A Software Framework for General-purpose Information Retrieval

Adrian O'Riordan

A Software Framework for General-Purpose Information Retrieval

Adrian P. O'Riordan
Department of Computer Science
University College Cork - National University of Ireland, Cork
Cork, Ireland
phone: +353-21-902143
fax: +353-21-274390
e-mail: a.oriordan@cs.ucc.ie

Abstract

A software frameworks is a way of tackling the issue of software re-usability. The concept of a software framework is introduced and we design a general-purpose framework for probabilistic information retrieval. We discuss software engineering aspects of retrieval systems in general and previous efforts at building re-usable IR systems. Design patterns are utilised throughout the framework. Issues associated with a C++ implementation are also touched upon.

1 Introduction

We developed a software framework for probabilistic information retrieval from the first principles of object oriented analysis and design. Analysis involved the study of the requirements of information retrieval test-bed systems from the perspective of the entities that make up the vocabulary of the problem domain. Design, both static and dynamic, followed analysis and encompassed a range of activities, i.e. the requirements are fleshed out and extra design-level constraints added. The choice of actual programming language and data storage facilities were postponed until quite late in the software development lifecycle. The system was eventually implemented in a modern (2-level instantiation) object oriented language, C++.

2 Domain Area: Information Retrieval

Automatic information retrieval (IR) dates from the early 1950s, and has been receiving increased interest with the popularity of the Internet since then [1][2][3]. The basic retrieval process is as follows. A user issues queries to the information retrieval system, which has a collection of documents the user wishes to search. The user query is a formulation of his/her information need (IN). Query formulation can take many forms: pre-defined keywords, Boolean expressions of index terms, free text or marked documents. The system then produces output which consists of a subset of the document collection. This subset is often ranked by document relevance. The measure of document relevance is called the retrieval status value (RSV). The documents should in some sense be "relevant" to the query so that the user's information need is satisfied. Of course the process is often iterative, with the user refining his/her query and re-submitting the request so as to produce a more satisfactory output. Subsequent retrieval runs are usually termed user feedback. Several iterations of query modification are often necessary to achieve acceptable results.

We adopt a probabilistic representation of text, as well as for the other essential entities that make up an IR system, documents and queries. Probabilistic models have been investigated in IR research since the early 60s, but have never become the major model in commercial systems. Research systems have shown the feasibility of the approach, but undoubtedly obstacles still exist preventing its widespread adoption. Commercial systems continue to be based primarily on well-understood Boolean and vector space models. Thankfully, probabilistic models can simulate all the the other major models quite easily, so they are general-purpose in that sense.

In the probabilistic model, the primary probability of interest is the relevance judgement for a particular document given that the document is described using a particular set of terms, i.e. $P(d \text{ is } R|d_{rep})$ where d_{rep} signifies the representation of the particular document d and R indicates relevance [4]. The query is implicit in the model. Relevance assessments aren't needed to start, but what is required are estimates of the distribution of terms within both the sets of relevant and non-relevant documents.

3 A Framework Approach

One definition of a framework, given by Cotter, is "an extensible library of cooperating classes that makes up a reusable design solution for a given problem domain" [5]. A simpler definition that we prefer is "a class hierarchy plus a

model of interaction among the objects instantiated from the framework" [6]. Frameworks can be viewed as miniature applications with dynamic as well as static structure—programmers add the extra pieces unique to their requirements. Frameworks are built using OO technology, and their adaptability and extensibility is a direct consequence of this. An application derived from a framework can be seen to communicate with the framework in two different ways. These are called the *calling API* and the *subclassing API*. When a user or application programmer extends a framework by writing a concrete class conforming to one of the framework's abstract classes, he or she is communicating using the subclassing API. When a user calls an operation which is part of the framework directly, he or she is communicating via the calling API. The calling and subclassing APIs are utilised by users playing the client role and ensemble role respectively.

Of course frameworks do have some disadvantages. Frameworks make heavy use of inheritance. Subclassing is a form of tight coupling and thus violates one of the tenets of software engineering. Mistakes or omissions from the base classes trickle all the way down the class hierarchy. A full re-compile may be necessary to remedy such a situation. Another problem relates to our use of C++. The problem is that pointers to a base class can only call operations defined in the base class. Therefore new operations declared in derived classes cannot be called polymorphically via these base class pointers. A good problem analysis minimized these problems though.

An unfortunate fact is that current IR systems utilise a large number of different techniques. Obviously, then, an IR library/framework cannot cover all aspects of an IR system. What is needed is a library/framework that concentrates on the core tasks in IR, and provides support that is superior to choosing from existing libraries which were not specifically written for IR. Choosing the core tasks becomes the central, most important, part the design.

With both SMART and INQUERY (see Section 5), the system designers' intentions have been to support a particular IR model (vector space and Bayesian Networks respectively). The systems are extensible and reusable by others only if they follow their basic approach to IR research. Investigators who adopt other approaches have to look elsewhere for library support. Efforts at re-use of systems' code/design by groups other that the developers have been ad hoc. For example, Hemmje et al. have used INQUERY in their LyberWorld system [7] and numerous others have utilised SMART as a starting point for their work.

The basic requirements of our framework are as follows:

- support for the basic domain objects: documents, collections, queries,

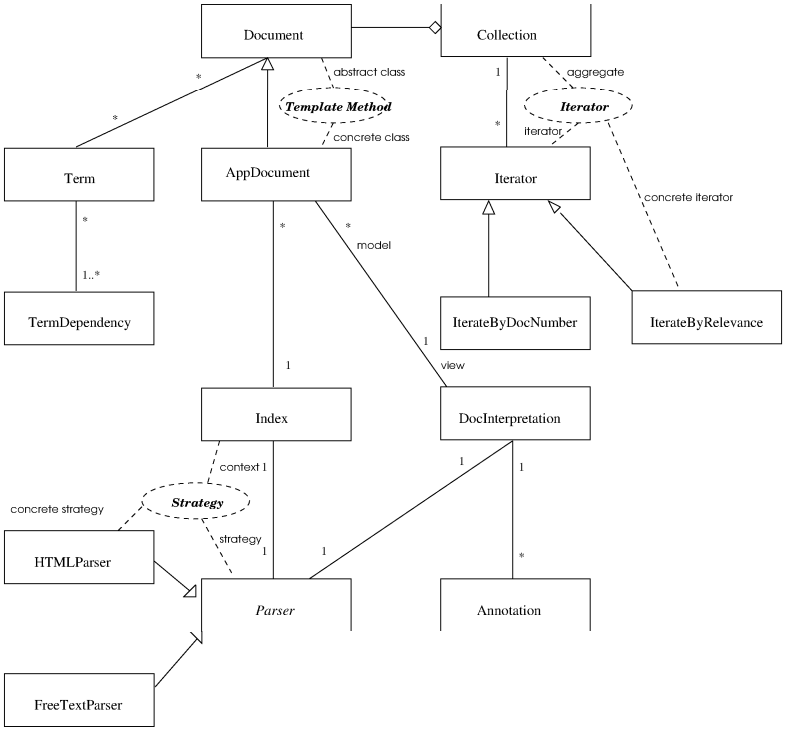

Figure 1: UML Class Diagram for Document Modelling

etc. These hide the details of the source and format by providing objects that are convenient for client applications.

- customisation of the initial text encodings and mapping of text fields to index terms.
- generation (and removal) of indices in a general way that is independent of the particular retrieval model in use.
- support for a very general model of retrieval that can be specialised by developers.
- formatting and sorting of valued documents list.
- a relevance feedback mechanism and the associated update of relevant objects.

A UML class diagram for the document modelling component is shown in Figure 1.

4 Behaviour of the Framework

In the design and documentation of our framework we make use of many design patterns [8]. Specifically we employ iterator, observer, template method, strategy and singleton. All of these, bar the last, are behavioural. Singleton is a creational pattern.

A document collection is a static entity, in the sense that it represents largely a set of data structures for storing documents. It is implemented as a container class. But there is a need to view or process documents in different orders. Hence we need methods for traversing container objects and we want them to be de-coupled from the data structures themselves. For example, you may want to iterate through the documents in order by document number or by ranked relevance. The question arises as to whether this flexibility should be built into a Collection class. But this would require something like a switch statement (in C++), which is undesirable from the perspective of re-usability. Adding a new operation (a means of traversal) then requires a recompilation. It is preferable if each new operation can be added separately, and the Collection class itself is largely independent of the traversals that apply to it. We adopt this approach by implementing operations such as traverseByDocNumber as a new object called an iterator (or generator). This is the iterator design pattern.

An iterator is a means of traversing a container class analogous to a pointer traversal of a linked list. The iterator is passed as a parameter to the now abstract Collection as required. All iterator classes are derived from a (abstract) base class Iterator. The major requirement on Iterator is that it declares operations for all the derived classes' responsibilities.

Here we have described an external iterator, where the client is responsible for controlling the traversal, i.e. advancement has to be requested explicitly by the client. In contrast, there exist internal iterators, which control the traversal order themselves. Two advantages of external iterators are that they (1) keep the container class itself small and (2) allow multiple simultaneous traversals. Benefits of this design pattern include the fact that adding new traversals is now relatively easy and that more than one traversal can be active at a time [1]

The Query and UserRequest objects possess a one-to-one correspondence. When a UserRequest changes, we need some mechanism for updating the associated Query object correspondingly. This requirement is a consequence of our decision, in the first place, to partition the system into co-operating

[1]Here we are referring to the iterator as being the active entity; this is not to be confused with Booch's "active iterator", which is the same as our external iterator. Booch is referring to the client's active role [9].

classes, in this case to split UserRequest and Query. We desire loose coupling without a compromise in consistency. When a user issues a request, say in the form of a Boolean expression, this needs to be translated into the form of a Query object, replacing the previously active query.

We chose the observer pattern to implement this dependency. The key roles are the subject (UserRequest) and the observer (Query). The mechanism by which the observer pattern is implemented may seem confusing at first. It is in fact the subject that first notifies the observer of any changes in its state. When a user issues a new request, UserRequest notifies Query, Query then calls UserRequest (the subject) in turn to synchronise state.

Consider the objects Document and AppDocument. What is the advantage of partitioning the document abstraction into two classes? The answer is that fundamentally documents in IR systems fulfill multiple roles. In our design, the class Document takes care of low-level activities such as opening, reading, saving and closing files. (Document files are assumed to be read-only.) It also participates in a relationship with Collection. AppDocument, in contrast, represents the internal documents of the system and instead participates in relationships with Index and DocInterpretation. We need to define the relationship between Document and AppDocument in a technical form that makes their roles clear. We make use of another design pattern called template method.

Essentially template method involves the participation of two classes, one of which is defined as a subclass of the other. In our case, AppDocument is a subclass of Document. Certain algorithms are defined as skeletons in the superclass, deferring some steps to subclasses. An example is the open operation. The skeletal open() defined in Document handles file precessing, which is assumed to be a static requirement, whereas the extended open() in AppDocument handles the activities relating to the internal representation of a document. In our implementation the indexer is invoked. Following the terminology in Gamma et al., open() is a template method. This is perhaps an unfortunate choice of term since template is used to describe generic classes and functions in C++.

The main advantages of introducing this extra level of indirection is that AppDocument can be changed without recompiling the Document class and multiple AppDocument objects can co-exist facilitating radically different internal document representations and retrieval models. This is orthogonal to the fact that AppDocuments can already have different indexing procedures associated with them. It should be noted that with the template method pattern, superclasses call the operations of their subclasses. In the more conventional use of inheritance, child objects usually call their parents. We instead have a type of programming sometimes called inheritance-by-extension.

The template method pattern exhibits the inversion of control that is characteristic of the framework approach to software architecture. Unfortunately, inheritance compromises encapsulation by exposing implementation details to inherited classes [10]. Exposing superclass attributes (making them publicly visible in C++) is the problem. The remedy we adopt is to make attributes visible to subclasses but hidden otherwise. Protected visibility in C++ enables this.

Associated with each AppDocument is a DocInterpretation object. An application document may have many different interpretations depending on the type of retrieval being performed. DocInterpretation is the bridge between AppDocument, the internal representation of a document, and the Parser class, the abstract class representing text parsing activities. Derived classes of Parser will implement parsing algorithms for different document formats. For example, the class HTML Parser and FreeTextParser implement algorithms for parsing HTML and unformatted text respectively.

A general design pattern exists for the situation when you have many related classes differing only in their behaviour. For example, when you need different variants of an algorithm. The design pattern is called strategy and the main roles involved are context and strategy. The role strategy is played by an abstract class, Parser. DocInterpretation fills the role of context. Parser has an operation parse(), defined in all (leaf) subclasses, that is invoked by DocInterpretation, which passes the data needed by Parse as parameters. The strategy pattern is a type of delegation [11]. Gamma et al. identify the following benefits (eliminates conditional statements, gives choice of implementations) and liabilities (communication overhead, increased number of objects) of the strategy pattern [8].

We employ one final design pattern called singleton. Singleton ensures that a class has only a single instance, i.e. only a single instance can exist at any point in time. We designed the retrieval engine so that RetrievalRun and FeedbackRun are singleton classes for a number of reasons. The computation involved in doing a retrieval run (whether ad-hoc or routing) is intensive, so we made the requirement that only one process is active at any time. We maintain strict control over how and when clients access this instance. Singleton offers us more flexibility than class operations (static members in C++). It is easy to remove the singleton restriction if needs be, say for example we wanted to make use of a parallel processing environment or see an advantage in having multiple retrieval runs executing concurrently.

5 Related Work

Two important existing IR systems which are available for reuse, SMART and INQUERY, are briefly described below. An object-oriented class library for IR is also outlined, as is a proposed software framework for IR.

SMART embodies the vector space model of IR [2]. SMART includes a complete set of programs for indexing and document matching, including natural language pre-processing code. SMART has continued to be used, developed and extended by researchers at Cornell and by IR groups elsewhere.

INQUERY is an IR system developed at the University of Massachusetts [12]. INQUERY models text and queries using Bayesian networks to help identify relevant documents. Retrieval and routing are viewed as a probabilistic inference process. Retrieval based on a combination of evidence is emphasised. INQUERY has been extensively tested in TREC and is available for others to use as a basis for their work.

ECLAIR (an Extensible Class Library for Information Retrieval) is an object-oriented class library implemented in C++ which can be used to construct information retrieval systems and applications [13]. It makes use of an ODMS (object-oriented database management system) for persistent object storage. The design is such that applications can exploit the features of the ODMS which includes support for modeling complex objects, concurrent access to data, and reliable processing of data in the presence of system failures.

FIRE (Framework for Information Retrieval Applications) is a reusable framework for IR [14]. FIRE is being developed by Sonnenberger *et al.* at UBILAB in co-operation with the IR group at the Robert Gordon University in Aberdeen. FIRE is designed to support the experimental evaluation of both indexing and retrieval techniques. An object model was developed to model the IR process and give support to retrieval system development. Their design makes use of Rumbaugh's object-oriented design method. The implementation of FIRE is based on ET++, a portable, homogeneous class library and application framework. The persistent storage of objects in handled by ObjectStore.

6 Summary

Software frameworks and design patterns have a lot to offer IR system developers. The existence of better class libraries, database systems and CASE tools, though helpful to development, will not cope with all the complexity involved in building modern retrieval applications and supporting the

extensive empirical testing of research systems that has become the norm. A framework specially designed for the retrieval domain offers constraints that can bring increased re-usability. We believe IR is homogenous and well-understood enough for a framework to be extremely useful.

References

[1] C.J. van Rijsbergen. *Information Retrieval.* Butterworths London, 2nd edition, 1979.

[2] G.A. Salton and M.J. McGill. *Introduction to Modern Information Retrieval.* McGraw Hill International, 1983.

[3] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley, 1999.

[4] S.E. Robertson and K. Sparck Jones. Relevance weighting of search terms. *Journal of the American Assoc. for Info. Sci.*, 27:129–146, 1976.

[5] S. Cotter and M. Potel. *Inside Taligent Technology.* Addison-Wesley, 1995.

[6] T. Lewis, editor. *Object-Oriented Application Frameworks.* Manning Publications Co., 1995.

[7] M. Hemmje, C. Kunkel, and A. Willett. Lyberworld — a visualization user interface supporting fulltext retrieval. In *Proc. of the 17th ACM SIGIR*, 1994.

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

[9] G. Booch. *Object-Oriented Analysis and Design with Applications.* Addison-Wesley, 1994.

[10] A. Snyder. Inheritance and the development of encapsulated software systems. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming.* MIT, 1987.

[11] R. Johnson and J. Zweig. Delegation in c++. *Journal of Object-Oriented Programming*, 4(11):22–35, 1991.

[12] J. Broglio, J.P. Callan, W.B. Croft, and D.W. Nachbar. Document retrieval and routing using the inquery system. In D.K. Harman, editor, *Proc. of the 3rd Internl. Text REtrieval Conference (TREC-3)*. NIST Special Publication 500-225, 1995.

[13] D.J. Harper and A.D.M. Walker. Eclair: an extensible class library for information retrieval. *The Computer Journal*, 35(3):256–267, June 1992.

[14] G. Sonnenberger and Hans-Peter Frei. Design of a reusable ir framework. In *Proc. of the 18th ACM SIGIR*, 1995.