IOE_Speech* class for representing speech, and corresponding methods for deriving indexing features and for matching. Another application may make use of *IOE_Text* or *IOE_String*. Both these applications are driven by exactly the same framework implementation, knowing only that it deals with *InfoObjectElements*, and relying on the concrete subclasses to provide specialized functionality.

Although FIRE as a framework first and foremost supports the development of whole applications, it may also be feasible to reuse components of FIRE in other IR related contexts. This requires a loose and explicit coupling between the framework components, which mostly is the case for FIRE. An *Index* subclass may for example be reused in a document management system. The programmer would have to provide (or reuse from FIRE!) most notably functionality needed for maintaining the index structure, for instance methods for computing hash values or for comparing objects. *InfoObjectElement* subclasses may prove attractive for use in applications requiring querying of or uniform handling of multimedia objects, since data representation and operations for manipulation as well as for supporting a graphical user interface are readily available.

Another important issue is that of extendibility. The framework can be enhanced with new functionality or media formats by creating new subclasses of existing abstract classes. This is for instance attractive for *InfoObjectElement*, where new media formats can be added as subclasses without interfering with existing parts of the framework or with existing applications. It should be noted that when adding a new subclass of *InfoObjectElement*, a number of associated methods must be provided. These include methods for deriving indexing features and for handling the user interface. The newly added components and their functionality are immediately integrated into, and fully usable from, the existing framework.

3.2 Ease of Use

FIRE provides a rich object model that is designed for the implementation of complex IR applications. Making a framework powerful in terms of functionality is however of limited value if it is hard to use, or if its use requires the understanding of many implementation specific details. It is therefore an important aim in the development of FIRE to make it user friendly. By this we do not only mean a good interface to the end-user, but also that the framework components, although encapsulating potentially complex functionality, should be easy to use when building applications. In particular it should be easy to develop applications requiring standard functionality; a “simple things simple” policy.

FIRE's class hierarchy must be implemented such that the supported functionality is easily accessible. Developing a new application, in particular if standard functionality is sufficient, should involve little more programmer effort than to specify the document structure as a *ReprInfoUnit* (subclass) using *InfoObjectElement* building blocks, and to select from the range of functionality offered by the framework. This is a declarative approach, where the application developer selects some key properties for the application, and leaves the rest to the framework. Thus, most of the complexity of the application is within FIRE rather than in the code describing the application. This has important positive implications for the resources needed to develop and maintain applications.

For demanding, non-standard applications, special functionality not already supported by FIRE may be required. Application development is then a two phase process, where the programmer first adds the new

functionality by extending FIRE, and then proceeds to describe the application in declarative terms as outlined above.

There are several aspects of the FIRE implementation that contribute to making the framework easy to use. The structure of documents is specified by creating a subclass of `ReprInfoUnit`. This subclass has `InfoObjectElement` attributes corresponding to the document structure, as depicted in figure 1. There is no need to supply methods for indexing, retrieval or for supporting a graphical user interface, since this code is already provided with, and automatically invoked from the framework. FIRE exploits meta-information about `ReprInfoUnit` subclasses, partly supported by ET++. For instance, it is possible to iterate over all attributes of a class. This allows the framework to index or to display a `ReprInfoUnit` subclass object by directly accessing its `InfoObjectElement` attribute values, and then to invoke appropriate methods for deriving indexing features or for displaying. Thus, from a relatively small application description, FIRE can produce a full-fledged application, complete with functionality for indexing and retrieval, and with a ready-to-use graphical user interface. Hence, application development using FIRE will often involve only minimal amounts of programming.

For specifying indexing and retrieval modalities, there is also modest programming efforts required. Methods for common indexing and retrieval functionality are supported, and all that needs to be done is to select what is desired in the particular application. For example, for selecting tokenization and stemming functionality for an `IOE_String` feature representing a report title, the modalities may be set with a command like

```
document->SetModality("title","PorterIndexer")
```

The framework also supports the setting of modalities by the end-user, for instance through menus of available functionality.

Another mechanism designed to reduce the complexity of building or using an application is that of system supported defaults. In FIRE we will support defaults at different levels. For example for an `IOE_String`, the framework can have as a default Porter's stemming algorithm. The application programmer can override this by stating that a particular `IOE_String` attribute in the particular application should have a different default. The end-user may again override this by setting his/her own value. Whenever indexing features are derived from an `IOE_String`, for instance when evaluating a query, the currently valid default will be used if no other value is explicitly provided. Supporting default values does not solve all problems of navigating in a large space of parameter settings, but can nevertheless make a framework considerably easier to use.

FIRE is a relatively complex piece of software, and it should certainly not be necessary to know all of it in detail in order to use it. Pree et al [7] suggest the idea of providing *cookbooks* of recipes for how to use a framework for certain tasks. A recipe can take the form of a hypertext document where the application developer performs a list of steps that guide him/her through the development process. Recipes can be supplemented with various tools, such as automatic code generators or graphical browsers to aid the developer. We find this idea very attractive for use with FIRE, and we have started to provide some documentation in HTML, possibly to be incorporated into a more extensive cookbook at a later stage.

A particular benefit of having an easy-to-use framework is that developing prototype applications is fast. This is of particular interest to IR researchers who want to evaluate specific IR techniques without having to focus on the broader scope of developing complete applications.

3.3 Performance

The benefits of high-level abstractions or flexibility often introduce performance overheads. In FIRE, a number of decision points are deliberately made explicit rather than being hard coded into a special purpose application. Consider for example the choice of stemming algorithm for a piece of text. FIRE cannot make any assumptions about which algorithm to use or indeed if the text is to be stemmed at all. Instead, indexing modalities have to be inspected, and the appropriate functionality invoked. Clearly this is less efficient than having the choice of algorithm hard-coded into the application. Similar overheads occur in a number of places in the framework, e.g. the choice of matching algorithm, the choice of which index to use, the choice of calculation function for retrieval scores, etc. FIRE relies on several forms of such meta-information that have to be "consulted" about the behavior and appearance of the application.

Another potential source of overhead is the ODBMS, which is used for storing document collections, indexes, and some administrative data. In particular, indexing may prove to be problematic since indexes supported by DBMSs tend to be designed for database applications where exact matching is normally used. For IR applications, approximate matching is often desirable, and we will work towards supporting this using DB indexes. An ODBMS does however provide a high level of abstraction in that persistence of complex objects can be virtually transparent. As framework developers it is our task to examine if and when this level of abstraction carries a performance overhead. This will allow us to tune the framework to a certain degree. Application developers using the framework can later benefit directly from our experiences, rather than each having to make tedious experimentation and performance tuning on his/her own. At the present stage of the FIRE development it is however

too early to give any conclusive indication of the significance of the performance overhead incurred by the framework.

Although an important issue, performance is not our main concern when developing FIRE. We view reusability and ease of use to be more important properties for the framework in the long term, while any performance overheads may be compensated for by continuous improvements in hardware speed, or more importantly, in reduced time for application development and maintenance.

4 Related Work

Much relevant work is in progress both in the areas of frameworks and reusable software, and in the areas of multimedia information retrieval. Our work lies in the intersection of these paths of research.

For most IR systems, reusability is not an explicitly stated aim, and is not a factor dictating the system design. Nevertheless, there are several examples of IR systems being reused, both as systems and as components. Both the SMART [8] and the INQUERY [9] systems have been used in a number of different IR applications, and have also been reused by other than their developers. INQUERY supports an interface to its retrieval engine allowing application programmers to produce customized front ends. ECLAIR [10] goes a step further in terms of reusability by providing an extensible IR class library that can be used for providing IR functionality to applications.

Being a framework, thus explicitly designed for reuse, FIRE provides “glue” in form of modalities and meta-information which when used for application development, ensures flexibility and smooth co-working of the framework components. A framework provides a much more dynamic development interface to the individual application than do traditional APIs or configuration languages. As IR applications become complex, we believe this to be of increasing importance. FIRE also provides support for reuse by several groups of users: IR researchers wanting to extend the framework, application developers, and end-users through a graphical user interface.

An interesting approach is that taken by Özsü et al [11] in developing an object-oriented multimedia database system for a news-on-demand application. This is essentially a multimedia IR system, and has several common points with FIRE in the modeling of documents and document components. However, FIRE has far richer support for representing indexing and retrieval functionality, and is also more explicitly designed for reuse.

Components of IR systems are frequently interchanged and reused between research practitioners. These can include simple parts like tokenizers or stemmers, but also more complex components like index structures or retrieval engines. Also in a number of commercial products are components reused, e.g. the commercial retrieval engines from Fulcrum (Ful/Text) and Verity (Topic) have proved popular. In the development of FIRE we would also like to benefit from reusing software components, in particular for providing specialized functionality for storage, indexing and matching in InfoObjectElement subclasses.

5 Conclusion

We have discussed a number of implementation issues for a multimedia IR framework with reference to the FIRE implementation. In particular, we have focused on three issues. We have argued that a high degree of reusability and flexibility can be achieved by decoupling IR concepts and modeling them as abstract classes. We further believe that ease of use, despite the internals of the framework being complex, is essential. In FIRE we achieve this by using abstract classes and meta-information for IR concepts, by providing ready-to-use building blocks requiring little programming effort by the application developer, and by providing support for several levels of defaults to ease configuration. Using a framework approach rather than building an application from scratch, may well involve run time overheads and reduced performance. We believe, however, that this is in most cases fully compensated by considerable resource savings in application development and maintenance.

Finally, we concluded that although many present IR systems have been reused both as components and in parts, a framework can offer significant additional benefits, maybe most notably a uniform appearance of framework components, and “glue” in form of meta-information. This gives a flexible and dynamic interface between the framework and the individual application that allows framework components to be reused flexibly and seamlessly.

Acknowledgments

The authors thank David Hendry at the Robert Gordon University for many fruitful discussions on the design of FIRE. Thanks are also due to Andrin Bachmann, Ruxandra Domenig, and Andreas Pfenninger, all from ETH Zurich, who have been of great help in implementing the FIRE prototype. We would also like to thank our other colleagues at UBILAB who have provided valuable input and help, and in particular to Manuel Bleichenbacher for technical assistance with ETOS, and to David James for his cooperation in integrating speech retrieval functionality.

References

1. Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995
2. Sonnenberger G, Frei HP: Design of a Reusable IR Framework. In: Fox, Ingwersen, Fidel (eds) Eighteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Seattle, USA, 1995, pp 49-57
3. Bleichenbacher M, Schmidt D: ETOS - An Example for the Integration of a Class Library with an ODBMS (extended abstract). In: ET++ Workshop on Developing Building Blocks and Frameworks. Department of Computer Science, University of Zurich, Switzerland, 1995
4. Lamb C, Landis G, Orenstein J, Weinreb D. The ObjectStore Database System. Communications of the ACM 1991; 34(10): pp 50-63
5. Weinand A, Gamma E: ET++ - a Portable, Homogeneous Class Library and Application Framework. In: Bischofberger WR, Frei HP (eds) Proceedings of the UBILAB Conference '94, Zurich, Switzerland, 1994, pp 66-92
6. Hendry DG, Harper DJ: Coordinating Information Seeking on Interactive Displays. In: Collier M, Arnold K (eds) Proceedings of the 2nd International Conference on Electronic Library and Visual Information Research. Aslib, London, UK, 1995, pp 127-136
7. Pree W, Pomberger G, Schappert A, Sommerlad P. Active Guidance of Framework Development. Software -- Concepts and Tools 1995; 16: 136-145
8. Salton G. The SMART Retrieval System - Experiments in Automatic Document Processing. Prentice-Hall, 1971
9. Callan JP, Croft WB, Harding SM: The INQUERY Retrieval System. In: Proceedings of the 3rd International Conference on Database and Expert Systems Applications, 1992, pp 78-83
10. Harper DJ and Walker AM: ECLAIR. An Extensible Class Library for Information Retrieval. The Computer Journal 1992; 35(3): 256-267
11. Özsu MT, Szafron D, El-Medani G, Vittal C. An Object-Oriented Multimedia Database System for a News-on-Demand Application. Multimedia Systems 1995; 3: 182-203