Design of a Reusable IR Framework

Gabriele Sonnenberger and Hans-Peter Frei

UBILAB, Union Bank of Switzerland
P.O. Box, 8033 Zurich, Switzerland
{sonnenberger,frei}@ubilab.ubs.ch

Abstract

In this paper, we describe the design of a reusable IR framework, called FIRE, that is being implemented to facilitate the development of IR systems. In addition, FIRE is designed to support the experimental evaluation of both indexing and retrieval techniques. First, we discuss the development of reusable software in the IR domain and derive essential criteria for the design of an IR framework. Next, we sketch the object model developed for FIRE. We present the basic concepts and their modeling and show how the components interact when performing indexing and retrieval tasks.

1. Introduction

Everyday information is heterogenous and usually consists of structured and unstructured parts. In addition, it may be composed of different media. Furthermore, information is invariably embedded in a context and is associated with additional information. In the past the focus in information retrieval (IR) has been on text, which was mostly considered to be unstructured. Recently, different methods have been developed to improve the storage and retrieval of real world information. Most notable are approaches for

- integrating fact and text retrieval in order to deal more appropriately with information that is structured to various degrees (e.g. Fuhr [1992]),
- indexing and retrieving non-textual media such as images or speech (Glavitsch & Schäuble [1992]),
- connecting information units by links and providing both a more intuitive access to information (e.g. Halasz & Schwartz [1994]) and retrieval techniques that exploit these links (e.g. Frei & Stieger [1995]).

Broadening the scope of IR makes the development of systems more complicated and laborious. Furthermore, most of these advanced approaches are at an early stage of development. Thus, there is special need for evaluating and comparing the various approaches by experimental applications. The goal of the work described in this paper is to ease the development of IR systems and to support the experimental evaluation of indexing and retrieval techniques by providing reusable software in the form of an IR framework.

The paper is organized as follows: Section 2 gives an overview of the different approaches for developing reusable software for the IR domain and sketches previous work. Subsequently, we identify essential criteria for the design of a framework that is tailored to the IR domain. Section 4, the main part of this paper, describes the object model that we have developed for FIRE[1], as we call our IR framework. We present the basic concepts and their modeling and then show in section 5 how the components interact when performing indexing and retrieval tasks. The paper concludes with some implementation details and remarks on future work.

2. Reusable software in the IR domain

Generally, software reuse is discussed within the object-oriented software design and programming paradigm. Krueger [1992] gives a comprehensive survey of the software reuse literature. Two approaches which seem most promising with regard to application to the IR domain are the definition of a *class library* and the development of a *framework*. In the first case, the goal is to define a relatively complete and practical library of methods. In certain areas such as numerical or statistical analysis, this has been done quite successfully. Applying this approach to IR, we would have to define methods for stemming terms, indexing natural language texts, determining the similarity of items (e.g. strings, numbers), etc. A framework is a program skeleton that defines the basic concepts of an application domain and how its components are interrelated. In the IR domain, we would need to model concepts like text, index and query, to define appropriate methods, and to make the components work together. Only if such a framework provides suitable generalizations and abstractions, does it allow to realize various applications and different IR techniques.

In the IR community, the development of reusable software has so far not been a special concern. Exceptions are some full-fledged systems like SMART (Salton [1971]) and INQUERY (Callan et al. [1992]) that have been reused by others than the developers. LyberWorld (Hemmje et

1. FIRE is an acronym for “Framework for Information Retrieval Applications”. The FIRE project is developed in cooperation with the IR group at the Robert Gordon University, Aberdeen.

al. [1994]) for instance utilized INQUERY as retrieval engine and built a prototype for a novel user interface on top of it. Full-fledged systems like INQUERY are usually not trimmed for reuse, thus their adaptability to the specific needs of an application is restricted. To our knowledge, the only work aiming explicitly at reusable IR software was undertaken by Harper & Walker [1992], who developed ECLAIR, an extensible object-oriented class library. ECLAIR is specialized on automatic indexing of texts and a best-match retrieval approach. However, the goal of the class library is less to support the development of IR applications than to deliver IR functionality to the developers of general-purpose application systems.

IR methods are not as established as methods for numerical or statistical problems and there is less agreement as to which data types the methods are to be based on. Let us consider for instance automatic indexing. As long as we merely derive single terms from unstructured ASCII text, we probably might succeed in defining a reusable method. However, if we want to consider other input as well, e.g. structured text, and derive, for instance, indexing features that are annotated with their position within the source, the situation is far more complicated. It is unreasonable to assume that such advanced methods can be developed without considering interrelations. Taking these arguments into account, the class library approach seems too restricted and not well suited for developing reusable software in the IR domain. Thus, we are working toward a framework for providing reusable software in the IR domain.

3. Design criteria

The question “what are the characteristics of a good framework and how is one designed” has been discussed in the literature, for instance in Wirfs-Brock & Johnson [1990] and Booch [1994]. Most of the publicized frameworks focus on user interfaces and are relatively domain-independent. The framework described in this paper is, in contrast, specialized to a particular domain, IR. Thus, special requirements are imposed. In the following, we will discuss these requirements and outline the design criteria which we consider most important for developing a reusable framework for the IR domain.

3.1 Generalization

A framework is usually developed to support a whole set of related applications in contrast to developing individual applications from scratch. In the IR domain, we face the problem 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 shall be represented. Moreover, a useful IR framework should provide the option of applying different models for indexing and retrieving information (e.g. the probabilistic or the vector space model). To cope with these problems, we need to develop a *generic* model for managing and retrieving information.

3.2 Well-defined scope

The scope of an IR framework has to be defined very carefully in order to achieve reusability. With an underspecified framework, application developers have to redesign and recode their solutions over and over despite the fact that a solution might be equivalent for differing applications. By overspecification, we may impose unnecessary constraints, thus making the development of an application more difficult and restricting the reusability of the framework. From our experience, the scope of an IR framework should be as follows:

- The framework has to define the basic concepts of the IR domain (index, indexing feature, query, etc.) and the relations between them. To define the interaction between the IR concepts and the information units to be indexed and retrieved, we also need a generic model of the representation of information units.
- As already discussed above, the modeling of information units cannot be covered by a framework, but must be done individually. However, an IR framework can support the application developer by providing elementary data types that are frequently used in the IR domain to represent information units (e.g. name, date, bibliographic reference, etc.) and by specifying a set of general methods on these data types (present, edit, match, etc). Our idea is that an application developer models information units by composing such elementary data types rather than modeling information units from scratch.
- In dealing with information, some general tasks must be solved that are largely independent of the functionality envisaged by an application. These tasks are the creation, storage, and removal of information units as well as the controlled access to the units stored in a system. It is important that an IR framework cares about access control as access privileges are often to be considered when presenting and manipulating information units.

3.3 Transparency

Frameworks are “white boxes” (Wirfs-Brock et al. [1990]). Therefore, transparency is generally a crucial design criterion. In the IR domain, there are additional requirements with regard to the modeling of information units and the organization of the retrieval process. A transparent modeling of information units is an important prerequisite for building flexible user interfaces, which for instance present information in a situation-specific manner (how shall a user interface present information flexibly without ‘knowing’ what it is presenting?). Consequently, an IR framework has to provide constructs for delivering metainformation. Making the retrieval process transparent is important for the development of user interfaces, too.

Consider for instance a user interface that aims at helping the user to optimize a complex query by drawing the attention to the effects of single search conditions and their possible combinations. This requires some knowledge about the evaluation of a query.

3.4 Efficiency

Similar to the properties mentioned earlier, efficiency in execution is also of general importance. The developer of a framework has to devote special attention to the time-consuming operations of an IR system. These are essentially search and approximate match algorithms. In the first case, a framework cannot offer general solutions, but can provide facilities to reduce computing complexity by restricting the search space. In the second case, efficiency can be increased both by a type-specific index organization (e.g. providing a sorted index for numerical values) and by carefully tuned matching methods.

The acceptance of a framework will depend essentially on the efficiency of its implementation. Providing general constructs and techniques to support efficiency in execution is a particular challenge. Furthermore, with a well-designed framework, it should be possible to replace basic methods by optimized methods for time-critical algorithms of the specific application to be developed.

4. The FIRE object model

In this section, we outline the FIRE object model. The design and implementation of FIRE is based on an object-oriented approach. First, we give an overview of the class hierarchy. Subsequently, we present those classes in more detail which are most important for IR. Our goal is to convey the basic ideas underlying the object model. Therefore, we present the model in a slightly simplified way and omit some of the technical details.

The object model of FIRE defines the basic IR concepts and supplies constructs that support the realization of an IR application. The modeling of concrete types of information units (e.g. books, tables, etc.) is not part of the object model as this is an application-specific task. To avoid misunderstandings, we use the term 'object model' for the concepts and constructs provided by FIRE and the term 'data model' for the application-specific modeling of information units.

4.1 Sketch of the class hierarchy

The class *InformationObject* constitutes the root of the class hierarchy. It defines the basic operations for managing and manipulating information units in a system. Furthermore, this class defines the basic constructs for controlling the access to information units. Via inheritance, these constructs are also defined for the descendants of *InformationObject*. This is essential for a consistent control of information access.

InformationObject is an abstract class which serves to handle communalities between classes. All operations of *InformationObject* are abstract operations which provide a uniform interface. The implementation of these operations must be done by concrete subclasses. Figure 1 shows the operations defined by *InformationObject*. For describing classes and objects, we use the notation developed by Rumbaugh et al. [1991]. For those not familiar with the notation, we explain the most important symbols in Figure 2.

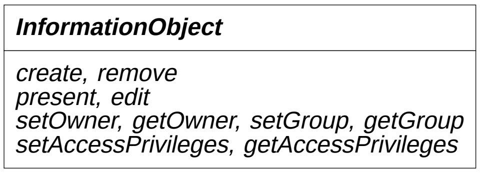

InformationObject
create, remove
present, edit
setOwner, getOwner, setGroup, getGroup
setAccessPrivileges, getAccessPrivileges

Figure 1: Operations defined by *InformationObject*

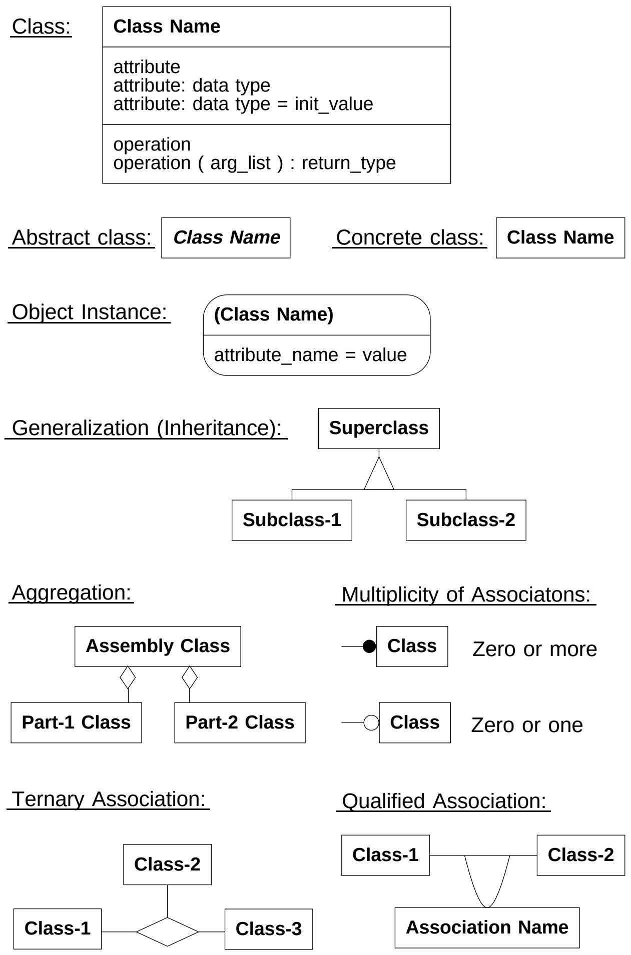


Figure 2: Legend to figures

InformationObject has three subclasses which are also abstract classes. These classes are shown below:

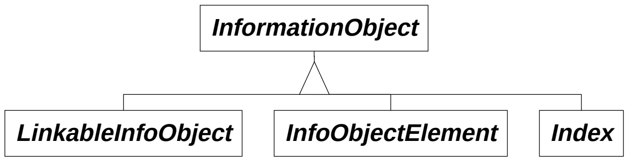


Figure 3: Subclasses of *InformationObject*

LinkableInfoObject is the base class of all objects that can be linked to other objects. It offers operations for defining and removing links and operations for following a link to its source or destination. The most important subclasses of *LinkableInfoObject* are *ReprInfoUnit*, *InformationStock* and *QueryRecord*. These classes have fairly different functions. *ReprInfoUnit*, which is described in section 4.2, provides a basic skeleton for modeling the information units to be managed by an application, *InformationStock* can be used to structure a database, and *QueryRecord* supplies information about queries.

By the use of *InformationStock* the usually large number of *ReprInfoUnit* objects stored in a database can be partitioned. Partitioning can be done recursively since an *InformationStock* object may contain other *InformationStock* objects. Furthermore, an object may belong to different *InformationStock* objects. This class helps to delimit computing effort by restricting the search space.

QueryRecord serves to record queries and their results. An object may record a whole query as well as a single condition of a query (the modalities are controlled by *ReprInfoUnit* and its subclasses). Since this class is a subclass of *LinkableInfoObject*, we can establish links between objects. In this way, we can record a complex query in a fine-grained manner and preserve the relations between its constituents. Such a declarative description of a query and its evaluation supports the development of advanced user interfaces.

InfoObjectElement defines a set of data types that are to be used for the application-specific modeling of information units. This class is described in more detail in section 4.3.

The class *Index* is responsible for managing the indexing features derived for information units. Moreover, it supports the retrieval of information. We elaborate on this class in section 4.4.

4.2 Class “ReprInfoUnit”

The class *ReprInfoUnit* serves to realize an application-specific data model. Concrete subclasses of *ReprInfoUnit* define how information units of a certain type are represented in a particular application. Dealing with text for instance, there may be features like *Title*, *Authors*, *Text*, and *PublicationDate*. Such modeling is part of the data model of an application. In order to make the data model transparent and provide an application-independent interface, the definition of *ReprInfoUnit* includes an attribute, called *DataDictionary*, that can be used to supply metainformation.[2]

ReprInfoUnit and its subclasses are responsible for organizing the indexing and retrieval of the information units they are representing. The modalities can be defined specifically for each of the features constituting a *ReprInfoUnit*. For specifying indexing modalities, *ReprInfoUnit* provides a generic ternary relation between the value of a feature, which is an *InfoObjectElement*, the class *Index* and the class *IndexingFeature*. This relation is qualified by the class *IndexingParams* which serves to specify the parameters to be used for indexing. Figure 4 shows this generic definition.

When modeling a subclass of *ReprInfoUnit* for representing a certain type of information units, the application developer can define for each feature

- by which methods the feature values are to be indexed,
- the type of the indexing features to be derived and
- by which indexes the results are to be managed.

The definition of *ReprInfoUnit* also allows the application of different methods and parameters for the same feature values in parallel. Figure 5 shows as example a concrete subclass of *ReprInfoUnit*, called *ReprText*, that represents textual units by a feature “Text” and a feature “Authors”. The “Text” feature is indexed by different methods using specific parameters and the results are kept in different *Index* objects.

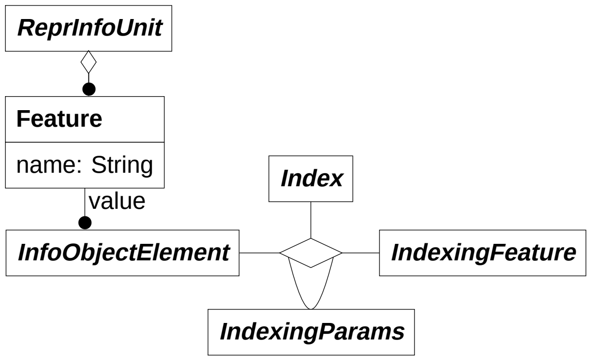


Figure 4: Generic definition for specifying indexing modalities

The modalities for retrieving information are specified in a quite similar way. Here, a ternary relation is defined between the value of a *Feature*, the class *Index* and the class *IndexEntry*. The relation is qualified by the class *MatchingParams*.

2. At present, the application developer has to provide metainformation. We will investigate how we can support the application developer in integrating a data model in the object model of FIRE and how the framework can automatically provide metainformation.

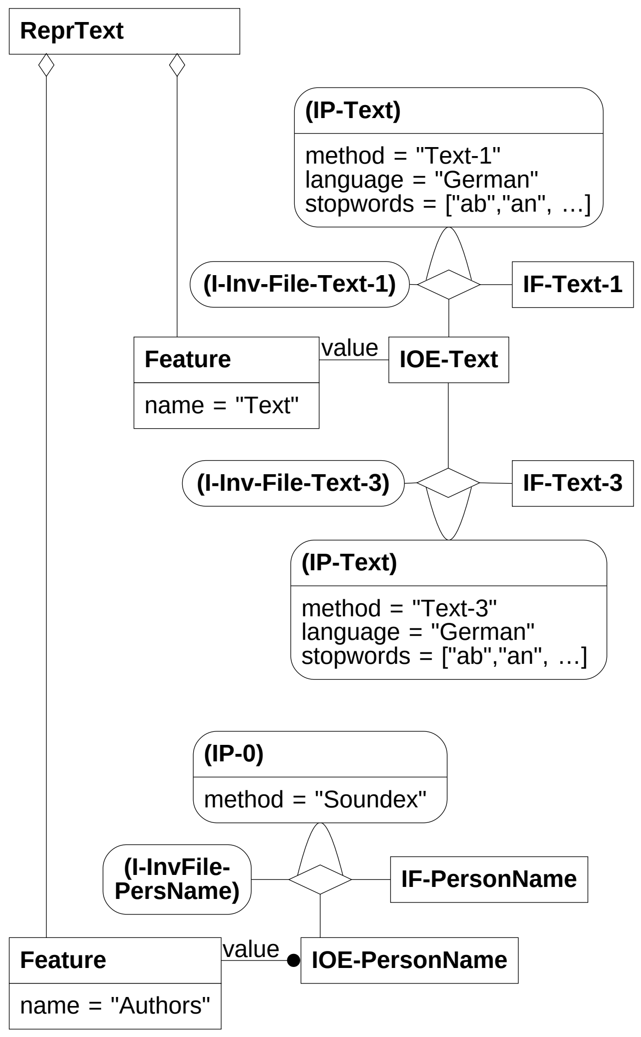


Figure 5: Example of a concrete subclass of *ReprInfoUnit*

4.3 Class "InfoObjectElement"

The class *InfoObjectElement* defines a set of data types that are intended to be used for the application-specific modeling of information units. The framework comes with definitions for elementary data types like string, integer, name, date, set, and list (see Figure 6). If needed, this set of data types can be easily extended by the application developer. *InfoObjectElement* helps to reduce the effort for developing an application and supports a uniform representation of information units, e.g. the author of a paper is specified in the same way as the author of a chart table.

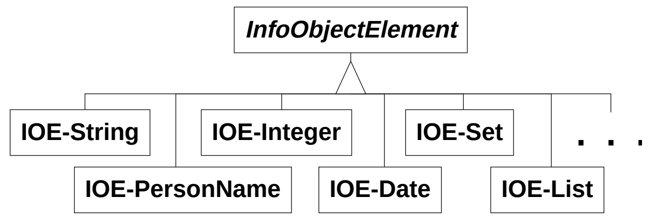
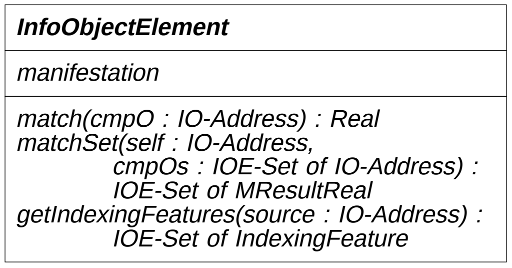


Figure 6: *InfoObjectElement* and some of its subclasses

The class definition of *InfoObjectElement* includes an attribute, called *manifestation*, that serves to store the object represented by an instance. Furthermore, *InfoObjectElement* provides operations for solving matching and indexing tasks. The definition of *InfoObjectElement* is depicted in Figure 7. To keep the design as simple as possible, we use the shortcuts *IOE-Set of <Class>* and *IOE-List of <Class>*. The first shortcut denotes a set and the second a list of objects of the given type. During implementation, the shortcuts will be expanded and respective subclasses of *IOE-Set* and *IOE-List* will be defined in order to support type checking.



```
InfoObjectElement
manifestation
match(cmpO : IO-Address) : Real
matchSet(self : IO-Address,
            cmpOs : IOE-Set of IO-Address) :
            IOE-Set of MResultReal
getIndexingFeatures(source : IO-Address) :
            IOE-Set of IndexingFeature
```

Figure 7: Definition of *InfoObjectElement*

The operation *match* calculates the similarity between two objects. One of these objects is the object receiving a *match* message; the other object is given by an address. The address of an *InformationObject* is specified by an object of the class *IO-Address*.

A concept for addressing an *InformationObject* is of general importance, e.g. in modeling links, indexing features, and index entries. Therefore, we have decided to provide a general definition (see Figure 8). The FIRE model supports complex objects, i.e. objects composed of other objects. An *IO-Address* specifies a root object, i.e. the assembly object. By recursively giving a discriminator, e.g. the name of a feature, and the identifier of a constituent object, the address of an *InformationObject* can be specified at any level of detail.

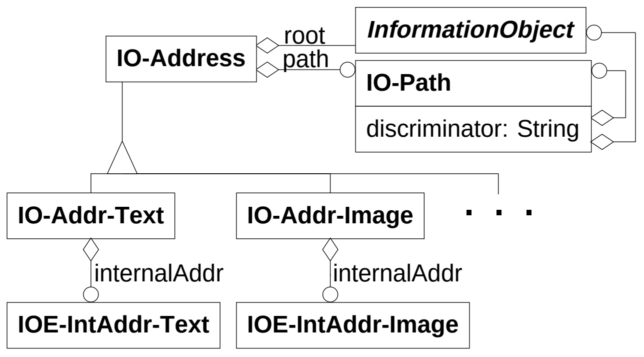


Figure 8: *IO-Address* and two important subclasses

For certain types of *InformationObject*s, an internal address may also be specified. Figure 9 shows as example the definition of internal addresses of texts and images.

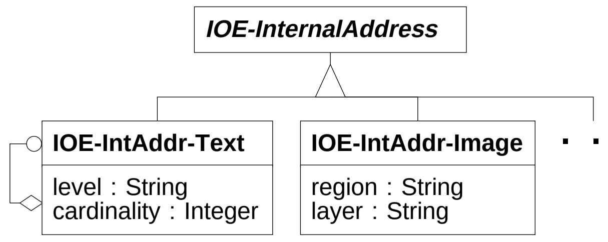


Figure 9: *IOE-InternalAddress* and two concrete subclasses

The calculation of the similarity of objects depends on the type of the objects and on the methods to be applied. Different matching methods may require quite different parameters. A method developed for integer values for instance may consider the mean value and the standard deviation of a collection, while a method designed for strings may take the phonetic similarity to a certain degree into account. To achieve a uniform interface, we define a class *MatchingParams* and a set of subclasses that cover the different extensions of matching parameters (see Figure 10).

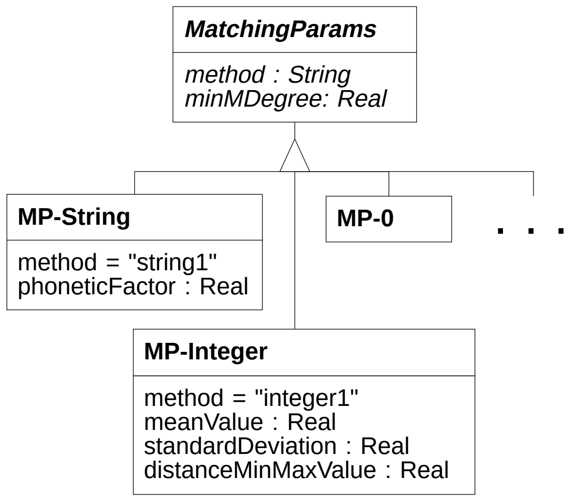


Figure 10: *MatchingParams* and a few subclasses. *MP-0* is defined for matching methods that do not require specific parameters.

As a further complication, different matching methods may be defined for a single subclass of *InfoObjectElement*, e.g. matching of integers with or without context information. To provide a uniform solution and to support a transparent modeling, *InformationObject* is twofold associated with *MatchingParams (*see Figure 11). The first relation, *definedMP*, serves to specify which methods and which combinations of parameters are defined for a given subclass of *InformationObjectElement*. The second relation, *activeMP*, may be used to select one of the methods as the one actually to be used.

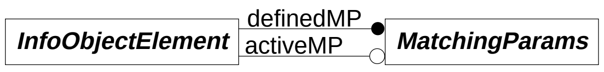


Figure 11: Relations between *InfoObjectElement* and *MatchingParams*

Concrete subclasses of *InfoObjectElement* may provide default *MatchingParams*. For an *InfoObjectElement* instance, individual parameters may be defined by associating it with a *MatchingParams* instance via an *activeMP* link (see for example Figure 12).

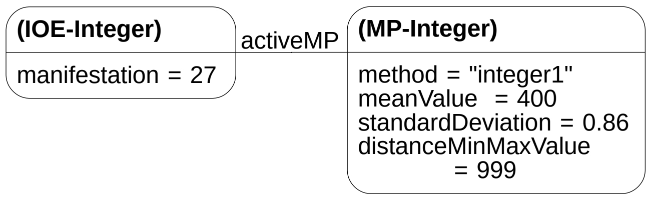


Figure 12: An *InfoObjectElement* instance, for which individual parameters are defined.

InfoObjectElement provides an additional operation, *matchSet*, that allows to compare an object with a whole set of objects. This matching mode is especially convenient if matching is based on the calculation and comparison of keys, e.g. hash codes, phonetic keys or n-grams. With optimized matching methods, the key for a given object needs to be calculated only once rather than being repeatedly recalculated.

The operation *getIndexingFeatures* derives indexing features from an *InfoObjectElement*. The operation is given the address of the assembly object as parameter to associate the *IndexingFeature*s appropriately with their source.

For providing indexing parameters, we have chosen a similar approach as has been described for matching parameters, since we encounter quite similar problems. *InfoObjectElement* is related to the class *IndexingParams* by a *definedIP* and an *activeIP* link. Subclasses of *InfoObjectElement* define concrete parameters. In contrast to *MatchingParams*, it is unreasonable to define individual parameters for an *InfoObjectElement* instance, since an object is hardly retrievable without knowing how it is indexed.

The result of a *getIndexingFeatures* operation is described by *IndexingFeature* objects. An *IndexingFeature* principally consists of a feature, which is an *InfoObjectElement*, and a source specification (see Figure 13).

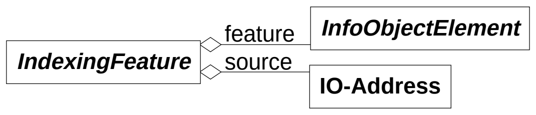


Figure 13: Generic definition of an *IndexingFeature*

Additionally, an *IndexingFeature* may be associated with a position within the source. Figure 14 shows as example a few concrete subclasses of *IndexingFeature*, for which a position is defined.

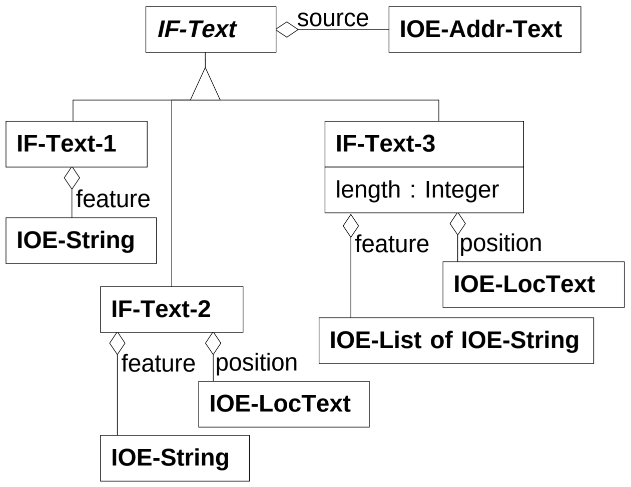


Figure 14: A few concrete subclasses of *IndexingFeature*

4.4 Class “Index”

The class *Index* is responsible for solving special indexing and retrieval subtasks. It defines a set of abstract operations (see Figure 15), which provide a uniform interface independent of a particular retrieval model.

```
Index
createIndexEntries(features :
            IOE-Set of IndexingFeature) : Boolean
retractIndexEntries(sources :
            IOE-Set of IO-Address) : Boolean
update() : Boolean
clear() : Boolean
getAllSources() : IOE-Set of IO-Address
getIndexingFeatures(source : IO-Address) :
            IOE-Set of IndexingFeature
getIndexEntries(source : IO-Address) :
            IOE-Set of IndexEntry
retrieve(query : IOE-Set of IndexEntry) :
            IOE-Set of MResultReal
```

Figure 15: Operations of *Index*

The operation *createIndexEntries* derives *IndexEntry* objects from a given set of *IndexingFeature*s. This operation does not cause any updating, e.g. sorting the entries of an inverted file or calculating weights. Such tasks have to be invoked explicitly by an *update* message. We chose to separate the processes in order to avoid unnecessary computations, for instance sorting an *Index* again and again when indexing a whole set of *InformationObject*s.

By the operation *retractIndexEntries*, the *IndexEntry* objects referring to a given set of *InformationObject*s can be retracted. The operation *clear* removes all entries of an *Index*. Furthermore, *Index* defines operations for providing information about an *Index*. These are the operations whose names start with ‘*get*’ (see Figure 15).

The *Index* receives a query in form of a set of *IndexEntry* objects. The retrieval process and the origin of the *Index-Entry* objects are explained in section 5.

For an efficient retrieval, different types of indexing features and retrieval models require specific types of index structures, e.g. B-trees or sorted lists. These structures are provided by subclasses of *Index* (see Figure 16). *Index-Unordered* for instance stores *IndexEntry* objects in an unordered list by using hash codes, while *Index-InvertedFile* works with an ordered list of entries.

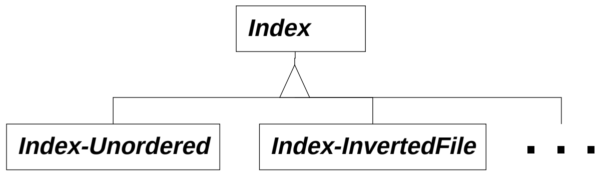
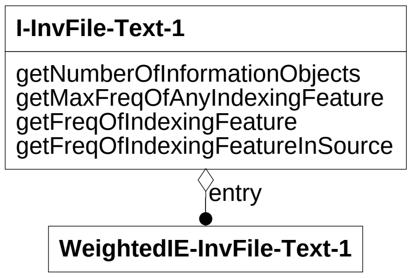


Figure 16: Subclasses providing specific index structures

An *Index* for a specific type of information unit or a specific retrieval model is built by introducing a new subclass. Essentially, the application developer has only to define the structure of the *IndexEntry* objects and provide the methods specific to the envisaged retrieval model. *I-Inv-File-Text-1* of Figure 17 is an example of such a concrete *Index* class.



I-InvFile-Text-1
getNumberOfInformationObjects
getMaxFreqOfAnyIndexingFeature
getFreqOfIndexingFeature
getFreqOfIndexingFeatureInSource
entry
WeightedIE-InvFile-Text-1

Figure 17: A concrete *Index* subclass

In FIRE, we distinguish between indexing features and index entries. *IndexingFeature*s are the items derived by indexing *InformationObjects*. An *Index* derives from these features *IndexEntry* objects, which provide the basis for retrieving information. The structure of an *IndexEntry* depends on the type of the *Index*. An *IndexEntry* for an inverted file consists of a key and a set of postings. A posting specifies the source where the feature occurs. Additionally, it may specify the position of the feature within the source. Figure 18 shows a concrete subclass of *IndexEntry* which allows the position of a feature to be specified.

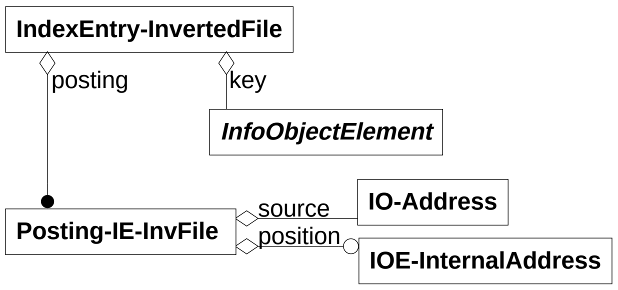


Figure 18: A concrete subclass of *IndexEntry*

5. Indexing and retrieval with FIRE

5.1 Indexing

FIRE divides indexing into three tasks:

1. Organization of the indexing process
2. Derivation of indexing features
3. Management of the indexing features derived and calculation of the data needed for realizing a specific retrieval model, e.g. corpus information, or indexing weights.

The indexing process is organized by *ReprInfoUnit* objects. This way we can index the different features of a *ReprInfoUnit* individually. The derivation of *IndexingFeatures* is done by *InfoObjectElements*, the feature values of *ReprInfoUnit* objects. This is a satisfactory way, since these objects know best the necessary details about their internal structure. The *IndexingFeatures* derived are managed by an *Index* object which also provides the required data about the features and the corpus.

Indexing is initiated by sending an *index* message to a *ReprInfoUnit* object. The object passes the individual details about the indexing modalities to its feature values and invokes the derivation of *IndexingFeatures* by a *getIndexingFeatures* message. The *IndexingFeatures* are sent to the *Index* object specified by the respective indexing modalities. Finally, an *update* message may sent to the *Index* objects in order to invoke necessary updating operations. The organization of the indexing process is depicted in Figure 19.

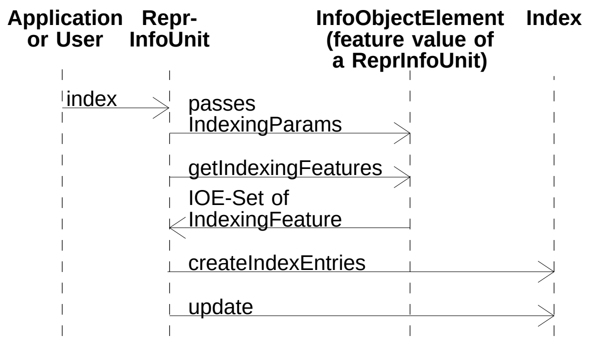


Figure 19: Typical interaction trace for performing indexing

This way of organizing the indexing process has several advantages:

- The framework does not impose a specific indexing technique and the overall organization of the indexing process is independent of a particular method.
- Different indexing techniques can be applied in parallel.
- The results achieved for a feature value by applying different methods can be managed by separate indexes. This prepares the basis for working with different retrieval models in parallel.

5.2 Retrieval

FIRE favors retrieval interfaces where the user creates a model information unit by (partially) filling a template to specify the information need (similar to query-by-example). Such interfaces are comprehensible and easy to use, and – from a software-engineering point of view – the framework can offer most support to the application developer. However, FIRE is not restricted to query-by-example interfaces, but also allows other approaches.

A template-based query is created, represented, and indexed in quite the same way as an information unit to be stored in a FIRE application. This way, it makes no difference whether we create a query object or use an existing *ReprInfoUnit* as query. Further, we avoid having to define separate methods for solving similar tasks.

In FIRE, the object representing a query is responsible for organizing the retrieval process. It determines the order by which the individual conditions are to be evaluated and combines the partial results achieved.

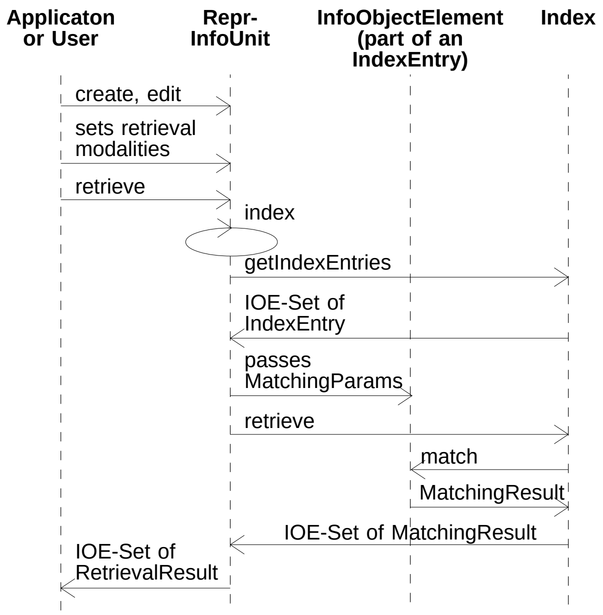


Figure 20: Typical interaction trace for retrieving information

The user can specify the retrieval modalities individually for each of the feature values of a *ReprInfoUnit* object, e.g. choose the *Index*es to be looked up or select a specific matching method. Before starting retrieval, the *ReprInfoUnit* object representing a query is indexed. However, this is only done if the object has not already been indexed, since it is an object of the application database. The retrieval process is invoked by sending a *retrieve* message to a *ReprInfoUnit* object. *ReprInfoUnit* asks the respective *Index* for the resulting *IndexEntry* objects. It passes the *MatchingParams* to be used to the *InfoObjectElements* that constitute these *IndexEntry* objects. Finally, the entries are given to the *Index* to be looked up as parameter of the *retrieve* operation.

Note, the *Index* keeping the entries of a query and the *Index* to be looked up may be different objects. The *Index* delegates matching tasks to the *InfoObjectElement*s involved. This is convenient, since the *Index* does not have to care about the type of indexing features it is managing. Figure 20 illustrates the retrieval organization.

In FIRE, the parts specific to a retrieval model are concentrated in two classes, namely *Index* and *ReprInfoUnit*. The overall retrieval process is independent of the retrieval model(s) implemented. In a framework, this is highly desirable, since extensions and modifications are easier to perform if they affect only a small set of classes.

6. Conclusions and future work

We have identified essential criteria for the design of reusable software in the IR domain and have presented the object model underlying FIRE. In particular, we have shown that with appropriate generalizations and abstractions we can provide a program skeleton that allows to realize various indexing techniques and retrieval models. Furthermore, a FIRE application may work with different indexing and retrieval techniques in parallel.

We are currently implementing the basic concepts of the FIRE object model including approximate matching methods for various data types. The implementation of FIRE is based on ET++, cf. Weinand et al. [1989]. The persistent storage of objects is handled by ObjectStore. ObjectStore and ET++ are fully integrated.

In the future, we will focus on

- supporting the application developer in integrating the data model of an application into the object model of FIRE, and
- providing basic facilities for dealing with different media.

The development of applications will also be a significant part of our work, since a framework needs applications to validate its strategic and tactical decisions (cf. Booch [1994]).

Acknowledgements

The authors would like to thank David Harper, Dave Hendry and Jan-Jaap IJdens from the Robert Gordon University, Aberdeen, as well as Daniel Hümbeli of UBILAB for fruitful discussions on the FIRE model. Thanks also to Tore Bratvold, UBILAB, for his support on finishing this paper.

References

Booch [1994]:
G. Booch: *Object-Oriented Analysis and Design* (Second Edition). Redwood City/CA: Benjamin/Cummings, 1994

Callan et al. [1992]:
J.P. Callan, W.B. Croft and S.M. Harding: The INQUERY Retrieval System. In: *Proc. 3rd Int. Conf. On Database and Expert System Application*, pp. 78-83, 1992

Frei & Stieger [1995]:
H.P. Frei and D. Stieger: The Use of Semantic Links in Hypertext Information Retrieval. In: *Information Processing and Management*, Vol. 31, No. 1, pp. 1-13, 1995

Fuhr [1992]:
N. Fuhr: Integration of Probabilistic Fact and Text Retrieval. In: *Proc. of the 15th Annual Int. ACM SIGIR Conference on R & D in Information Retrieval (SIGIR-92)*, pp. 211-222, 1992

Glavitsch & Schäuble [1992]:
U. Glavitsch and P. Schäuble: 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

Halasz & Schwartz [1994]:
F. Halasz and M. Schwartz: The Dexter Hypertext Reference Model. In: *Communications of the ACM*, Vol. 37 (2), pp. 30-39, 1994

Harper & Walker [1992]:
D.J. Harper, and A.D.M. Walker: ECLAIR: An Extensible Class Library for Information Retrieval. In: *The Computer Journal*, Vol. 35 (3), pp. 256-267, 1992

Hemmje et al. [1994]:
M. Hemmje, C. Kunkel and A. Willett: LyberWorld – A Visualization User Interface Supporting Fulltext Retrieval. In: *Proc. of the 17th Annual Int. ACM SIGIR Conference on R & D in Information Retrieval (SIGIR-94)*, pp. 249-258, 1994

Krueger [1992]:
Ch.W. Krueger: Software Reuse. In: *ACM Computing Surveys*, Vol. 24(2), pp. 131-183, 1992

Rumbaugh et al. [1991]:
J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen: Object-Oriented Modeling and Design. Englewood Cliffs/N.J.: Prentice Hall, 1991

Salton [1971]:
G. Salton: *The SMART Retrieval System – Experiments in Automatic Document Processing.* Englewood Cliffs/N.J.: Prentice Hall, 1971

Wirfs-Brock & Johnson [1990]:
R.J. Wirfs-Brock and R.E. Johnson: Surveying Current Research in Object-Oriented Design. In: *Communications of the ACM*, Vol. 33 (9), pp. 104-124, 1992

Wirfs-Brock et al. [1990]:
R. Wirfs-Brock, B. Wilkerson, L. Wiener: *Designing Object-Oriented Software.* Englewood Cliffs/N.J.: Prentice Hall, 1990

Weinand et al. [1989]:
A. Weinand, E. Gamma, R. Marty: ET++ – An Object-Oriented Application Framework in C++. In: *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pp. 46-57, 1988