

Apple
Information Access Toolkit
v1.0
Programmer's Guide

Apple Computer, Inc.
© 1996, 1997 Apple Computer, Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Computer, Inc. Printed in the United States of America.

No licenses, express or implied, are granted with respect to any of the technology described in this book. Apple retains all intellectual property rights associated with the technology described in this book.

Every effort has been made to ensure that the information in this manual is accurate. Apple is not responsible for printing or clerical errors.

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Apple, the Apple logo, Macintosh, and Mac OS are trademarks of Apple Computer, Inc., registered in the United States and other countries.

Adobe Illustrator and PostScript are trademarks of Adobe Systems Incorporated, which may be registered in certain jurisdictions.

FrameMaker is a registered trademark of Frame Technology Corporation.

Helvetica and Palatino are registered trademarks of Linotype Company.

ITC Zapf Dingbats is a registered trademark of International Typeface Corporation.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

The paper used in this book meets the EPA standards for recycled fiber.

Contents

	Figures, Tables and Listings	i
Preface	Overview of this Manual	v
Chapter 1	Introduction to the Apple Information Access Toolkit	1-1
	Some Possible Applications	1-3
	RecipeSwap	1-3
	How It Was Done	1-4
	Indexing Facility	1-4
	Search Facility	1-6
	Analysis and Filtering	1-8
	Storage and Document Type	1-9
	Construction with IAT	1-10
Chapter 2	Overview of IAT Content	2-1
	Facilities of IAT by Category	2-3
	Index	2-4
	Accessor	2-6
	Analysis	2-8
	Corpus	2-12
	Storable	2-14
	Storage	2-15
Chapter 3	Designing an Application	3-1
	Determining High Level Requirements	3-2
	Determining the External Interfaces	3-2
	Mapping to IAT Classes	3-3
	Internal Task Design	3-5
	Recipe Query	3-6
	Submit Recipe	3-11
	Duplicate Recipe	3-15
	Stop Word Maintenance	3-16
	Database Creation	3-18

Primitive Types	4-3
Globals	4-3
Exceptions	4-3
Exception codes	4-4
Throwing Exceptions	4-4
Memory Allocation	4-5
The Memory Functions	4-6
Base Classes	4-10
IAObject	4-10
IAStruct	4-11
Deletion of Allocated Memory	4-13
IADeleteOnUnwind	4-13
IADeleteArrayOnUnwind	4-14
IADeletePointerArrayOnUnwind	4-15

Choosing an Index Type	5-3
Index Types Currently Available	5-4
Comparison of Searches Available	5-5
Index Size vs. Speed	5-7
Common Operations	5-8
Creating an Index	5-8
Establishing an Existing Index	5-11
Updating an Index	5-14
Iterating Through the Documents in an Index	5-19
Merging Indexes	5-20
Compacting an Index	5-22
Index Class Category Reference	5-23
Header Files in the Index Category	5-23
Class Specifications	5-25
DocInfo	5-25
FreqPosting	5-27
FreqPS	5-29
FreqTerm	5-31
IAIndex	5-31
IAIndexTypes	5-43
InVecIndex	5-45
InvertedIndex	5-46
TermIndex	5-49
TermInfo	5-59
TFComponent	5-63
TFVector	5-63
VectorDocInfo	5-66

VectorIndex	5-67
Class Utilities	5-72
Typedefs	5-74
Extern Data	5-76
Constants	5-77
Index Exceptions and Error Handling	5-78

Chapter 6

Accessor Category 6-1

Choosing an Accessor Type	6-3
Query Logic	6-4
Query Analysis	6-5
Common Operations	6-6
Building an Accessor	6-6
Answering Queries	6-7
Answering a Simple Ranked Query	6-8
Answering a Query by Example	6-11
Answering a Boolean Query	6-14
Describing a Document	6-16
Finding Related Words	6-18
Accessor Class Category Reference	6-20
Header Files in the Accessor Category	6-20
Class Specifications	6-22
IAAccessor	6-22
IAHit	6-26
IAProgressReport	6-29
InVecAccessor	6-32
InvertedAccessor	6-33
RankedAccessor	6-36
RankedHit	6-43
RankedProgress	6-45
RankedQueryDoc	6-46
TWComponent	6-47
TWVector	6-48
VectorAccessor	6-51
Typedefs	6-53
Constants	6-53
Accessor Exceptions and Error Handling	6-54

Chapter 7

Analysis Category 7-1

Understanding Tokens and Terms	7-3
Understanding Tokenizers	7-4
Understanding Filters	7-6

Existing Filters	7-6
Filter Sequence	7-8
Creating Analysis Subclasses	7-9
Creating a SimpleAnalysis Subclass	7-9
Creating a Subclass of IAnalysis	7-10
Creating a Subclass of ITokenFilter	7-12
Creating a Subclass of ITerm	7-13
Creating a Text Utility	7-13
Analysis Class Category Reference	7-15
Header Files in the Analysis Class Category	7-15
Class Specifications	7-17
AlphaTokenizer	7-17
DocTextCharStream	7-21
DowncaseFilter	7-23
IAnalysis	7-24
ITerm	7-27
IToken	7-31
ITokenFilter	7-33
ITokenStream	7-35
ICharStream	7-37
ShortWordFilter	7-42
SimpleAnalysis	7-43
StringTerm	7-45
Constants	7-47
Exceptions	7-48

Chapter 8

Corpus Category 8-1

Introduction	8-3
The HFS Implementation	8-4
HFS Corpus	8-4
HFSTextFolderCorpus	8-5
Common Procedures	8-5
Using a Corpus to Provide Documents	8-5
Creating a New Corpus	8-6
Establishing an Existing Corpus	8-7
Using an HFSCorpus to Locate a Document in HFS	8-7
Creating Corpus Subclasses	8-7
Creating a Subclass of IACorpus	8-8
Creating a Subclass of IADoc	8-9
Creating a Subclass of IADocIterator	8-10
Creating a Subclass of IADocText	8-11
Creating a Subclass of HFSIterator	8-13
Corpus Class Category Reference	8-16
Header Files in the Corpus Category	8-16
Class Specifications	8-17

DirectoryInfo	8-17
HFSCorpus	8-17
HFSDoc	8-21
HFSDocText	8-25
HFSVolumeInfo	8-29
HFSIterator	8-32
HFSFolderCorpus	8-35
HFSFolderDoc	8-39
IACorpus	8-41
IADoc	8-47
IADocIterator	8-49
IADocText	8-50
Constants	8-51
Exceptions	8-51

Chapter 9	Storage Category	9-1
-----------	-------------------------	-----

General Storage Logic	9-3
HFS Implementation	9-5
Creating New Storage	9-5
Sample Code to Create Storage	9-5
Opening Existing Storage	9-6
Sample Code for Establishing Existing Storage	9-6
Allocating and Deallocating Blocks of Storage	9-6
Reading and Writing Storage	9-8
Reporting on Storage	9-8
Compacting Storage	9-8
Using the Mutex Facility	9-9
Cloning Store Streams	9-10
Creating Storage Subclasses	9-10
Creating a Storage Construction Utility	9-10
Creating a Subclass of IASToreStream	9-11
Creating a Subclass of IAMutex	9-15
Storage Class Category Reference	9-17
Header Files in the Storage Class Category	9-17
Class Specifications	9-18
HFSSoreStream	9-18
IAInputBlock	9-22
IAOutputBlock	9-25
IALock	9-28
IAMutex	9-28
IAStorage	9-30
IAStoreStream	9-39
Storage Class Utilities	9-45
Typedefs	9-48
Storage Exceptions and Error Handling	9-50

Chapter 10	Storable Category	10-1
	Understanding Storables and Ordered Storables	10-3
	Creating Subclasses	10-4
	Creating a Subclass of IASortable	10-4
	Creating a Subclass of IAOrderedStorable	10-6
	Creating a subclass of IAOrderedStorableSet	10-7
	Common Operations	10-7
	Creating an Ordered Storable Set	10-7
	Open an Existing Ordered Storable Set	10-8
	Updating an Existing Ordered Storable Set	10-8
	Sample Code for Updating an Ordered Storable Set	10-10
	Searching and Iterating through an Ordered Storable Set	10-11
	Storable Class Category Reference	10-13
	Header File	10-13
	Class Specifications	10-14
	IAOrderedStorable	10-14
	IAOrderedStorableIterator	10-17
	IAOrderedStorableSet	10-18
	IASortable	10-27
	Class Utilities	10-32
	Externs	10-33
	Exceptions and Error Handling	10-33
Appendix A	Alphabetical List of Functions	A-1

Figures, Tables and Listings

Figure 1-1	An inverted index	1-5
Figure 1-2	A vector index	1-6
Figure 1-3	Sample output of a simple query	1-7
Figure 1-4	An example of output for a query-by-example.	1-8
Figure 1-5	The use of tokenizer and filters	1-9
Figure 1-6	The RecipeSwap corpus	1-10
Figure 1-7	The layers of an information access application	1-11
Table 2-1	Class categories within IAT	2-3
Figure 2-1	Relationships of the index classes	2-5
Figure 2-2	Index inheritance	2-6
Figure 2-3	Accessor hierarchy	2-7
Figure 2-4	Relationships in a ranked search	2-8
Figure 2-5	Relationships between analysis and tokens	2-9
Figure 2-6	Provided tokenizer and filters	2-10
Figure 2-7	A SimpleAnalysis	2-11
Figure 2-8	Analysis subclasses created for RecipeSwap	2-12
Figure 2-9	Corpus abstract classes	2-13
Figure 2-10	The Macintosh HFS subclasses	2-14
Figure 2-11	The storable classes	2-15
Figure 2-12	Logical relationships between storage classes	2-16
Figure 3-1	The external interface of RecipeSwap	3-2
Figure 3-2	RecipeSwap persistent data	3-3
Figure 3-3	The related categories	3-4
Table 2-2	Association with Classes	3-5
Figure 3-4	Subtasks of recipe query	3-7
Figure 3-5	Interaction diagram for establishing a recipe index	3-9
Figure 3-6	Interaction diagram for building an accessor	3-10
Figure 3-7	Interaction diagram for a simple query	3-11
Figure 3-8	The subtasks of submit recipe	3-12
Figure 3-9	Interaction diagram for add recipe	3-13
Figure 3-10	Interaction diagram for creating a RankedQueryDoc	3-14
Figure 3-11	Interaction diagram of query by example	3-15
Figure 3-12	Interaction diagram for deleting recipes from the index	3-16
Figure 3-13	An interaction diagram for a complete update	3-18
Figure 3-14	Interaction diagram for initializing an index	3-19
Listing 4-1	Defining and using your own memory allocator.	4-6
Figure 5-1	An overview of an index	5-3
Figure 5-2	Index inheritance tree	5-4
Figure 5-3	An inverted index	5-5
Figure 5-4	A vector index	5-6
Table 4-1	Comparison of index types for time and space	5-7
Figure 5-5	Interaction diagram for index creation	5-9
Listing 5-1	Creating an index	5-9
Listing 5-2	Differences when creating an index with a named block	5-11
Figure 5-6	Interaction diagram for establishing an existing index	5-12

Listing 5-3	Establish an existing index	5-13
Listing 5-4	Determining type of index	5-14
Figure 5-7	Interaction diagram of an update to match the corpus	5-16
Listing 5-5	An example of updating an index to match its corpus	5-16
Figure 5-8	Interaction diagram for individual update	5-17
Listing 5-6	Updating individual documents	5-17
Figure 5-9	Interaction diagram for iterating through an index	5-19
Listing 5-7	Iterating through an index	5-19
Figure 5-10	Interaction diagram for a merge	5-21
Listing 5-8	Merge a source index to a destination index	5-21
Listing 5-9	Compact an index	5-22
Figure 5-11	Index hierarchy	5-32
Figure 5-12	Overview of the index relationships	5-33
Figure 5-13	Inverted index overview	5-47
Figure 5-14	Vector index overview	5-69
Figure 6-1	Accessor inheritance tree	6-4
Figure 6-2	Query logic	6-5
Figure 6-3	Interaction diagram to build an accessor	6-6
Listing 6-1	Build an inverted vector accessor	6-6
Listing 6-2	Report search progress	6-7
Figure 6-4	Output from a simple ranked query	6-8
Figure 6-5	Interaction diagram for a simple ranked query	6-8
Listing 6-3	Answer a simple ranked query	6-9
Listing 6-4	Display search results	6-10
Listing 6-5	Get and print a document name	6-10
Figure 6-6	Sample output from a query by example	6-11
Figure 6-7	Interaction diagram for creating a RankedQueryDoc	6-12
Listing 6-6	Find documents matching example document	6-13
Figure 6-8	Sample output from a Boolean query	6-14
Listing 6-7	Find documents satisfying Boolean expression	6-15
Figure 6-9	Sample output from describing a document	6-16
Listing 6-8	Find the words that best describe a document	6-17
Figure 6-10	Sample output from finding related words	6-18
Listing 6-9	Find the words related to a given wordt	6-19
Figure 6-11	Accessor inheritance	6-22
Figure 6-12	IAHit relationships	6-27
Figure 6-13	IAProgressReport relationships	6-30
Figure 6-14	RankedAccessor relationships	6-37
Figure 7-1	Class diagram of tokens and terms	7-4
Figure 7-2	A tokenizer	7-5
Figure 7-3	Token and filter classes	7-6
Figure 7-4	Illustration of sequential filtering	7-8
Figure 7-5	Analysis subclass	7-10
Listing 7-1	SimpleAnalysis subclass header	7-10
Listing 7-2	SimpleAnalysis subclass body	7-10
Listing 7-3	IAAnalysis subclass header	7-11
Listing 7-4	IAAnalysis subclass body	7-11
Listing 7-5	StopWordFilter header	7-12
Listing 7-6	StopWordFilter implementation of GetNextToken	7-13
Listing 7-7	DocTextCharStream header	7-14
Listing 7-8	DocTextCharStream body	7-14

Figure 7-6	AlphaTokenizer relationships	7-18
Listing 7-9	Sample implementation of filtered MakeTokenStream	7-27
Listing 7-10	Sample Implementation of GetNextToken for an IATokenFilter	7-35
Figure 8-1	Corpus relationships	8-3
Figure 8-2	HFS instantiation of corpus classes	8-4
Figure 8-3	Interaction diagram for iterating through a corpus	8-6
Listing 8-1	List text files	8-6
Listing 8-2	Establishing an existing corpus	8-7
Listing 8-3	Sample header file of an IACorpus subclass	8-8
Listing 8-4	Sample implementation of GetProtoDoc	8-9
Listing 8-5	Sample implementation of GetDocText	8-9
Listing 8-6	Sample header of an IADoc subclass	8-9
Listing 8-7	Sample Header for an IADocIterator subclass	8-11
Listing 8-8	Sample Implementation of GetNextDoc	8-11
Listing 8-9	Sample header of an IADocText subclass	8-12
Listing 8-10	Sample implementation of GetNextBuffer	8-12
Listing 8-11	Creating a custom corpus iterator—header file	8-13
Listing 8-12	IADocIterator subclass header	8-14
Listing 8-13	Corpus subclass body	8-14
Listing 8-14	IADocIterator subclass body	8-15
Listing 8-15	Using HFSIterator	8-34
Figure 8-4	IACorpus relationships	8-42
Figure 9-1	Logical storage classes	9-4
Listing 9-1	Constructing storage	9-5
Listing 9-2	Establish existing storage	9-6
Listing 9-3	Allocating a named block of storage	9-7
Listing 9-4	Opening a named block of storage	9-7
Listing 9-5	Report amount of space in storage	9-8
Figure 9-2	A sample result of compacting storage	9-9
Listing 9-6	A utility to construct storage	9-10
Listing 9-7	Sample header file of an IASToreStream subclass	9-11
Listing 9-8	Sample implementation of Clone()	9-13
Listing 9-9	Sample implementation of IsWritable()	9-13
Listing 9-10	Sample implementation of IsOpen()	9-13
Listing 9-11	Sample implementation of Initialize()	9-13
Listing 9-12	Sample implementation of Open()	9-14
Listing 9-13	Sample implementation of GetEof()	9-14
Listing 9-14	Sample implementation of SetEof()	9-14
Listing 9-15	Sample implementation of Write()	9-14
Listing 9-16	Sample implementation of Read()	9-15
Listing 9-17	Current implementation of IAMutex	9-16
Figure 9-3	IStorage relationships	9-31
Figure 10-1	Object storage structures	10-3
Listing 10-1	Sample header file for an IASTorable subclass	10-4
Listing 10-2	Sample Constructor	10-5
Listing 10-3	Sample Implementation of DeepCopy	10-5
Listing 10-4	Sample Implementation of Restore	10-5
Listing 10-5	Sample Implementation of StoreSize	10-6
Listing 10-6	Sample Implementation of Store	10-6
Listing 10-7	Sample Implementation of Equal	10-7

Listing 10-8	Sample Implementation of Less Than	10-7
Listing 10-9	Creating an IAOrderedStorableSet	10-8
Listing 10-10	Open an existing Ordered Storable Set	10-8
Listing 10-11	Adding a storable to an OrderedStorableSet	10-10
Listing 10-12	Updating additional data for an existing storable	10-10
Listing 10-13	Removing a storable from an OrderedStorableSet	10-11
Listing 10-14	Get an object by key	10-11
Listing 10-15	Make a sequential iterator	10-12
Listing 10-16	Make a positioned iterator	10-12
Figure 10-2	IAOrderedStorable relationships	10-15
Listing 10-17	Sample Implementation of Equal	10-16
Listing 10-18	Sample Implementation of LessThan	10-17
Listing 10-19	Sample Implementation of DeepCopy	10-28
Listing 10-20	Sample Implementation of Restore	10-29
Listing 10-21	Sample Implementation of StoreSize	10-30
Listing 10-22	Sample Implementation of Store	10-30
Listing 10-23	Sample Implementation of Deep Copy and Deep Copying	10-31
Listing 10-24	Sample Implementation of Restore and Restoring	10-32

Overview of this Manual

This manual is a combination of conceptual guide, programmer's guide, and reference for the Apple Information Access Toolkit.

As not all users will want all of this information, here is a guide to what is in the chapters.

Chapter 1, "Introduction to the Apple Information Access Toolkit," is a quick introduction to the toolkit. It emphasizes information access in general, and introduces an example application. This is the source for basic information access definitions. Everyone should read this one; it is short, but basic.

Chapter 2, "Overview of IAT Content," is the introduction to the tools of the toolkit. This is key to the rest of the organization of the manual, and is the first look inside the toolkit. Everyone should read this also.

Chapter 3, "Designing an Application," is a overview of the design of the application outlined in Chapter 1. It should be read by anyone doing application development with IAT.

Chapter 4, "Common Practices in IAT," documents those common classes used by many others within IAT such as memory allocation or exceptions.

The remaining chapters are the programmer's and reference guide to IAT. The should be referred to as needed while doing application development.

Each of these chapters has some introductory material, common procedures that are done with the classes, sample subclasses where they apply, and a reference to the classes and their members.

This manual is still in draft format. Please forward all corrections and comments to v-twin@apple.com.

P R E F A C E

Introduction to the Apple Information Access Toolkit

Some Possible Applications	3
RecipeSwap	3
How It Was Done	4
Indexing Facility	4
Search Facility	6
Analysis and Filtering	8
Storage and Document Type	9
Construction with IAT	10

Introduction to the Apple Information Access Toolkit

In this chapter we

- propose some sample applications that might be built using the IAT
- provide an overview of information access items required to do these tasks
- describe how IAT provides the features needed for strong information access application development.

The Apple Information Access Toolkit (IAT) is an object-oriented information access engine that provides required capabilities to index, search, and analyze large volumes of documents. The IAT, formerly known by its code name “V-Twin,” is a collection of tools which can be used separately or together to perform a variety of information access tasks.

Some Possible Applications

IAT is a flexible toolkit that may be used in many applications. This is one possible scenario that we’ve chosen to present some of its features.

RecipeSwap

Chef Irina Suflay has a very successful cookbook business going. She provides an on-line service to other gourmet cooks to distribute recipes. They must provide one new recipe per week (and a small fee...) to receive full access to her impressive database. This access allows them to search the database for certain recipes.

The application that allows Chef Suflay this success does these things:

- It automatically picks up e-mailed recipes, checks them against the database, files ones that are unique, and produces a report of those that may be duplicates of ones already on file.
- It allows a rapid search and delivery of recipes by natural language query. This search delivers the “top ten” recipes that fit the query closest.

For example, if Ira Goodcook sends in his favorite recipe for prune confit, Irina’s system receives the recipe, matches it against the stored e-mails to see if there might already be a recipe for prune confit, and indexes it if it is not likely to be a duplicate. If it might be (perhaps a recipe for prune loaf exists), a report is produced listing those likely candidates for duplication for Irina’s information. Irina can manually cause the system to accept the recipe if she decides it is not a duplicate.

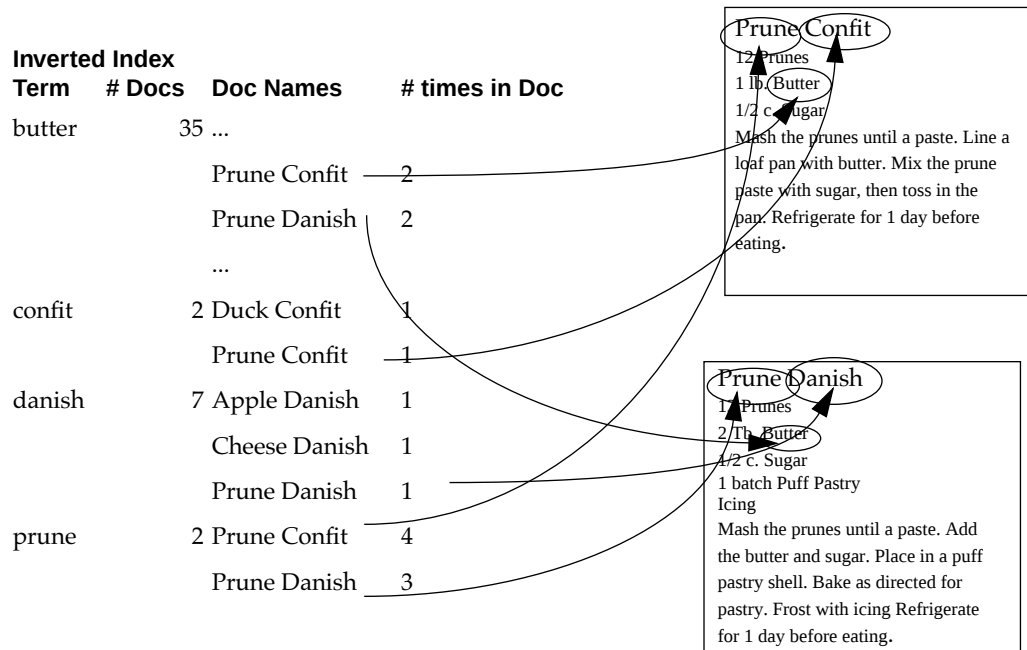
When Ira inquires to find a recipe that has kumquats, cilantro, and avocado (those being the best things located at the produce market that day), the system will list the top ten recipes most likely to be a match, and show which of those ingredients are found in the recipe. These recipes will be in priority order, that is, one containing all three ingredients and very little else will be at the top of the list; one containing only one among many other ingredients will be found at the bottom.

And, currently, this system does it all on a Macintosh sitting in Irina’s extra large pantry.

How It Was Done

Indexing Facility

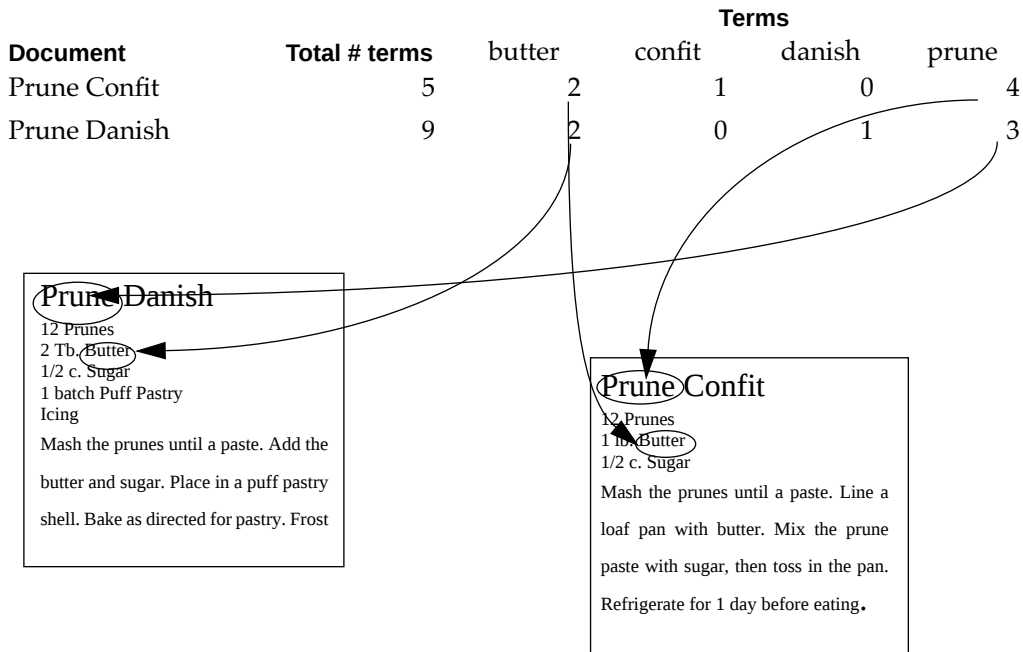
- * Applications of the RecipeSwap type require that indexes be made of a large set of documents. A **document** is a collection of data containing some text. Documents may be on different media, or in different formats. For example a word processing file, an e-mail message, or a database record may be documents. An **index** is a representation of the contents of a set of documents. Different types of indexes have different information about the set so different operations may be run on the index.
- * Indexes generally contain terms. A **term** is the basic unit of text that gets indexed. A term is typically a word, but may be a phrase or a modified form of a word.
- * There are different types of indexes. An **inverted index** is a table of all terms found in the collection, with pointers to which documents contain the term. An inverted index is similar to the index in the back of a book; rather than point to a page, it points the document containing the term. Figure 1-1 shows an example of an inverted index.

Figure 1-1 An inverted index

An inverted index makes it very easy to search for documents containing specific words. However, Irina wants to be able to search for recipes similar to a given recipe. A vector index can do this efficiently.



A **vector index** is a table of all documents stored in the collection which points to terms that are contained in each document. In order to determine if a document exists that is very similar, a vector index is used. Figure 1-2 shows an example of a vector index.

Figure 1-2 A vector index**Vector Index**

To allow more efficiency of time and space, IAT supports a single **inverted vector index** that combines the organizations found in both inverted and vector. As RecipeSwap has may uses for its index, it uses an inverted vector index.

Search Facility

The RecipeSwap system requires a ranked search facility for its queries. It promises a list of the top ten recipes, not just any recipes that match the query from the patron. This means the system must weigh the results of the search and know which documents are the best match for the query. A **ranked search** provides a score for the closeness of the match, which allows the system to list the search results from best to worst.

❖

❖

RecipeSwap system requires a search to find matches to a **simple query**. A simple query is a list of terms. The search provides a ranked list of hits, that is the document that contains at least one those terms, its score, and the terms it has. The patron requesting recipes would be prompted to put in a few terms. Then the top recipes matching those terms could be found. A request for prunes, butter, sugar would find all recipes containing any of these, but would only report those with the whose score were among the top ten, as seen in Figure 1-3. Recipes containing more than one of these terms,

containing these terms many times, or containing very few other terms than these terms, would get the highest scores.

Figure 1-3 Sample output of a simple query

Recipes Containing Prune, Butter, Sugar	
Score	Document Name
1.0	Prune Confit (prune, butter, sugar)
.84	Prune Danish (prune, butter, sugar)
.74	Amazing Prune Danish (prune, sugar)

* A ranked search is more powerful than a Boolean search, found in older information retrieval systems. A **Boolean search** requires the user to specify whether matches must contain all the query words or only any of the query words. The result is often a “feast or famine,” either a daunting unsorted list of too many items, or a sparse list of too few. A query written with find prunes OR butter OR sugar would find every recipe made with butter whether or not it had prunes. There would be an unranked long list. One written as find prunes AND butter AND sugar would not find “Amazing Prune Danish” if it used margarine rather than butter. (Note: The IAT does support Boolean search for those applications that require it.)

* There is another type of query in RecipeSwap. When a new recipe comes in, it is used as the query with a request to find a similar recipe. That is, the recipe itself is used as a source of terms, and the system is asked to locate the any documents that might be duplicates. This is a **query by example**, which will start a ranked search using all of the terms found in the sample document. These terms will be weighed by their frequency both in this document and in all the documents within the index. This allows the search to provide those documents which are most relevant, that is, most similar to the query document. It would not be useful just to get a list of any document that contains any term in the recipe, and nearly impossible to find a document containing exactly the same terms as in the recipe. The search provides a list of the closest documents which are scored for their closeness. This score provided is the **relevance factor**.

* This search would produce output that lists just those documents scored sufficiently high, as seen in Figure 1-4. Those which may have some hits, but are below a chosen threshold, would not appear.

Figure 1-4 An example of output for a query-by-example.

Possible Duplicates of Prune Danish	
Score	Document Name
1.0	Amazing Prune Danish
.90	Prune Puffs
.83	Apple-Prune Danish
.81	Prune Confit

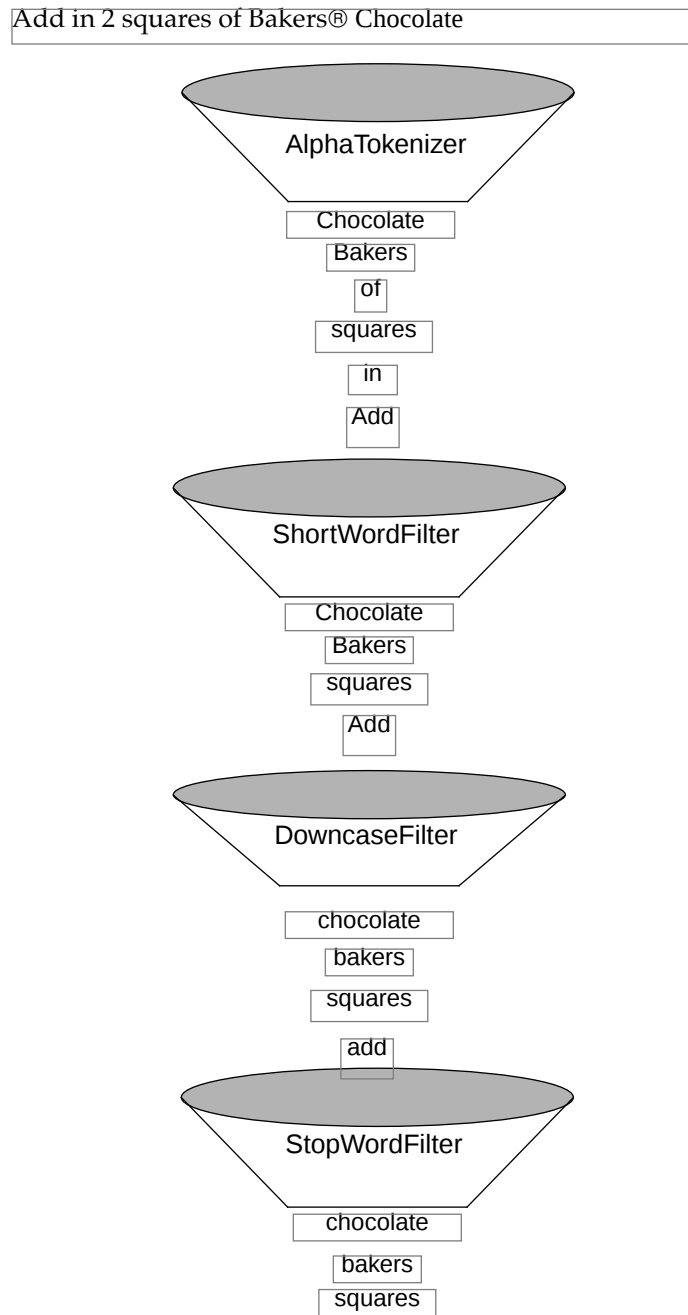
Analysis and Filtering

A system must be able to identify the terms in a document to build an index from it. This analysis of its content is done in several steps. First a document's characters are grouped into tokens. A **token** is a set of consecutive characters in a document which might be considered a term upon further analysis. The tool which converts a text stream into tokens is known as a **tokenizer**. An alphabetic tokenizer, for example, will take a text stream and gathers consecutive alphabetic characters into tokens. When it finds a number, space, or punctuation mark in the stream it ends the token it was building, discards the intervening non-alphabetic characters, and starts the next token when it finds an alphabetic character.

Indexes are more useful if the analysis filters tokens. A **filter** removes or alters tokens based on certain rules. For example, the RecipeSwap index should not think that Prune, with a leading upper-case character, and prune, all lower-case, are different terms. A downcase filter will convert any upper-case character in a token to a lower case character.

IAT provides the ability for a developer to build and include specialized application filters. This allows the facility for smarter queries. One of these might be a **stop word filter**, which discards terms that are found in a stop list. Stop words are typically common words that do not add to the meaning of a document such as "the," but might be terms that are not useful in a specific application. Few patrons would be interested in locating recipes that contain the term "cup" in common. Irina needs the ability to successively create a **stop list**, or those terms to be excluded by the stop word filter, of words that should not be considered when comparing recipes. When the documents are indexed, these words would not be taken into consideration.

Figure 1-5 shows the effects of the use of a tokenizer and successive filters on a phrase from a recipe. The ShortWord Filter removes tokens that are under three characters. The Downcase Filter turns all characters to lower case.

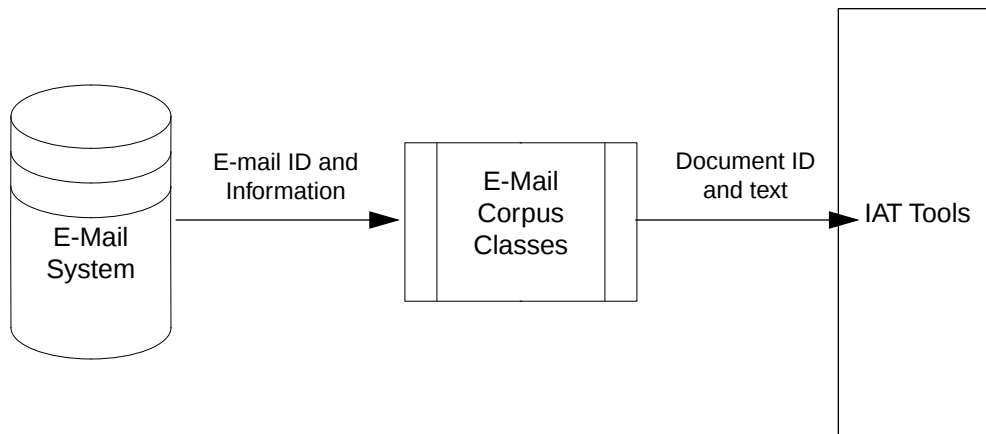
Figure 1-5 The use of tokenizer and filters

Storage and Document Type

IAT uses the power of object-oriented design to keep the storage media and document type separate from the indexing and analysis logic.

- * Indexes must be maintained in persistent **storage**, such as a hard disk. IAT builds its own tools, or classes, for the logical storage of indexes. Developers may create sub-classes of these classes to work optimally on their media, including cross-platform support.
- * IAT allows the developer to build an interface from the actual document to the logical document used to build the indexes. Each index maintains a separate **corpus**, or body of documents. This corpus does not contain the actual documents, but maintains a directory of them. It can then access these documents and locate the text within them much as a directory points to the actual files in a file system. A facility within the corpus accesses the document and provides text streams for analysis. Because of this, the RecipeSwap corpus, seen in Figure 1-6, needs only to provide a means of locating the e-mail messages and translating them to a text stream for analysis. IAT remains independent of the document type from there on.

Figure 1-6 The RecipeSwap corpus



IAT is written in ANSI C++ for compatibility with a variety of development and target environments.

Construction with IAT

The classes of the IAT toolkit constitute the core of an application. IAT contains base classes to do the storage, analysis, etc. required for information access applications. Many applications can be developed by adding little more than a GUI to the toolkit.

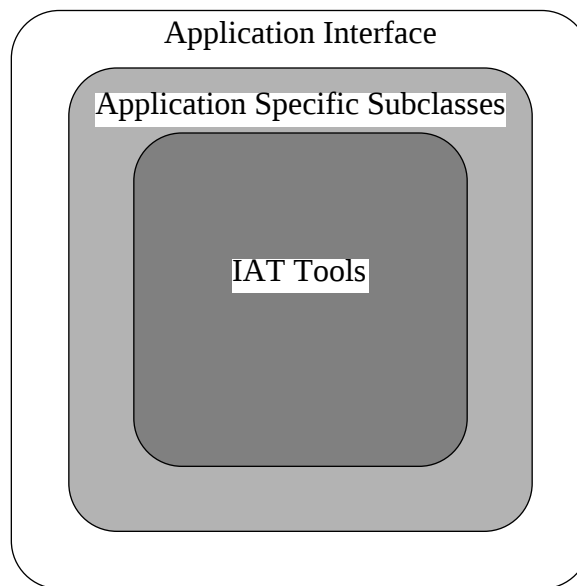
The power of object-oriented design, however, allows developers to modify the behavior of the IAT classes by creating subclasses. For example, IAT provides a class to do a simple analysis. RecipeSwap requires a slightly smarter analysis; it must not include common terms that do not distinguish recipes in its index. The developer can create a

Introduction to the Apple Information Access Toolkit

subclass of the abstract filter class included in the toolkit, and just add the application specific code. There is no need to “alter,” and possibly impair, the provided classes. There is little need to duplicate logic already present in those classes.

A typical application will have three layers as seen in Figure 1-7. The core of the application will be the classes provided by IAT. The developer will develop specialized subclasses where required for his application in a second layer. The application itself will provide the user interface to the system and add the procedural structure for using the toolkit classes and functions.

Figure 1-7 The layers of an information access application



Introduction to the Apple Information Access Toolkit

Overview of IAT Content

Facilities of IAT by Category	3
Index	4
Accessor	6
Analysis	8
Corpus	12
Storable	14
Storage	15

Overview of IAT Content

In this chapter we

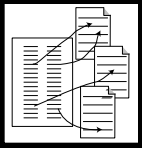
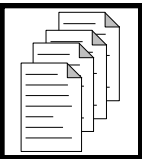
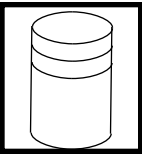
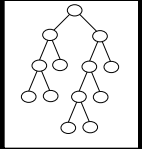
- review the major categories, or related areas of tools, in IAT
- show the major relationships between classes in those categories
- discuss the possible subclasses that might be built for applications.

This chapter is an overview. Please see the later detailed chapters for more information on each category.

Facilities of IAT by Category

The tools within the IAT are organized by category. Each category contains related classes for an area of Information Access.

Table 2-1 Class categories within IAT

	Index	Mapping between documents and terms. The construction and maintenance of indexes.
	Accessor	Use (usually search) of indexes.
	Analysis	Transform input text to index terms.
	Corpus	Definition of the set of documents; the means of obtaining text from the documents
	Storage	Management of persistent storage, the storage of indexes.
	Storable	Organization of persistent data. The data structure of stored data.

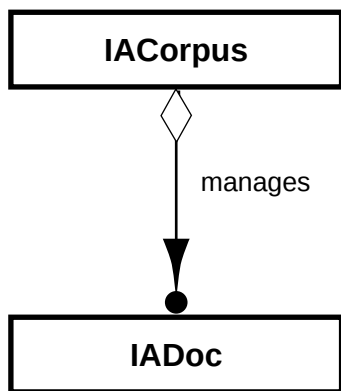
Each of the categories contains classes that provides base functionality. Many of the classes can be used as a base class for subclasses which can provide additional, application specific, functionality. The developer will generally have to add a control and user interface framework to use the tools provided. This section describes each of the class categories, some possible subclasses (many others not mentioned are possible), and

the specific subclasses required for to implement the sample applications described in the previous chapter.

The Class Diagram Notation

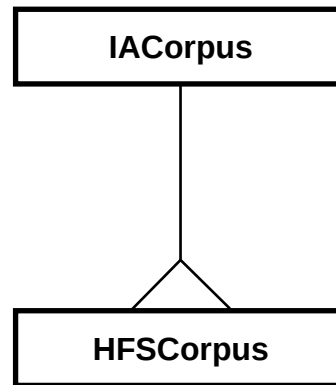
The class diagrams used in this chapter and throughout the chapter are based on a modified Object Modeling Technique (OMT) notation as used in the book Gamma, Erich et al. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Company, Inc. Reading, Massachusetts, 1995. More can be found on the OMT notation in Rumbaugh, James, et al. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

The arrowhead represents a relationship between the two classes.



The relationship is labeled from the perspective of the client class at the source of the arrow. A diamond at the base of the relationship indicates an aggregate relationship. This means the client “contains” the class, that is, the class is a part of the client. A dark circle at the end means that more than one of the client type may be instantiated by a single source object.

A triangle indicates inheritance.



If the class name is in italics, such as *IACorpus*, it is an abstract class and may not be instantiated. If it is in regular font, such as *HFSCorpus*, it is an instantiable subclass.

Index

The index category contains the classes required for the creation of an inverted index, a vector index, and the combination of the two. This class locates the document text through the corpus classes, extracts terms with the analysis classes, and builds the index. This index is persistently stored using the storage classes.

Figure 2-1 shows the relationships between the major classes. An *IAIndex* points to one *IAAnalysis*, which it uses to extract terms, one *IACorpus*, which is used to locate documents and get text, and one *IAStorage*, where any information that must persist (this includes the index, its corpus and its analysis) resides.

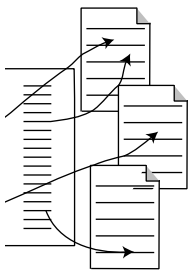
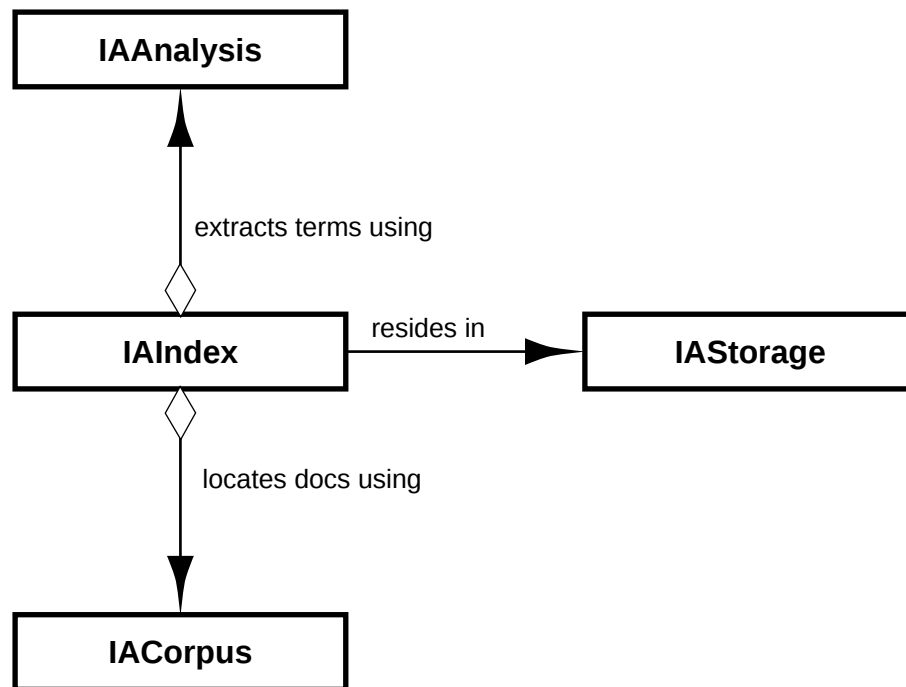
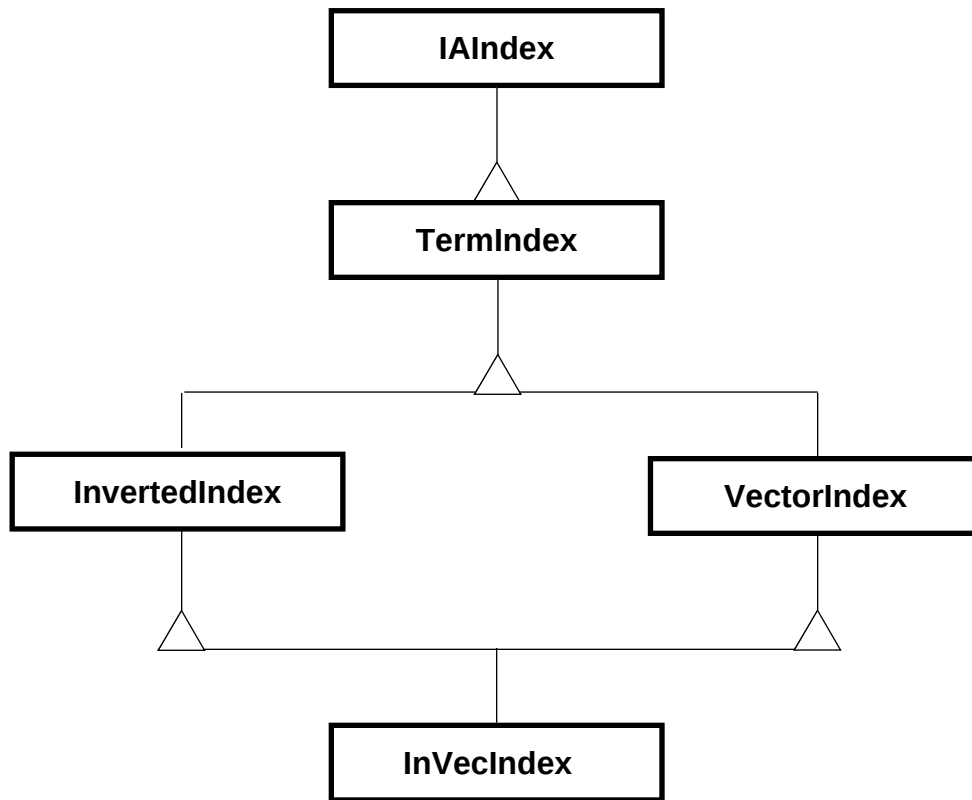


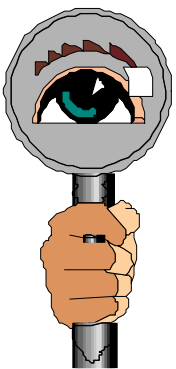
Figure 2-1 Relationships of the index classes

Possible Subclasses

Figure 2-2 shows the inheritance diagram for the index classes. Although TermIndex can be instantiated, it is unlikely an application would want to. It is used as the base class for InvertedIndex and VectorIndex.

Figure 2-2 Index inheritance

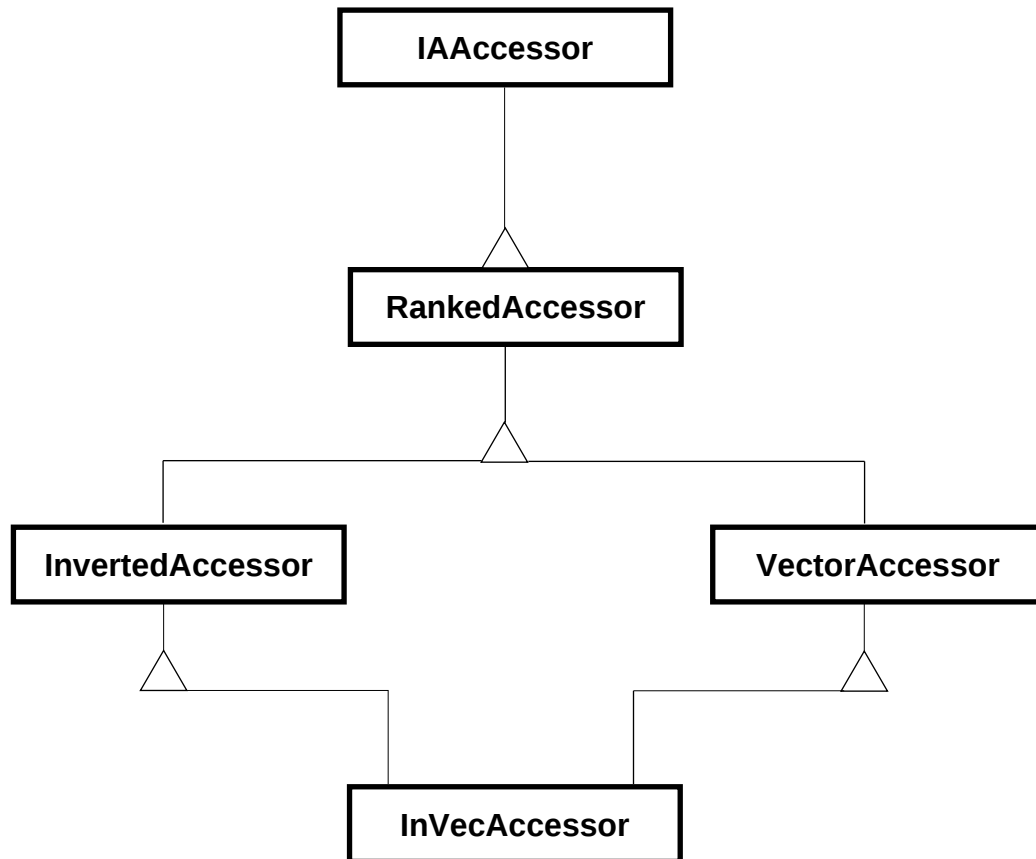
Most applications, including the RecipeSwap example, will not need to subclass an index. For more information about this category, please see Chapter 5, “Index Category.”



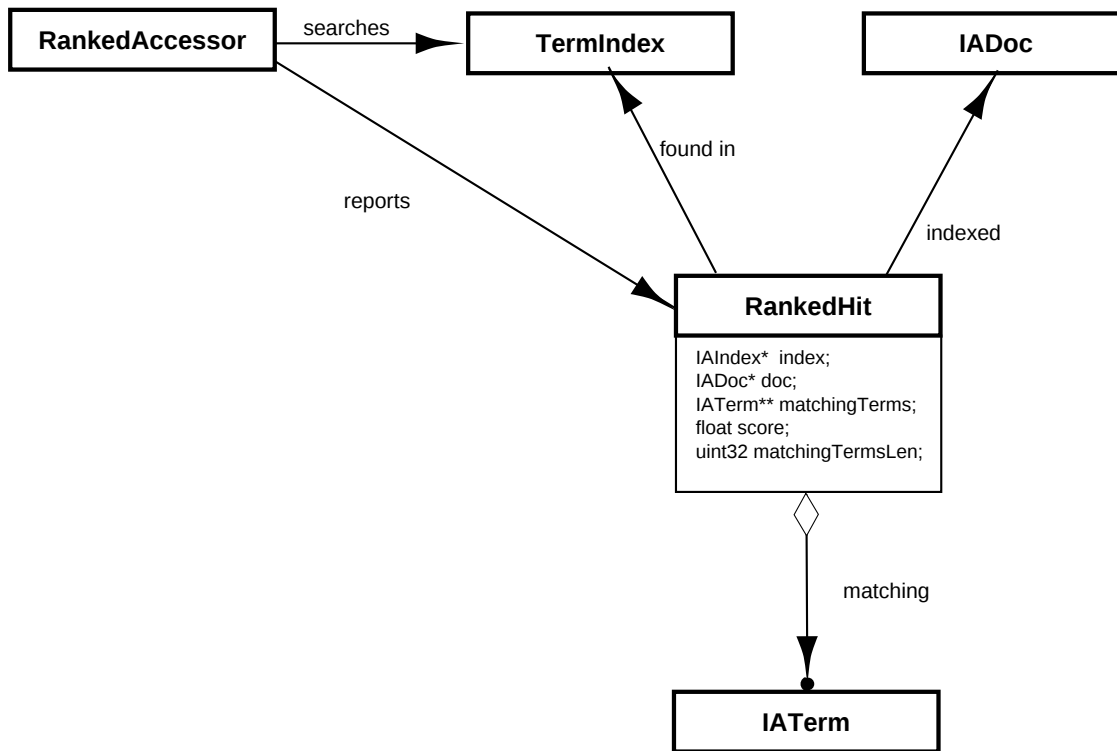
Accessor

The accessor classes allow for the search and comparison of indexes.

There is a parallel inheritance structure for accessor classes (shown in Figure 2-3).

Figure 2-3 Accessor hierarchy

The RankedAccessor provides a Ranked Hit for every document that contains terms sought in a search. As seen in Figure 2-4, this RankedHit identifies the document, the index containing the document, and a list of terms found in the document. It also provides a score indicating how relevant this document is to the query.

Figure 2-4 Relationships in a ranked search

Possible Subclasses

Like the Index category, the Accessor category contains a class, `RankedAccessor`, that is the parent of the `InvertedAccessor` and `VectorAccessor`. Developers will only create an accessor subclass if they create a different index.

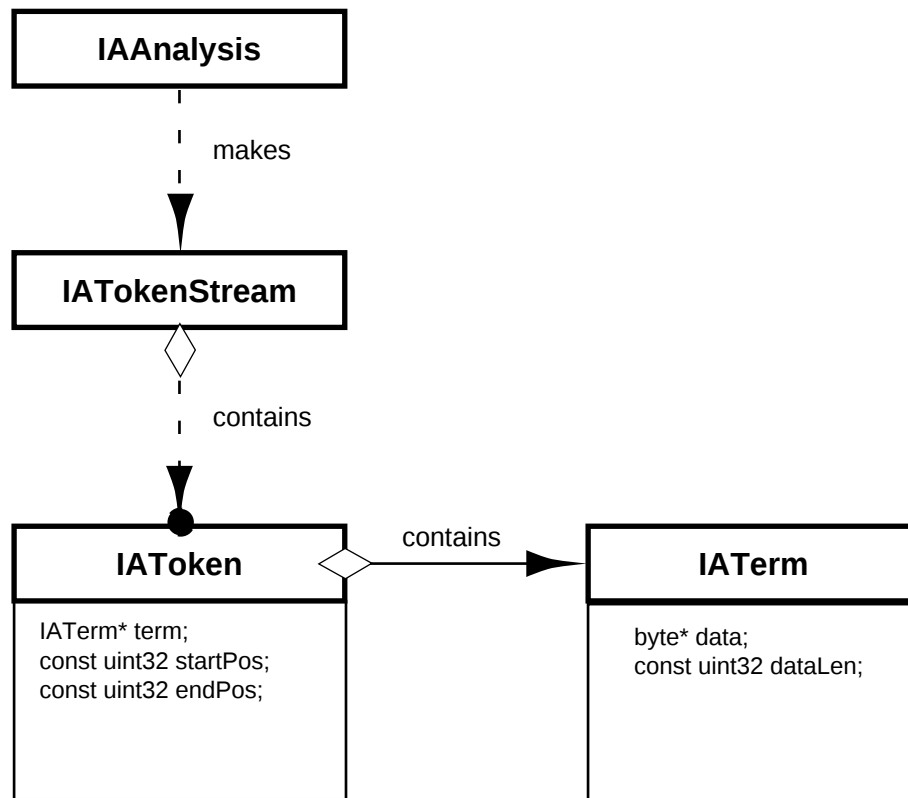
No subclasses of accessor were required in the `RecipeSwap` examples above.

For more detail on this class category, please see Chapter 6, “Accessor Category.”

Analysis

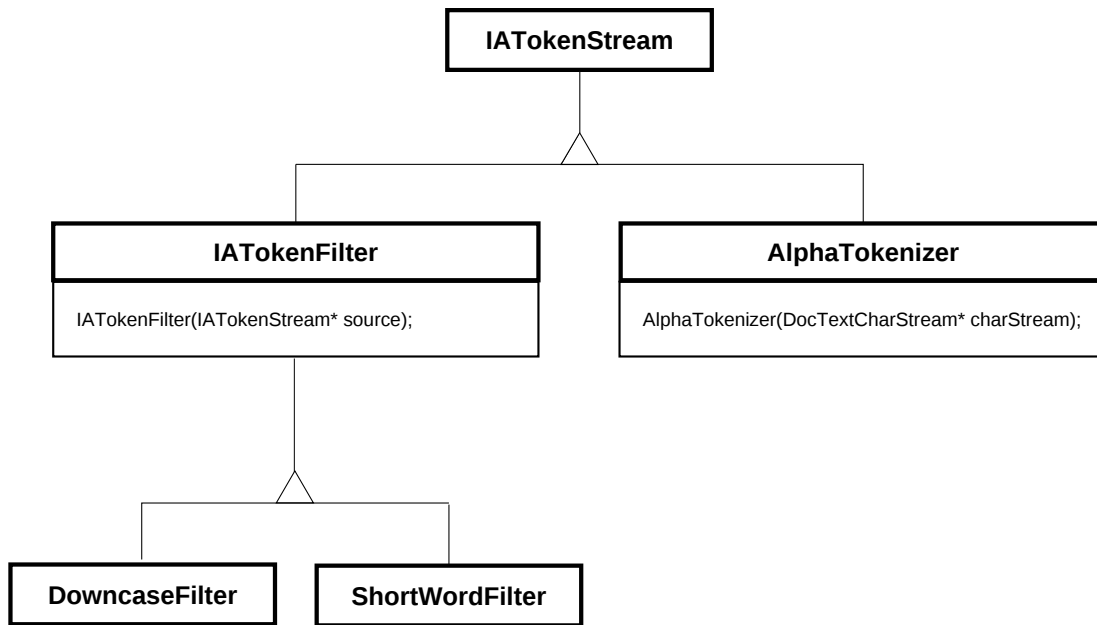
The analysis category provides the tools for locating terms in a text stream. Figure 2-5 shows that the abstract class, `IAAnalysis` produces a token stream, `IATokenStream`. This stream contains tokens, which are terms and the position of the term in the input text stream.



Figure 2-5 Relationships between analysis and tokens

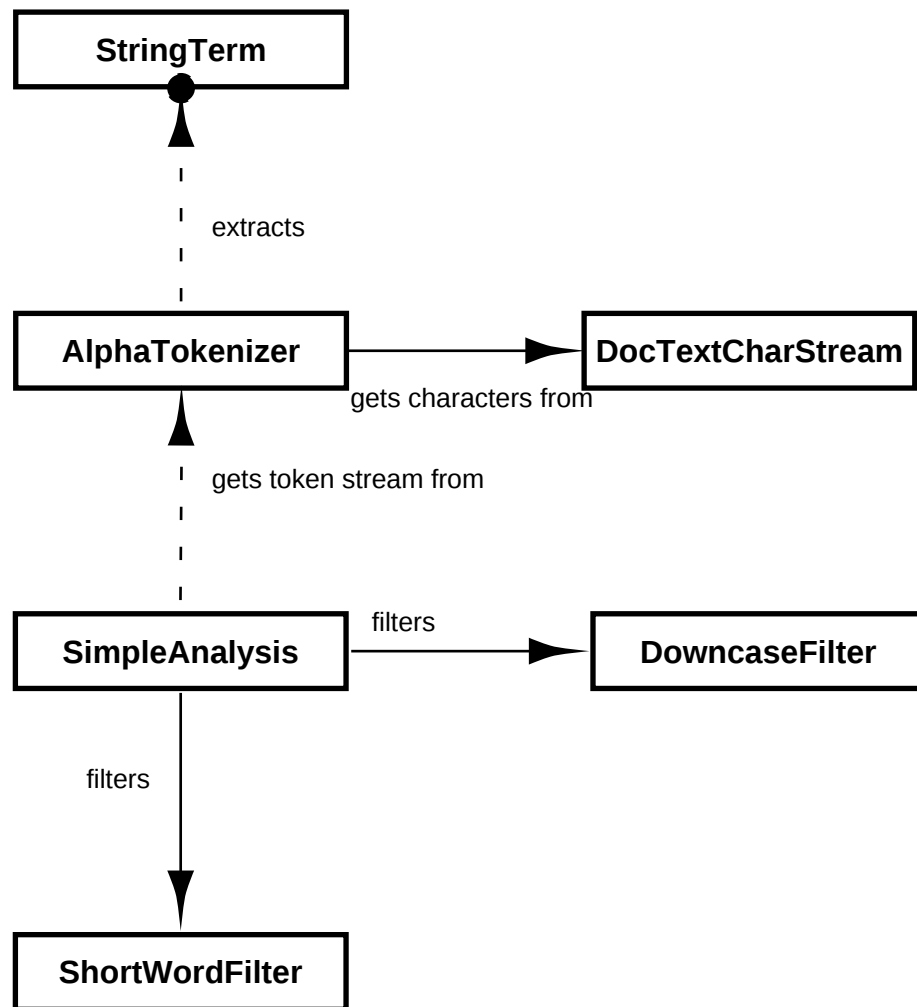
One implementation of **ITokenStream** must be a tokenizer; that is, the tool that takes a stream of characters and provides tokens. A specialized subtype of a **ITokenStream** is an **IATokenFilter**; that is, a tool which takes in a token stream, alters the stream in some fashion, and provides a new token stream.

IAT provides an example set of analysis subclasses. Figure 2-6 shows the tokenizer and filters provided.

Figure 2-6 Provided tokenizer and filters

AlphaTokenizer removes all blanks, punctuation and other special characters, and numbers from the input text stream. It provides StringTerms, which is a subclass of IATerm that implements terms as characters. The short word filter eliminates tokens shorter than a certain length and the lowercase filter converts all tokens to lower case letters.

This category provides SimpleAnalysis, a subclass of IAAAnalysis, which uses AlphaTokenizer, DowncaseFilter, and ShortWordFilter. Figure 2-7 shows the analysis implementation provided with IAT.

Figure 2-7 A SimpleAnalysis

DocTextCharStream is a utility which locates characters in an **IADocText**.

Possible Subclasses

The developer may wish to develop a specialized tokenizer that accepts a custom text stream, or tokenizers for other languages requiring different logic for locating tokens.

The developer may wish to create a subclass of **IATokenFilter** to provide specialized filters such as stemmers (those which shorten words to the root) or stop lists (lists of terms not to be included in the index).

There are abstract classes **IAToken** and **IATerm** which may require creation of subclasses if a non-textual language is chosen.

Overview of IAT Content

New analysis subclasses will be required whenever different filters or tokenizers are used than the combination provided in SimpleAnalysis. RecipeSwap, for example, will require a new filter, StopWordFilter, and a new analysis subclass, StopWordAnalysis.

Figure 2-8 Analysis subclasses created for RecipeSwap

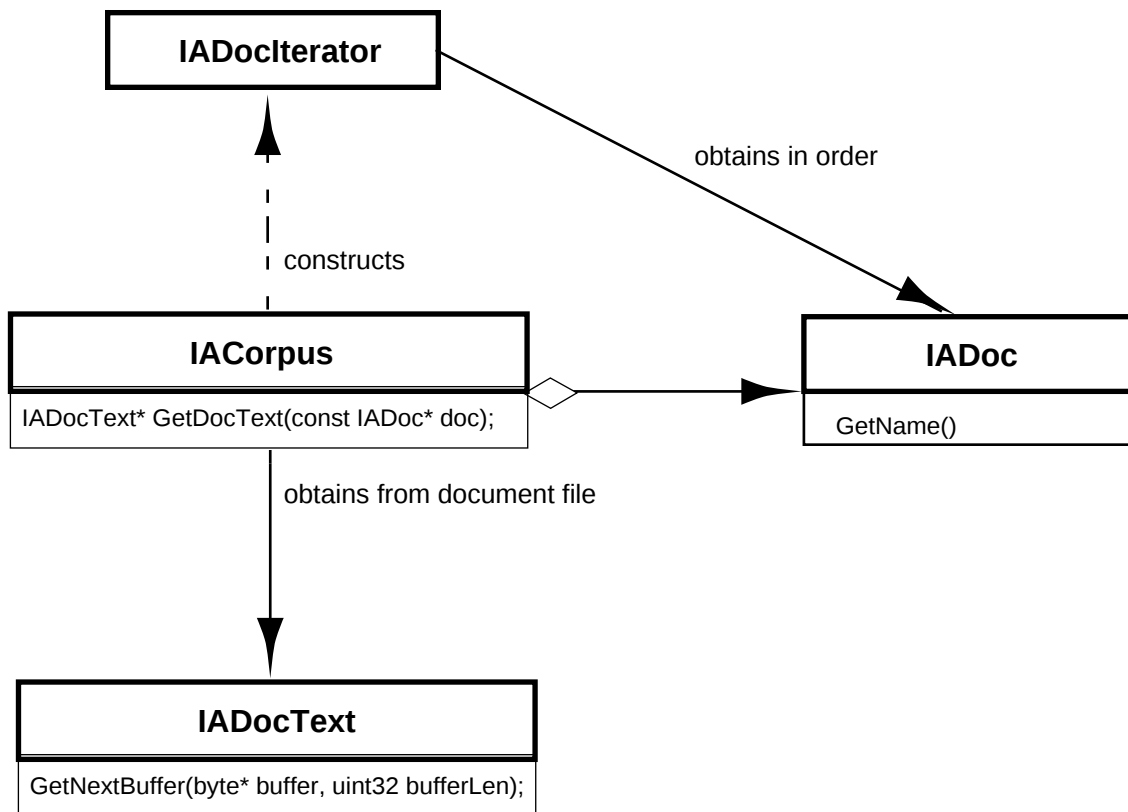


For more information on the analysis category, please see Chapter 7, “Analysis Category.”

Corpus

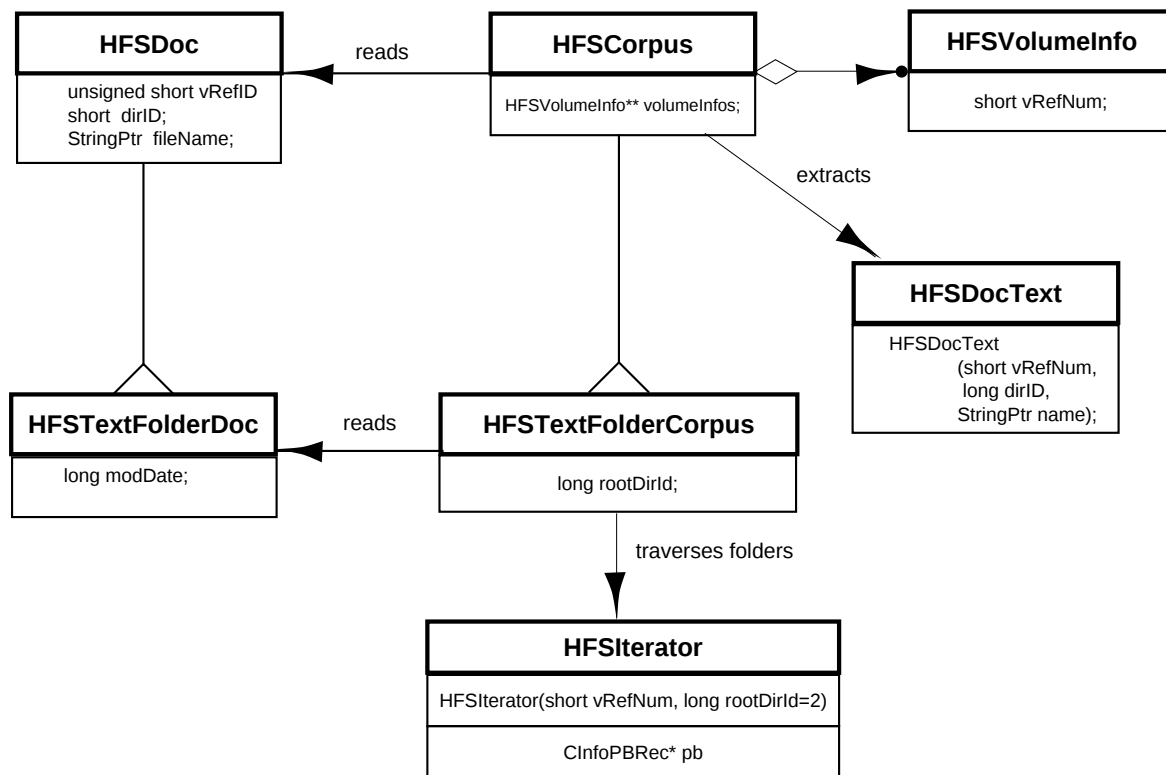
The IAT index is written to work with a set of logical documents. It is the job of the corpus classes to keep track of this set as it exists physically, and provide the text from the documents in a consistent logical format. The corpus is the interface between the IAT index and the actual items being indexed. This allows those items to be in a variety of formats, such as text files in a Macintosh HFS folder, e-mail messages in a database, or even subsets of text from a single physical document.

The logical document, characterized by the class IADoc, is similar to a directory entry: it contains the location of the document and pertinent attributes, not the document itself. When the document content is needed, it is obtained through the class IADocText. Figure 2-9 shows the relationships between the abstract classes.

Figure 2-9 Corpus abstract classes

The classes of the corpus category provide the facility to locate the documents, detect which have changed, which are new, or which have been deleted. The corpus maintains an iterator which can provide a list of the documents used for a particular index.

This category currently contains implementations for HFS text files, and an iterator to locate all text files in an HFS Folder. These are shown in Figure 2-10.

Figure 2-10 The Macintosh HFS subclasses

Possible Subclasses

Any other type of document or storage medium besides Macintosh HFS will require a subclass of IACorpus.

RecipeSwap, for example, would require an e-mail Corpus to locate the e-mail message body within the e-mail mailboxes. This will include subclasses of IACorpus, IADoc, and IADocText.

For more information, please see Chapter 8, "Corpus Category."

Storable

The storable classes provide a data structure mechanism to allow the organization, and access of very large sets of objects that must be quickly accessed from persistent storage.

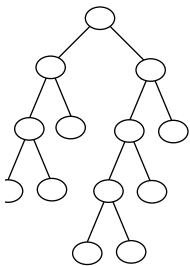
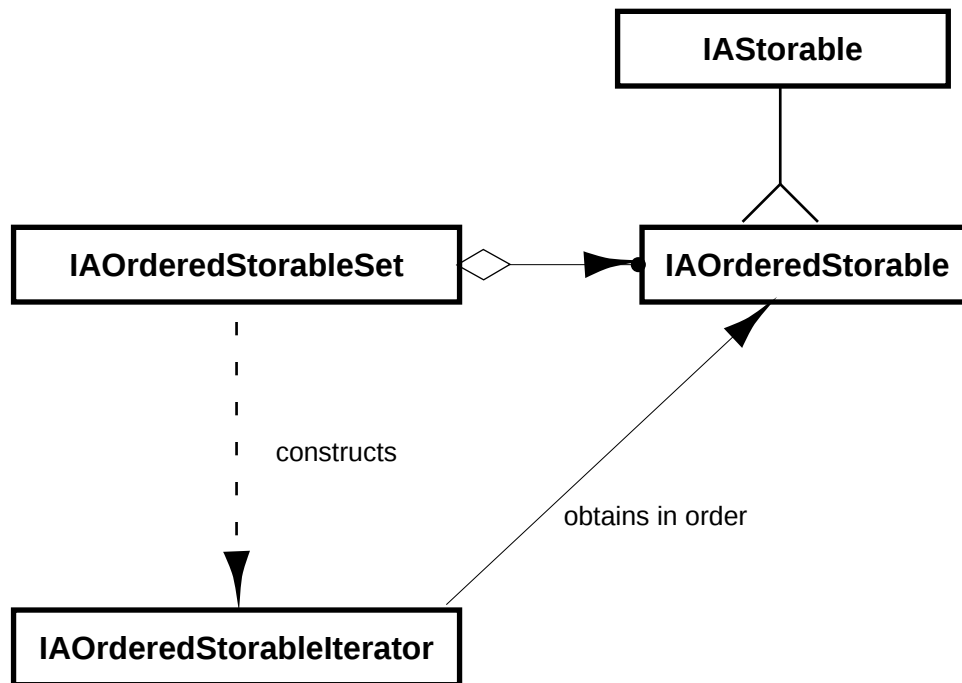


Figure 2-11 The storable classes

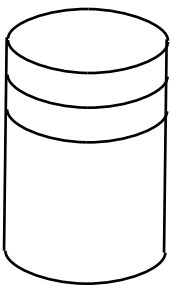
An **IASortable** is the supertype of any object that may be stored in persistent data. It provides for the access of storage. An **IAOrderedStorable** is a storable that can be uniquely identified; any subclass of **IAOrderedStorable** will contain one or more data members that can be used as a key, or unique identifier, for the item.

An **IAOrderedStorableSet** is the structure of **IAOrderedStorable**s. It allows for the update of the collection, and creates an **IAOrderedStorableIterator**, which obtains the stored objects in sequence.

Possible Subclasses

These classes are mainly used internally in IAT. For example, **IATerm** and **IADoc** are both **IAOrderedStorable**s. Applications may wish to use the facility, however, to create their own stored objects. To do so, a subclass must be created of **IAOrderedStorable**. **IAOrderedStorableSet** and **IAOrderedStorableIterator** do not require subclasses; they will work with any **IAOrderedStorable** subclass.

For more information, please see Chapter 10, "Storable Category."



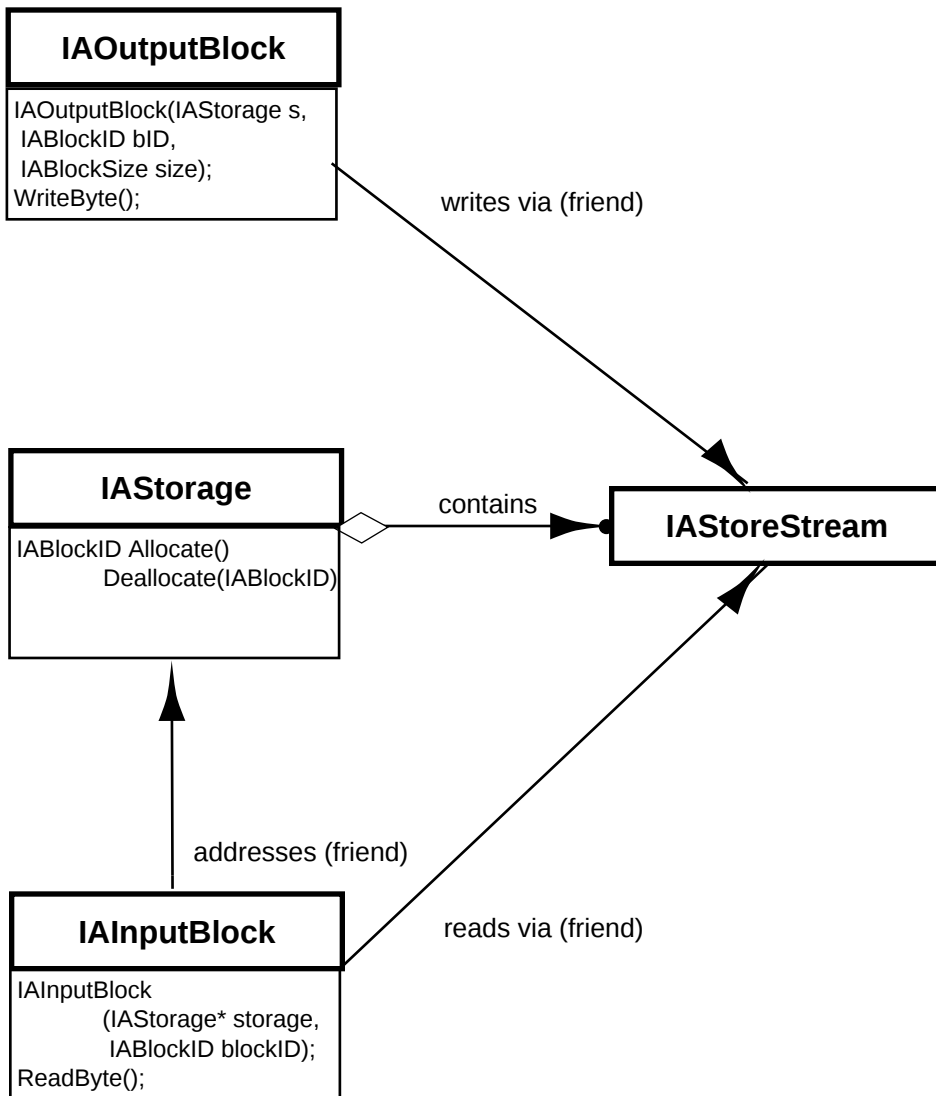
Storage

Storage contains those classes which allow for the access and creation of persistent files on storage media. IAT contains its own logical storage system that maintains blocks of

Overview of IAT Content

storage. Figure 2-12 shows the principal storage classes. IASTorage serves as the manager of the blocks; it allocates and deallocates them and maintains a table of contents of the blocks. IASTorage also creates the IASToreStream. IASToreStream does the actual I/O to the storage medium.

Figure 2-12 Logical relationships between storage classes



IAT provides an implementation of storage for Macintosh HFS. No HFSStorage class is needed; a utility, MakeHFSStorage, creates an HFSStoreStream and invokes the constructor for IASTorage with that HFSStoreStream. HFSStoreStream provides the access to the Macintosh Tool Box to open, read, and write the files.

Overview of IAT Content

Possible Subclasses

The developer may wish to subclass `IStoreStream` to allow for storage on other media or platforms than Mac HFS.

For more information, please see Chapter 9, “Storage Category.”

Overview of IAT Content

Designing an Application

Determining High Level Requirements	2
Determining the External Interfaces	2
Mapping to IAT Classes	3
Internal Task Design	5
Recipe Query	6
Submit Recipe	11
Duplicate Recipe	15
Stop Word Maintenance	16
Database Creation	18

Designing an Application

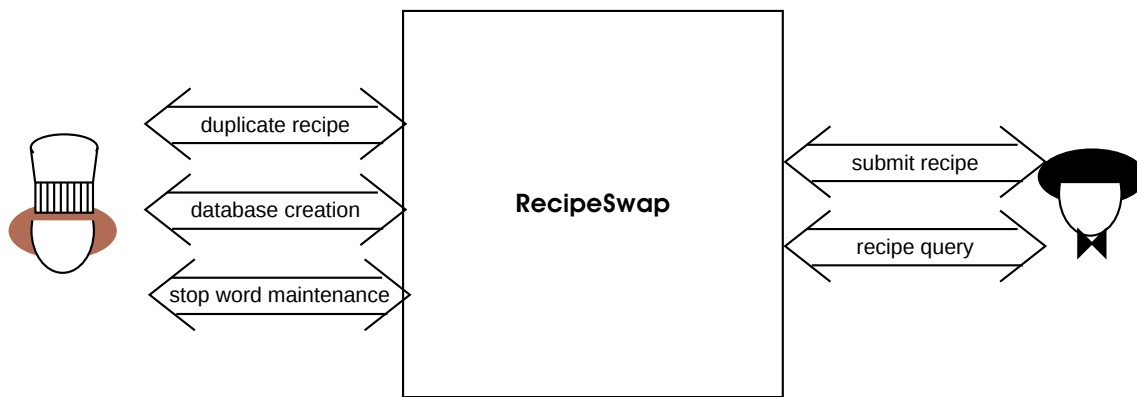
This chapter discusses the required application design for the RecipeSwap scenario outlined in the introduction. The focus is on choosing subclasses and the design of the controlling programs.

Determining High Level Requirements

Determining the External Interfaces

The application will provide the external interface of the system.

Figure 3-1 The external interface of RecipeSwap



Each of the dialogs illustrated in Figure 3-1 will become tasks.

Duplicate recipe: if the recipe submitted by a patron seems to be too close to another already on the database, Irina is notified. She can decide whether to delete the submitted recipe, or to keep it.

Database creation: the database must be initially created and stored. Chef Irina will decide upon its name and place.

Stop word maintenance: Chef Irina decides which terms to add to the stop word list to prevent them being used in indexing the recipes.

Submit recipe: patrons must submit at least one recipe per week. These come in by e-mail, and are added to the index. Those which are close to other recipes on the database may be flagged as possible duplicates, initiating a duplicate recipe dialog with Irina.

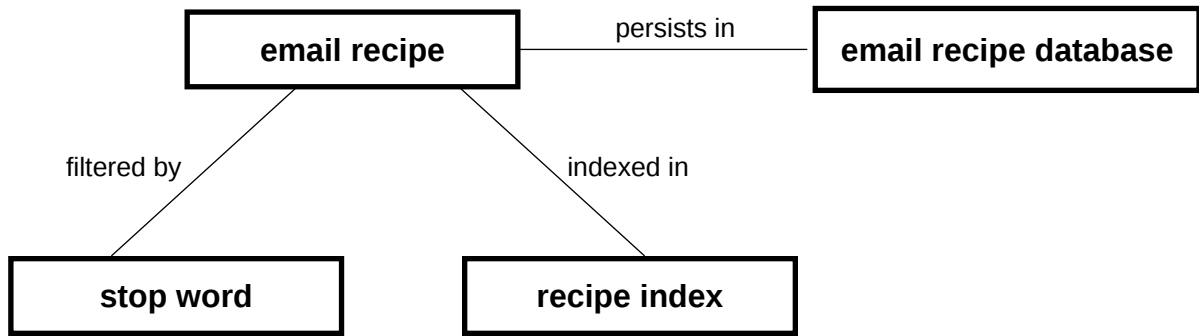
Recipe query: patrons may ask for the ten closest recipes containing the terms in the query.

The actual design of the dialogs is not directly connected with IAT, so we won't discuss it further. Our work is inside the box: the interface with IAT.

Mapping to IAT Classes

One way to begin determining the IAT classes required is to examine the persistent data required for the tasks. This data will most likely become stored objects.

Figure 3-2 RecipeSwap persistent data



Each of these can be assigned to the proper IAT category, then the category can be researched to see the closest IAT match.

The index is clearly an Index; stop word is a part of Analysis. The remaining data types are the storage of the recipes themselves. This is the corpus; the IAT classes will have to interface with this stored object.

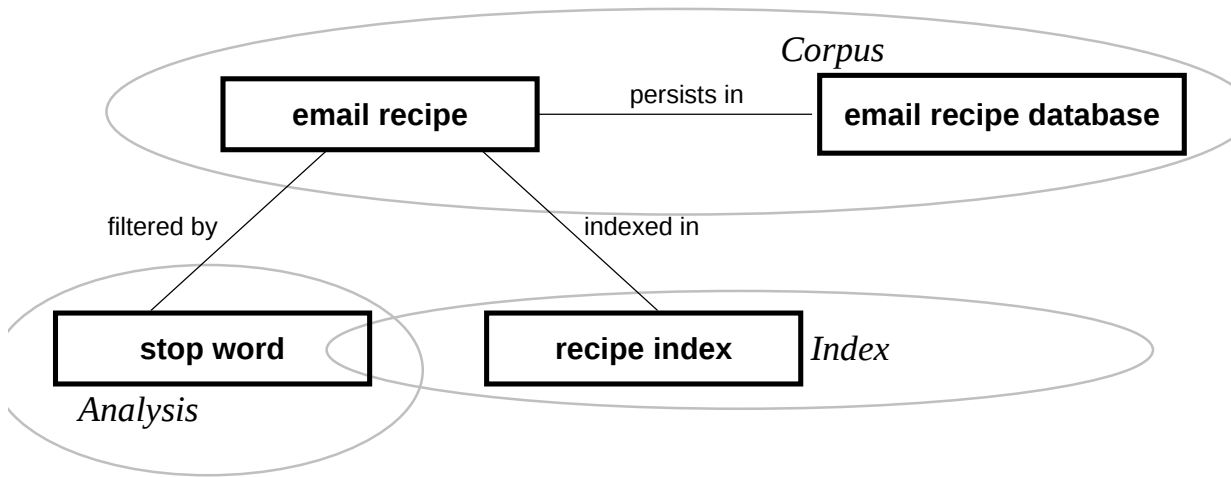
Figure 3-3 The related categories

Table 2-2 Association with Classes

Object	Associated Classes	Modification Required	New Subclass
e-mail recipe	IADoc	Specific subclass for e-mail body and related fields to locate this within folder. EmailDoc.	EmailDoc
	IACorpus	Specific subclass for the e-mail. Relates the doc to the text. EmailCorpus	EmailCorpus
	IADocText	Specific subclass to implement the location of text within an e-mail body. EmailDocText.	EmailDocText
	IADocIterator	Specific subclass to locate specific e-mails within the e-mail corpus, and to provide them in sequential order. EmailIterator.	EmailIterator
recipe index	InVecIndex	Most powerful index. Query speed important and must have ability to do similarity checking. No subclass required.	(none required)
	InVecAccessor	The accessor for an InVecIndex. No subclass required.	(none required)
	HFSSStorage	The required class to store the index; the existing IAT implementation of Macintosh storage is sufficient. No modification required.	(none required)
stop word	IATokenFilter	A filter to be used to eliminate terms. Subclass required; IAT provides no specific filter. StopWordFilter	StopWordFilter
	IAAnalysis	An analysis that is able to apply the stop word filter. Subclass required. StopWordAnalysis.	StopWordAnalysis

For information on how to create the corpus subclasses, please see “Creating Corpus Subclasses” beginning on page 8-7.

For information on how to create the analysis subclasses, please see “Creating Analysis Subclasses” beginning on page 7-9.

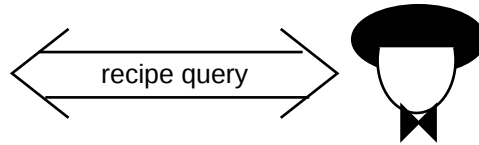
Internal Task Design

Each of the dialogs listed above is associated with a task. This section will break down each task into subtasks for clarity, and show the interaction with IAT objects.

Designing an Application

The subtasks are likely to reside in the same program; the decomposition is for clarity.

Recipe Query

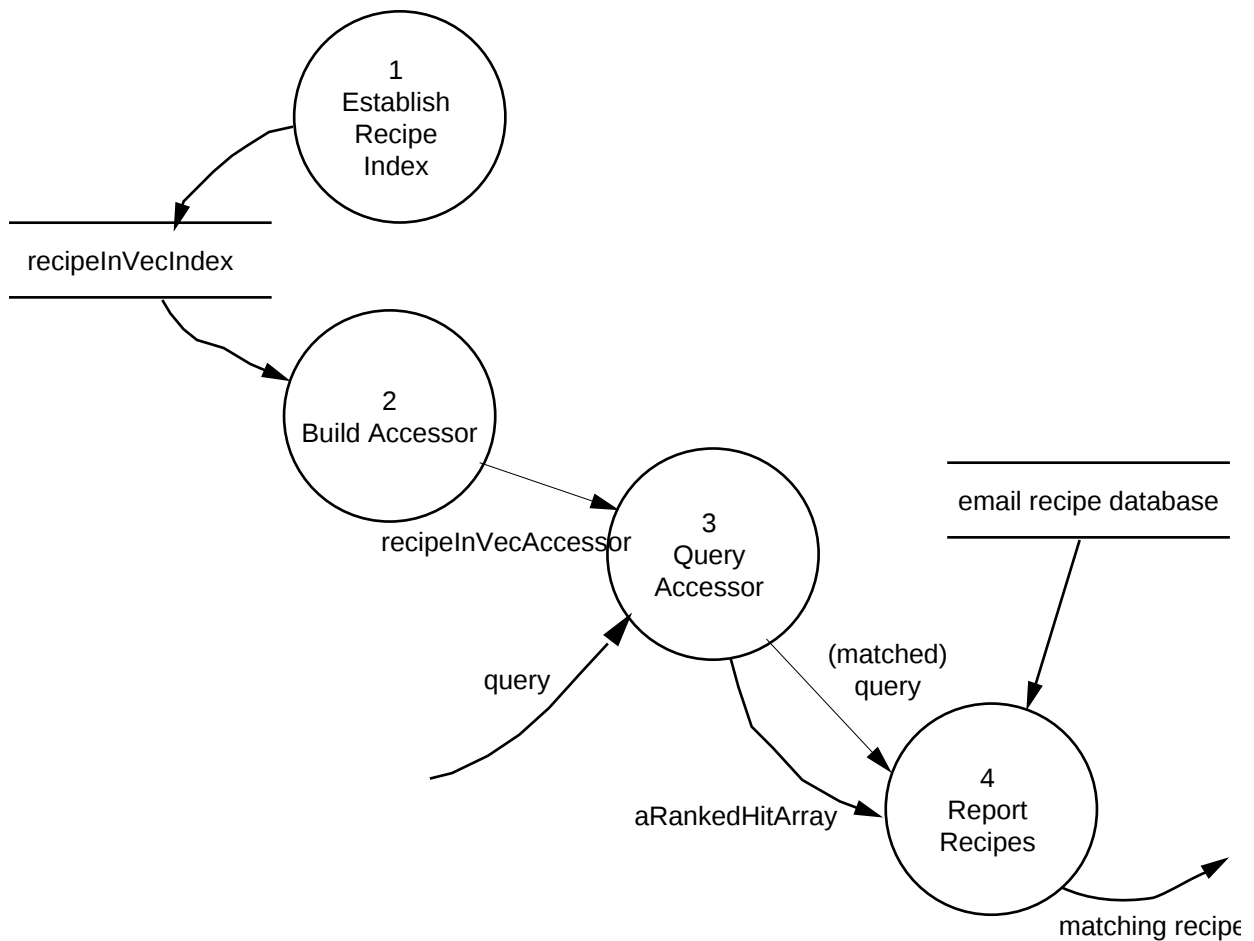


Description

The patron will submit a query as a string of terms. The application will search the recipes in the database, and provide those ten recipes that are the closest match to the terms.

Subtasks

Recipe query is a simple query. The application must construct a dialog with the patron to get a simple text query. Figure 3-4 shows the subtasks of this query.

Figure 3-4 Subtasks of recipe query

Establish recipe index

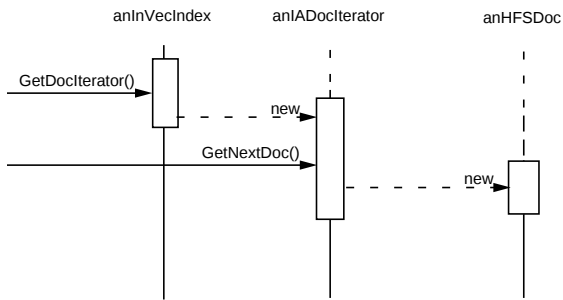
The recipe index is presumed to be existing. It must be located in storage and opened. It is opened as read only as no update is involved. See “Establishing an Existing Index” beginning on page 5-11 for a generic reference and sample code.

The new index object must be created with the same corpus type and analysis type as those with which the recipe index was originally created. The emRecipeDB is the e-mail Recipe Database folder. Opening the index allows the existing index to be read from storage. Figure 3-5 shows which objects and operations will be used to establish the recipe index.

The Interaction Diagram Notation

Figure 3-5 and other similar diagrams used in this manual are based on a modified interaction diagram notation as used in the book Gamma, Erich et al. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Company, Inc. Reading, Massachusetts, 1995.

An interaction diagram shows the member functions called over time during a specific task. Time flows from top to bottom. A column shows an object and its lifetime. Its vertical line is dashed before the object exists. A rectangle lies on the line when the object is active. A solid vertical line shows the object exists, but is not active. Objects are generally named as "aClassName."



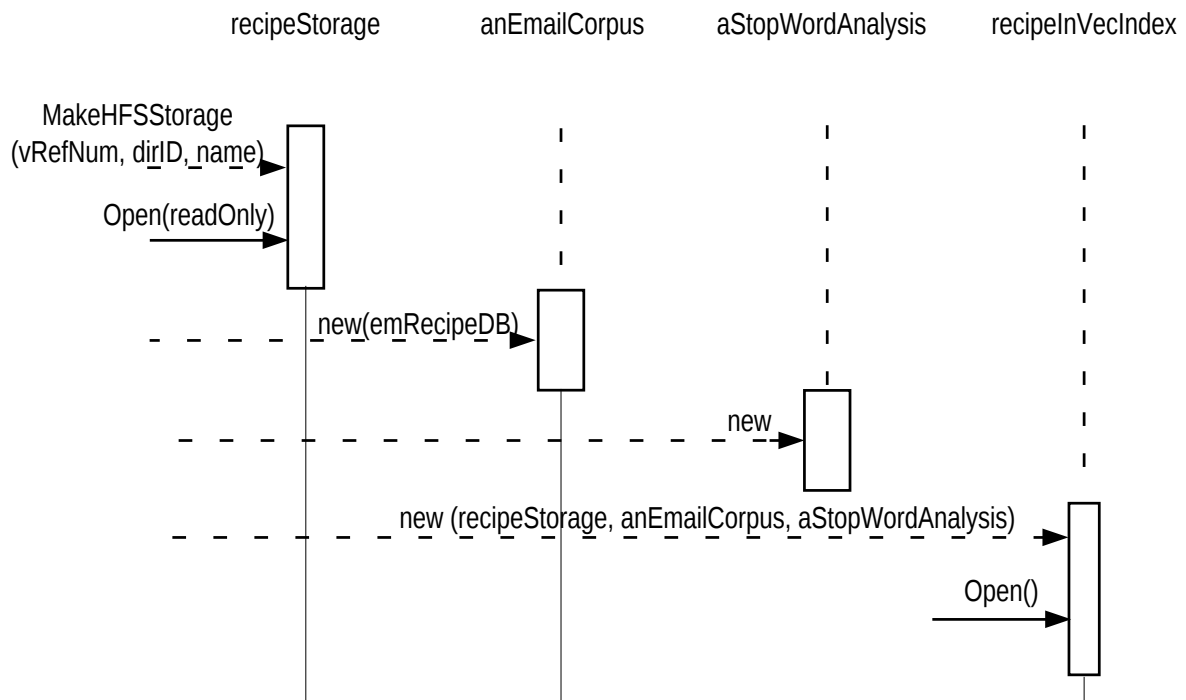
Arrows entering active objects are messages, or calls to member functions. Those not coming from another object are invoked by the task itself. Dashed arrows construct an object.

This example shows the process of creating an iterator and looping through documents. The task calls GetDocIterator on an existing object, anInVecIndex. This iterator will create the object anIADocIterator. When the task invokes GetNextDoc(), it will create the object anHFSDoc.

The code for this diagram would be as follows:

```

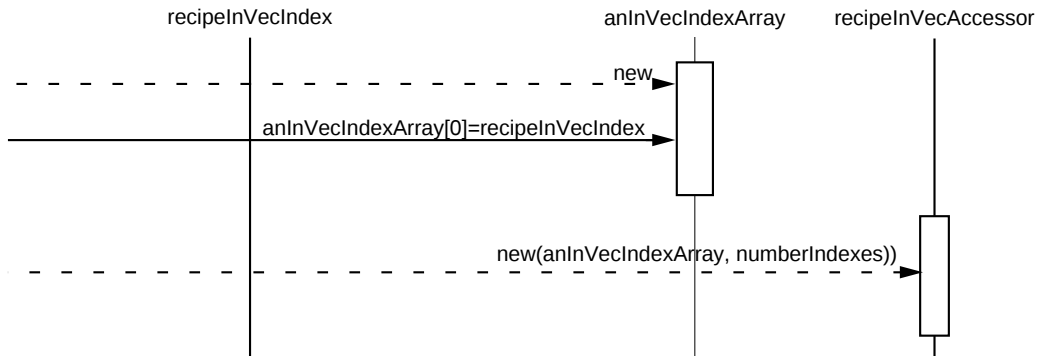
IADocIterator* anIADocIterator
anInVecIndex.GetDocIterator();
HFSDoc* anHFSDoc
while(anHFSDoc=(HFSDoc*)anIADocIterator->
  GetNextDoc()) {
}
  
```

Figure 3-5 Interaction diagram for establishing a recipe index

Build Accessor

After it is established, the recipe index can be used to create an accessor. Accessors can be built to handle more than one index; although we only have one, we must place the recipe index into an array for construction. Figure 3-6 shows the objects and operations required to build an accessor.

See “Building an Accessor” on page 6-6 for more general information on constructing accessors.

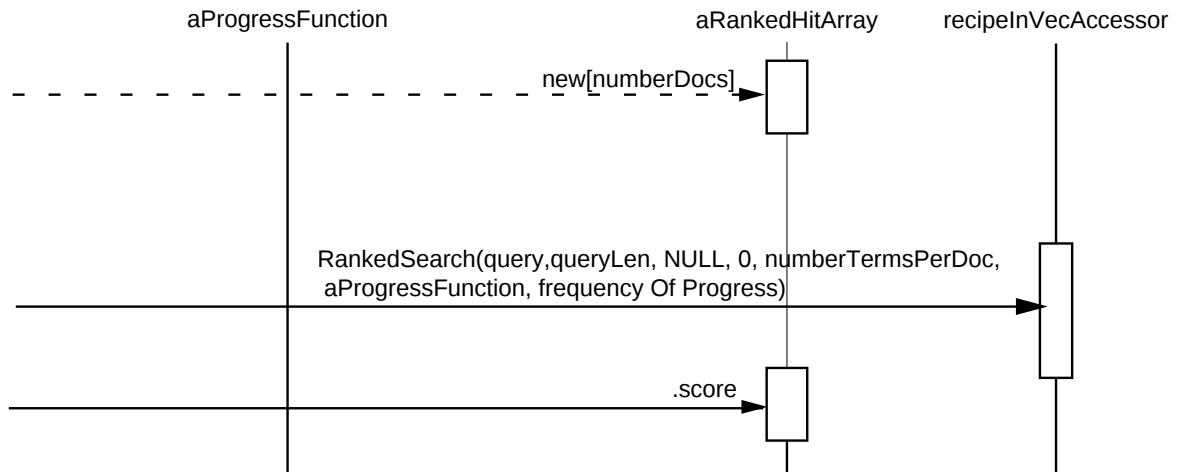
Figure 3-6 Interaction diagram for building an accessor

Query Accessor

This is a simple query. See “Answering Queries” on page 6-7 for more information on queries in general.

The application may wish to establish a progress reporting function. AProgressFunction is a function located within the application program whose address is passed to the accessor. The accessor will invoke this function every frequencyOfProgress ticks. See “Reporting Progress” on page 6-7.

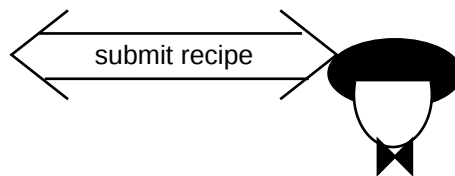
Figure 3-7 shows the objects and operations required for doing the search.

Figure 3-7 Interaction diagram for a simple query

Report Recipes

Each RankedHit will contain the necessary information to locate the email doc in the database. Report Recipes matches the hits to the database, and passes along the top matching recipes to the patron. This is application code outside of the IAT interface, so we won't show the internal design here.

Submit Recipe

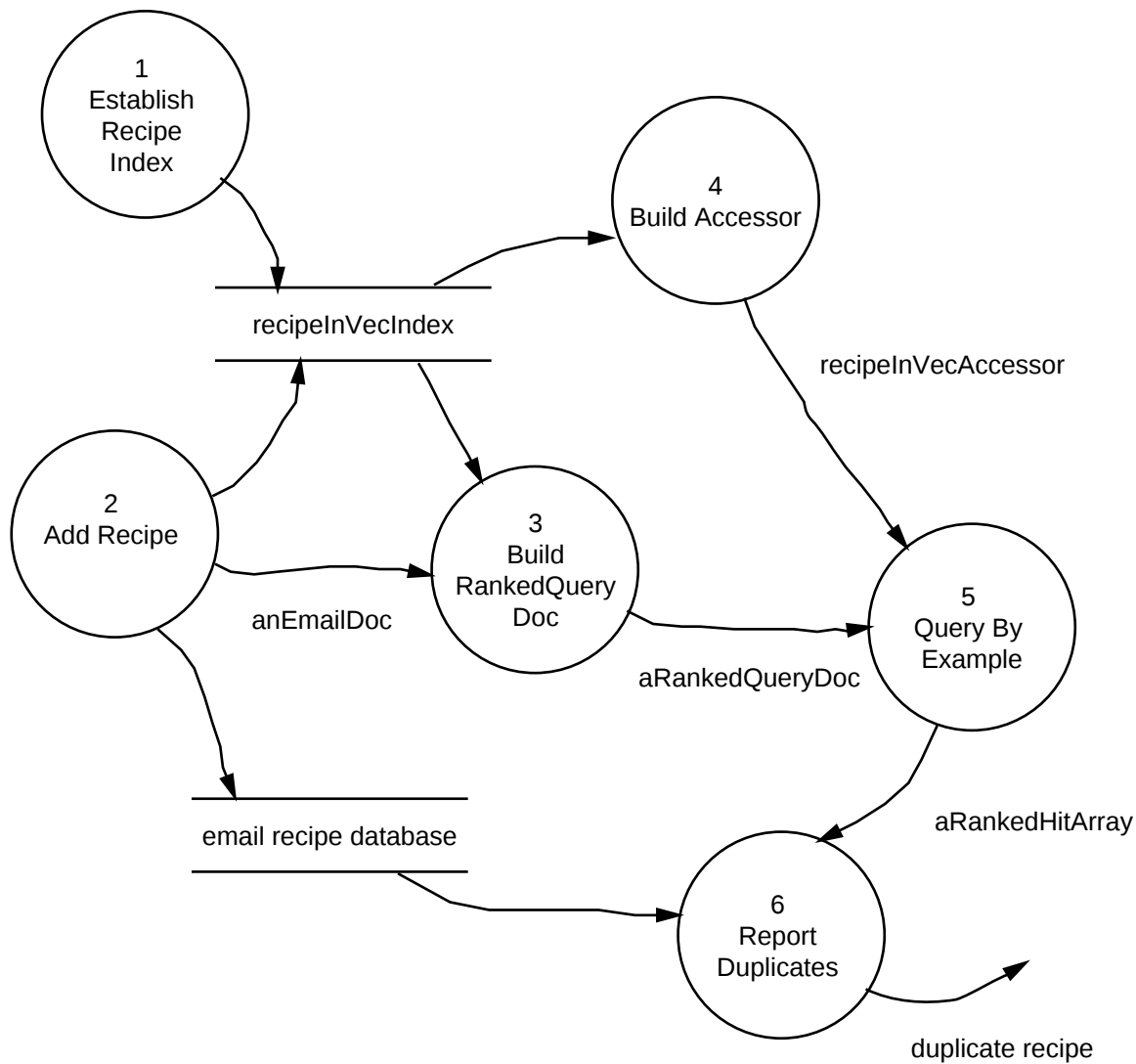


Description

The patron submits recipes to the system via e-mail. Each recipe is added to the index. The task then uses the recipe as a query to find similar recipes. These might be duplicates. If similar recipes are found, they are reported to the chef.

Subtasks

Figure 3-8 shows the subtasks used to add the recipe.

Figure 3-8 The subtasks of submit recipe

Establish Recipe Index

The recipe index is already in storage and must be established. This is the same as recipe query except that the storage must be opened “writable” to allow the index to be updated. See “Establish recipe index” on page 3-7.

Add Recipe

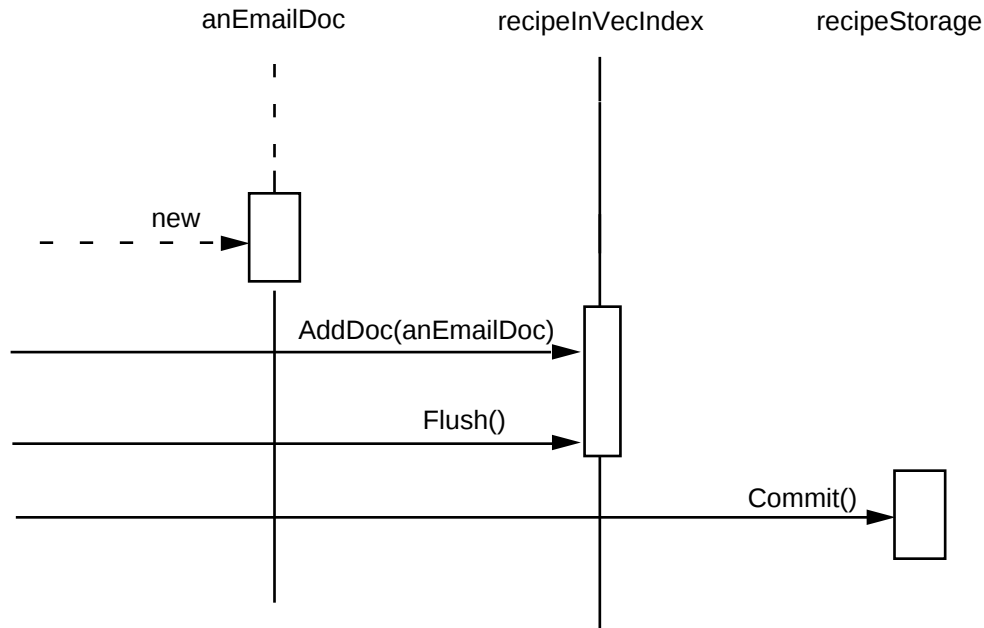
The email recipe must be added to the corpus (which is done by adding it to the email recipe database) and then added to the index. This is an individual update of the index;

Designing an Application

for more information on doing this in general, see “Updating by Individual Document” on page 5-16.

Figure 3-9 shows the objects and operations required to add a document to the index.

Figure 3-9 Interaction diagram for add recipe

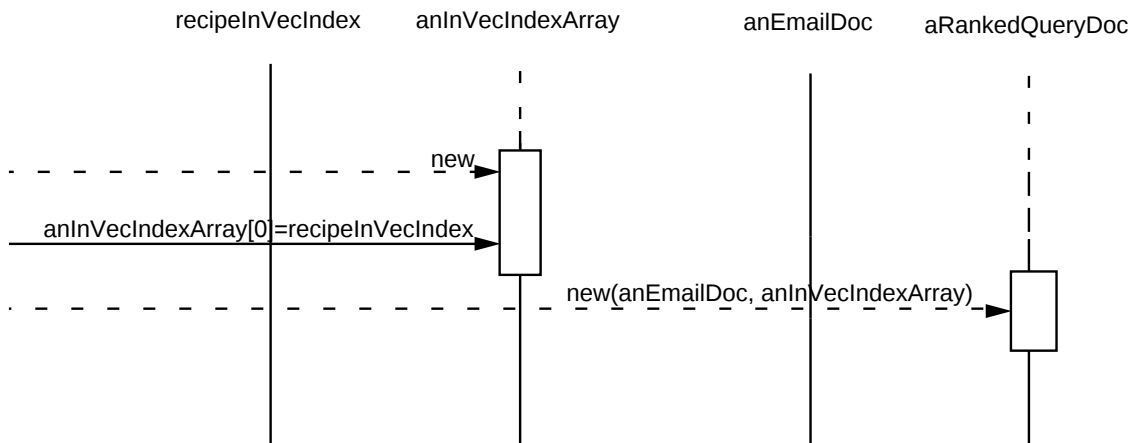


Build Ranked Query Doc

Locating possible duplicates requires a query by example. The recipe is used as an example to an accessor built from the index. The accessor will locate recipes using similar terms. Any recipe that is not the selected recipe but that scores above 0.8 is considered to be a possible duplicate.

See “Answering a Query by Example” on page 6-11 for more information on doing a query by example.

The first step in doing this query is to build the example document. This is a ranked query doc; that is, a document that will be used for a ranked query on the index. Figure 3-10 shows the objects and operations for creating this example.

Figure 3-10 Interaction diagram for creating a RankedQueryDoc**Build Accessor**

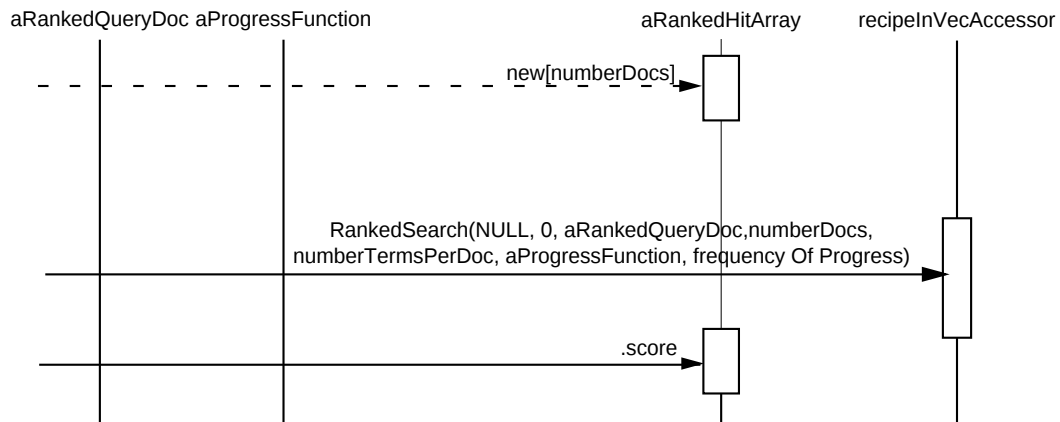
The accessor is built for the index just as it was in the recipe query. See “Build Accessor” on page 3-9.

Query By Example

The query by example is similar to the simple query, only a RankedQueryDoc, rather than a string of terms and its length, is provided to the RankedSearch function.

There may not be a progress function required if the application does not need to display progress.

Figure 3-11 shows the objects and operations to do the search.

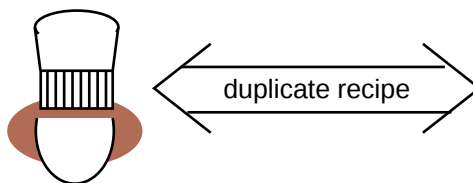
Figure 3-11 Interaction diagram of query by example

Report Duplicates

Report duplicates uses the array of ranked hits provided by the query. The top document in this array is the submitted recipe. Those following are the closest matches, in order.

Each ranked hit contains anEmailDoc for the recipe. This object has the necessary information to locate the recipe on the database and report any possible duplicates to the chef.

Duplicate Recipe



Description

Recipes which might be duplicates are queued. The chef will initiate a review of these recipes, and indicate which are to be deleted from the index.

Subtasks

This is also an individual update.

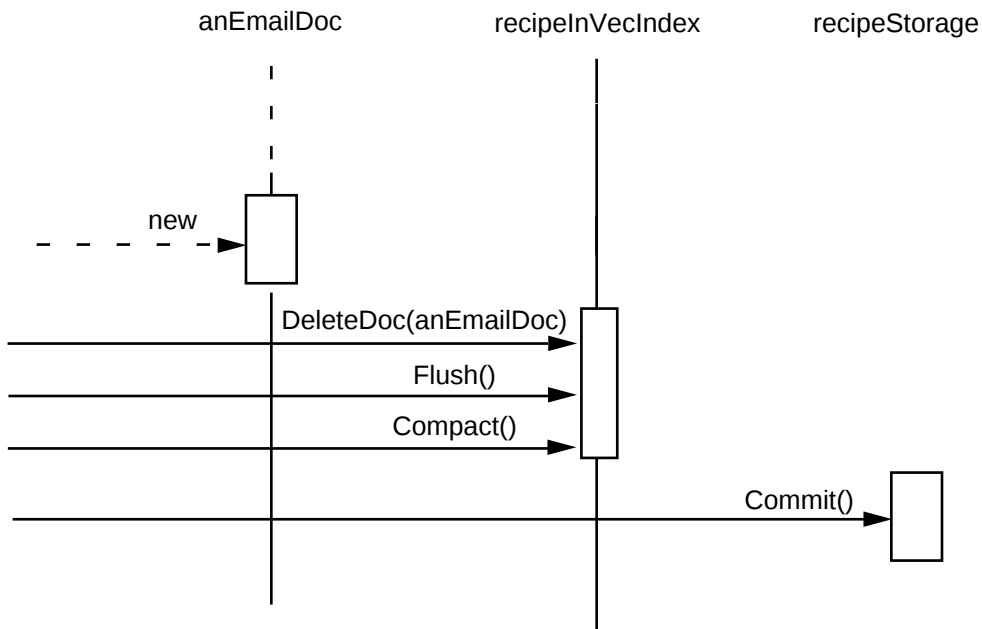
The recipe index must be established as above (see “Establish recipe index” on page 3-7). The storage should be opened as “writable.” Each document will be deleted from the

Designing an Application

index, then from the database. Storage is committed only after all deletions have been done.

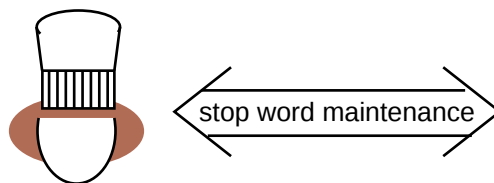
Figure 3-12 shows the objects and operations to delete recipes from the index.

Figure 3-12 Interaction diagram for deleting recipes from the index



This is another individual update. See “Updating by Individual Document” on page 5-16 for more general information.

Stop Word Maintenance



Designing an Application

Description

The application will provide a means for the chef to review and update a list of stop words, that is, words which should not be considered as content terms when found in a recipe (such as cup).

When the stop list is updated, all recipes should be re-analyzed to reflect the new filter. This is done by the Update() function, which will synchronize the index with its corpus and re-analyze all documents.

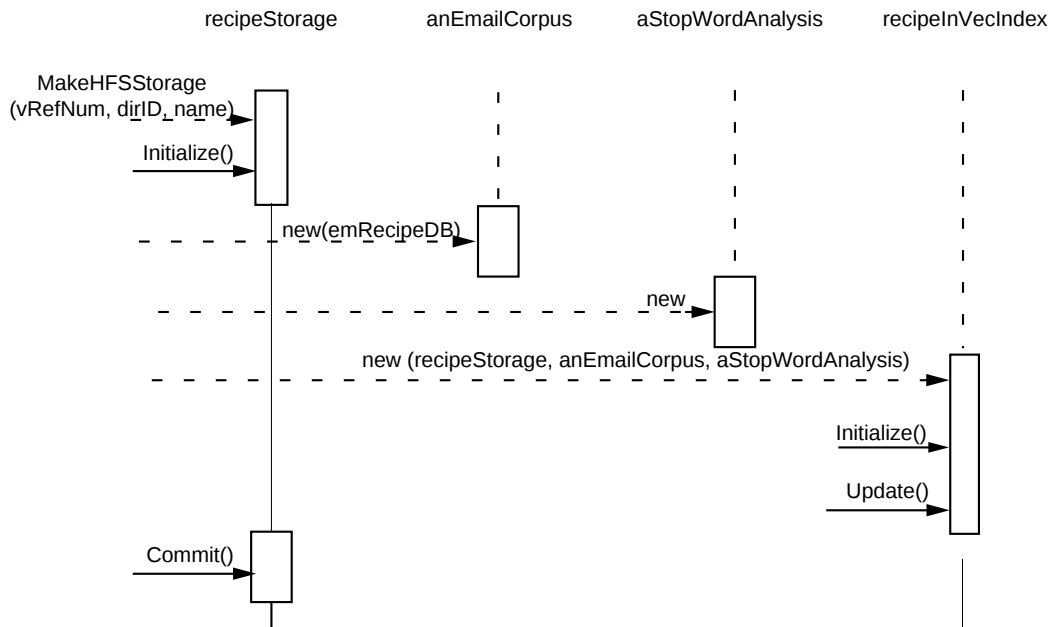
Subtasks

This task is small enough that no subtasks are required.

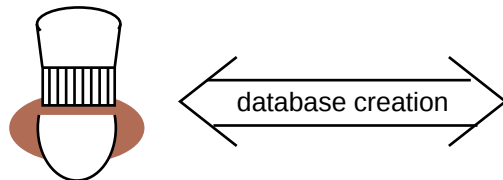
Even though the storage already exists, the application should initialize the storage and create the index as new for this update, as every document will be re-analyzed.

See “Synchronizing an Index to the Corpus” on page 5-15 for more information on this function in general.

Figure 3-13 shows the objects and operations required to rebuild the index and reanalyze all the documents.

Figure 3-13 An interaction diagram for a complete update

Database Creation



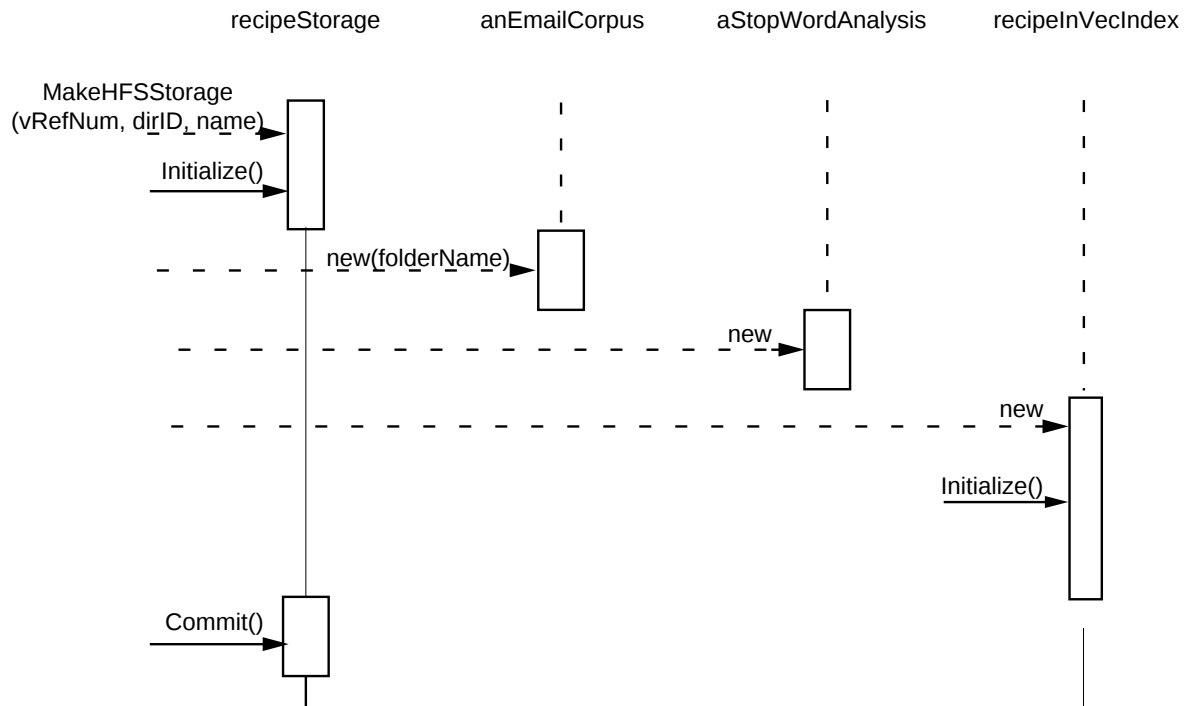
Description

The creation of a new database is a “one-time” task. It initializes an empty database and recipe index. The database initialization is application specific, and must be done before the index creation.

For more general information on creating indexes, see “Creating an Index” on page 5-8.

Figure 3-14 shows the objects and operations required to create a new index.

Designing an Application

Figure 3-14 Interaction diagram for initializing an index

Designing an Application

Common Practices in IAT

Primitive Types	3
Globals	3
Exceptions	3
Exception codes	4
Throwing Exceptions	4
Memory Allocation	5
Base Classes	10
Deletion of Allocated Memory	13

Common Practices in IAT

There are certain classes and practices used throughout IAT and recommended for use by applications. This chapter documents those practices including:

- the utility classes used throughout IAT
- general IAT error handling
- memory allocation
- copying without copy constructors

The header file `IACommon.h` contains these classes and utilities.

Primitive Types

There are certain types defined and used throughout all classes in IAT:

```
typedef unsigned char    byte;
typedef long             int32;
typedef unsigned long    uint32;
typedef int              bool;
```

Other types are more specific to a particular class or class category, and are documented in the reference for that category.

Globals

IACommon defines this global, which is used to determine the default block size for most I/O.

```
extern uint32  IADiskBlockSize
```

The default is 4096 bytes.

Exceptions

IAT uses exceptions for error handling. There is a class `IAException` that contains data members for an exception code, a brief message, and a debugging hint regarding the location where the exception was raised. The message is specified when the exception object is constructed.

```
IAException(const char* message);
```

and can be accessed by the `What` method

```
const char*      What();
```

The other two data members can be obtained and modified through these access methods:

```
const char*      GetLocation();
```

Common Practices in IAT

```

void          SetLocation(const char* location);

int32         GetCode();

void          SetCode(int32 code);

```

However, exceptions can be set up more easily by using the assertion macros described below under “Throwing Exceptions.”

Exception codes

The type `IAExceptionCode` is provided to allow definition of integer exception codes. .

```
typedef const int32      IAExceptionCode;
```

Several `IAExceptionCodes` are defined throughout the IAT classes, using 4-character mnemonics. These are documented together with the classes that raise them. The `IAAssertionFailure` is a general code used for error handling in classes.

```
IAExceptionCode         IAAssertionFailure = 'VTWN';
```

Throwing Exceptions

Several macros have been defined to facilitate use of IAT exceptions.

IAThrowException

Input

`exception`
the exception object to be thrown

Notes

The basic method for throwing an exception is `IAThrowException(exception)`.

IAAssertion

Input

`conditional`
A test to be made.

Common Practices in IAT

message

A string describing the exception.

code

An integer exception code.

Notes

Takes any boolean expression as an argument. If the condition is not true, an exception with the given code and message is thrown.

Usage

```
IDDoc* addIDDoc = (IDDoc*)iDDocs->Next();
IAAssertion(addIDDoc, "No more documents", 'VTWN')
```

In this example, addIDDoc is NULL if the iterator has reached the end of the set. IAAssertion will cause an abort if addIDDoc does not exist.

Memory Allocation

IAT uses specialized versions of malloc() and free() for all of its memory allocation. This is to improve performance; most provided malloc() functions perform poorly with large numbers of small objects. This section lists the functions available.

At some level, IAT still must call an external allocator to be given memory. Developers may register their own allocator to be called by setting the variables IAAllocationFunc and IADeAllocationFunc. If you register an allocator, you must also register a deallocator.

IAAllocationFunc is declared as a pointer to a function with the following prototype:

```
void* funcName(size_t size);
```

IADeAllocationFunc is declared as a pointer to function with the following prototype:

```
void* funcName(void* object);
```

Listing 4-1 shows how you might define and register your own allocator.

Listing 4-1 Defining and using your own memory allocator.

```

void* MyAllocator(size_t size);
void* MyAllocator(size_t size)
{
    void* mem = (void*)malloc(size);
    return mem;
}

void* MyDeAllocator(void* obj);
void* MyDeAllocator(void* obj)
{
    free(obj);
}

void main()
{
    IAAAllocationFunc = &MyAllocator; // allocation callback
    IADeAllocationFunc = &MyDeAllocator; // deallocation callbac

    // Now go ahead and call functions that will require memory...
    ...
    StringPtr folder = "\pMacintoshHD:MyFolder:Documents";
    DemoUpdate(folder);
    ...
}

```

The Memory Functions

IAMalloc

Input

size_t size
amount of memory required

Output

void*

IAFree

Input

`void* object`
 The item whose memory is to be freed.

IAMallocSized

Input

`size_t size`

Output

`void*`

Notes

A sized version of IAMalloc. Use when you know the size at free time.

IAFreeSized

Input

`void* object`
 The object to be deleted
`size_t size`
 The size of the memory to be freed

IAMallocArray

Input

`type`
 the type of the object in the array
`length`
 The number of objects in the array

Common Practices in IAT

Output

```
void*
```

Notes

This is a macro which will allocate the memory for `sizeof(type) * length`.

▲ **WARNING**

Do not use this function if the class contains virtual members, or if the default constructors do anything.

Use `IAFreeArray` to free the memory.

Usage

```
byte* name = IAMallocArray(byte, len + 1);
```

IAFreeArray

Input

```
void* object
        the array to be freed
```

Notes

A macro that frees memory allocated by `IAMallocArray`.

IAMallocArraySized

Input

```
type
        the type of the item in the array
length
        the number of items in the array
```

Output

```
void*
```

Notes

A sized version of `IAMMallocArray`; use when you know the size at free time. See `IAMMallocArray`.

IAFreeArraySized

Input

`void* object`
the array to be freed

Notes

See `IAMMallocSized` and `IAMMallocArray`.

IAMMallocStruct

Input

structure
the structure to be allocated

Output

`void*`

Notes

This is a macro that does an `IAMMallocSized()` allocation of a structure.
See `IAMMallocSized`.

IAFreeStruct

Input

`void* object`
the object to be freed

`type`
the type of the object

Notes

This does an IAFreeSized of the structure. Used with IAMallocStruct

Base Classes

IAStruct and IObject are base classes that serve as the parent for almost every IAT class or structure. They do this to allow:

- the use of IAMalloc and IAFree functions for new and delete functions
- the prevention of a copy constructor; IAT does not support copy constructor due to problems with C++; the base class defines a nil constructor in private to prevent creation.

Class IObject

Header: ICommon.h

Hierarchy

A public subclass of IAStruct. See “IAStruct” on page 4-11.

The base class of almost every IAT class.

Description

A class created to ensure the use of IAMalloc and IAFree for the new and deletion operators, ensure the presence of a virtual destructor, and ensure that no copy constructor exists for any of its subclasses.

Public Member Functions

constructor

destructor

Virtual.

operator delete

Input

`void* object`
 the item to be deleted

`size_t type`
 the type of the object to be deleted

Output

`void*`

Notes

Calls `IAFreeSized(object, size)`.

operator new

Input

`size_t type`
 the type of the object to be created

Output

`void*`
 the object

Notes

Calls `IAMallocSized(size)`.

IAStruct**Class**

Header: `IACCommon.h`

Hierarchy

The base class of almost every IAT class.

Description

A class created to ensure the use of IAMalloc and IAFree for the new and deletion operators.

Functions

operator delete

Input

`void* object`
the item to be deleted

Output

`void*`

Notes

Calls IAFree(object).

operator new

Input

`size_t size`
the type of the object to be created

Output

`void*`
the object

Notes

Calls IAMalloc(size).

Deletion of Allocated Memory

C++ does not ensure that memory is deleted when the stack is unwound, such as when exceptions are thrown. IAT provides utility classes that can be used to ensure that any `IAObject` or other item allocated with the `IAMalloc()` functions is deleted when an exception is thrown.

IADeleteOnUnwind

Class

Header: `IACommon.h`

Hierarchy

Base Class.

Description

A class that ensures the destruction of a pointer to an `IAObject` when an exception is thrown. This should be constructed following the creation of a pointer.

Public Member Data

`IAObject* object`
The created object which is to be destroyed.

Public Member Functions

constructor

Input

`IAObject* object`

Usage

```
IATokenStream* ts =
    analysis.MakeTokenStream(corpus.GetDocText(&doc));
IADeleteOnUnwind delTs(ts);
```

destructor

Class **IADeleteArrayOnUnwind**

Header: IACommon.h

Hierarchy

Base Class.

Description

A class that ensures the destruction of an array allocated with `IAMallocArray` when an exception is thrown. This should be constructed following the creation of an array. See also `IADeletePointerArrayOnUnwind`, which should be used for an array of pointers to `IAObjects`.

Public Member Data

`void* array`
The created array which is to be destroyed.

Public Member Functions

constructor

Input`void* array`**Usage**

```
DocID* docIDBases = IAMallocArray(DocID, indexCount);
IADeleteArrayOnUnwind delDocIDbases(docIDBases);
```

destructor

IADeletePointerArrayOnUnwind

Class

Header: IACommon.h

Hierarchy

Base Class.

Description

A class that ensures the destruction of a array of pointers to IAObjects when an exception is thrown. This should be constructed following the creation of the array.

Public Member Data

IAObject array**
The created array which is to be destroyed.

uint32 length
The number of items in the array

Public Member Functions

constructor

Input

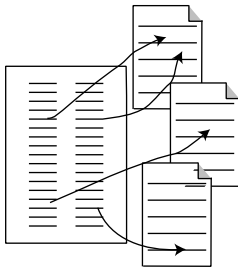
IAObject array**
The created array which is to be destroyed.

uint32 length
The number of items in the array

Usage

```
for (uint32 i = 0; i <= nDocs; i++) tfMaps[i] = Nil;
IADeletePointerArrayOnUnwind delTFMaps((IAObject**)tfMaps,
                                nDocs + 1);
```

destructor



Index Category

Choosing an Index Type	3
Index Types Currently Available	4
Comparison of Searches Available	5
Index Size vs. Speed	7
Common Operations	8
Creating an Index	8
Establishing an Existing Index	11
Updating an Index	14
Iterating Through the Documents in an Index	19
Merging Indexes	20
Compacting an Index	22
Index Class Category Reference	23
Header Files in the Index Category	23
Class Specifications	25
DocInfo	25
FreqPosting	27
FreqPS	29
FreqTerm	31
IAIndex	31
IAIndexTypes	43
InVecIndex	45
InvertedIndex	46
TermIndex	49

Index Category

TermInfo	59
TFComponent	63
TFVector	63
VectorDocInfo	66
VectorIndex	67
Class Utilities	72
Typedefs	74
Extern Data	76
Constants	77
Index Exceptions and Error Handling	78

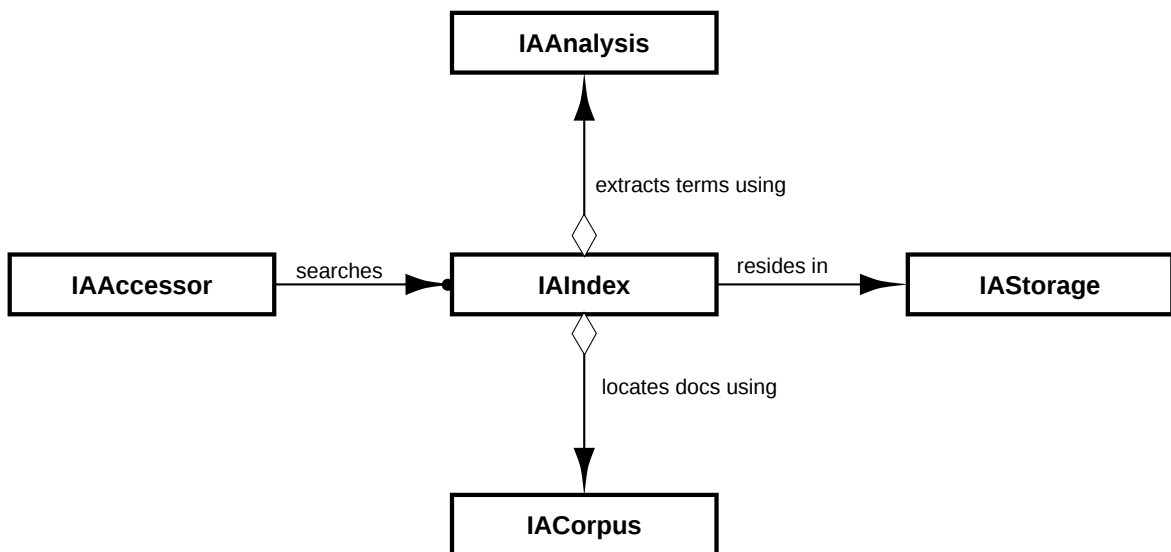
Index Category

The index is the heart of the IAT classes. The index uses the corpus to locate documents and extract their text, gives it to the analysis to extract terms, and builds and stores the mappings of terms to documents.

Once an index is created it can be searched. Searches can be alone or with other indexes.

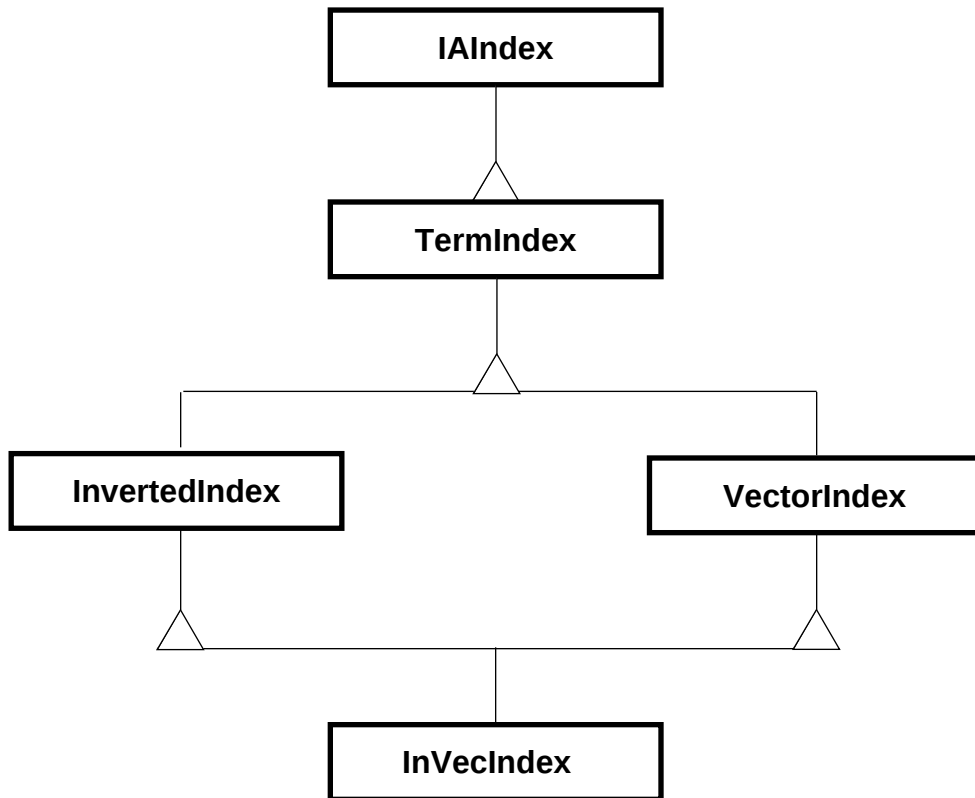
Figure 5-1 shows the abstract class, `IAIndex`, and its major relationships.

Figure 5-1 An overview of an index



Choosing an Index Type

Designers optimize the content and organization of an index to allow it to perform select functions efficiently. Different types of indexes exist to allow optimal performance of different functions. You should choose the appropriate index to deliver the best performance for the primary usage of the index.

Figure 5-2 Index inheritance tree

Index Types Currently Available

As shown in Figure 5-2, the current implementation of IAT has these types of indexes:

- term index (**TermIndex**), contains the primary structures and operations of any index containing terms. While instantiable, it functions primarily as a base class
- inverted index (**InvertedIndex**), which indexes each term to the documents in which it occurs
- vector index (**VectorIndex**), which maps each document to its terms
- inverted and vector index (**InVecIndex**), which stores a combination of the information found in inverted and vector indexes

For most uses of an index for ranked searching, your choice is between an inverted index or an inverted and vector index. The vector index, alone, is used primarily for measuring similarity between sets of documents.

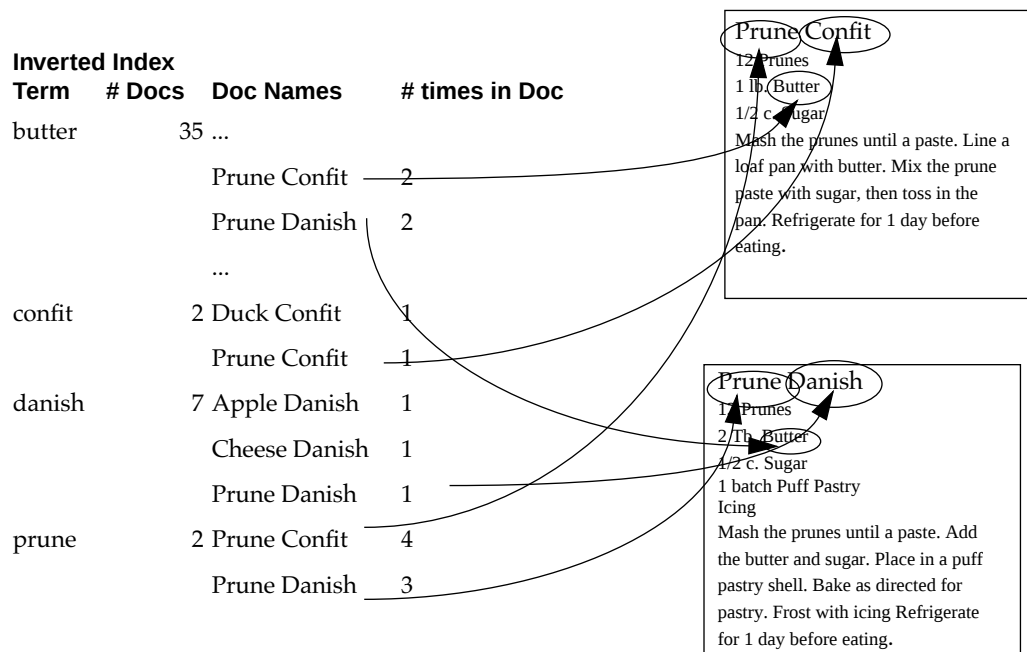
When you choose an index type you are making a trade-off between functionality, the time it takes to build an index, and the space the indexes occupy.

Comparison of Searches Available

Inverted Index

An inverted index allows a rapid search by a term; that is, given a term, it will have the location of all documents in the index that contain the term (see Figure 5-3). An inverted index also stores how many documents have a given term, the size of the document, and how frequently the term appears in the document. This allows ordering of the documents by how frequently the terms appear compared to the size of the document, that is, it allows the use of a statistically-based ranking system.

Figure 5-3 An inverted index



These questions could be rapidly answered based on an inverted index of a recipe collection:

- how many recipes contain cheddar cheese?
- list the recipes containing cheddar cheese
- which terms are used most often in this collection of recipes?
- which recipes uses some combination of cheddar cheese, mushrooms, and white wine?
- which recipe in this collection is closest to this example recipe?

Index Category

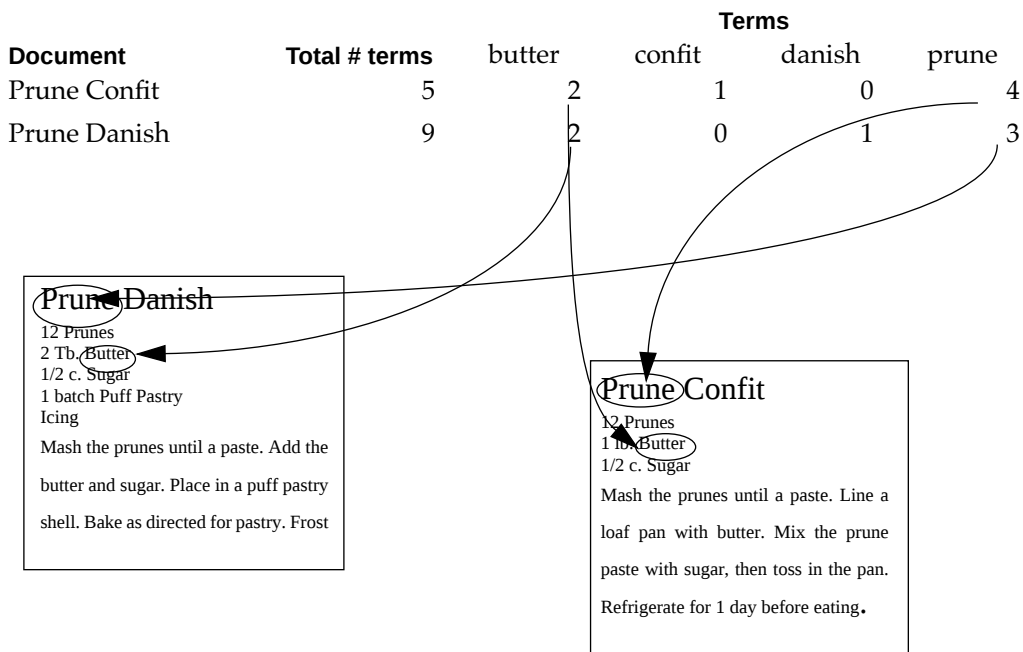
The time it takes to search a collection with an inverted index is proportional to the number of terms in the query, and, to a lesser extent, the total number of terms used by all documents in the collection.

Vector Index

A vector index records the terms in each document (see Figure 5-4). It computes how often terms occur in a single document relative to the distribution of terms over the collection. The vector index provides the data and functions to allow efficient comparison of two documents so you can judge how close they are in content. This index might be used to route messages into the sub-collections they match best; for example, a random set of recipes could be organized into groups with similar ingredients.

Figure 5-4 A vector index

Vector Index



Although you can do other searches with just a vector index, it is typically slower than using an inverted index as each vector must be read. The time to search is proportional to the number of documents in the collection.

Index Category

Inverted Vector Index

Unless index space is a particular concern, you will generally want to use an inverted vector index. This will speed up certain powerful operations such as relevance feedback (also known as query-by-example or similarity search), in which the application asks the IAT to find “documents like this one.” In the case of the RecipeSwap, for example, Irina uses this feature to find recipes similar to the newest one (when checking for potential duplicates).

An inverted index can also perform relevance feedback. However, it must re-analyze the text of the document in order to do so. To save this time (but at the expense of space), use an inverted vector index.

Comparison of Time and Space Requirements

Table 4-1 Comparison of index types for time and space

Index Type	Actual Time in Minutes	MB Per Hour	Index Space	Index Space Overhead
Inverted Index	11.18	196	5.9 MB	16.3%
Vector Index	6.83	321	6.9 MB	19.1%
Inverted Vector Index	11.03	199	10.8 MB	29.6%

This table compares relative times to build the index and the amount of space the index takes for a folder and sub-folders containing about 12,000 documents. The document set occupies 36.6 MB. The indexing was done on a Power Macintosh 9500/132. Although results will vary by document content, this may give some idea of the trade-offs involved.

Note

The documents were SGML-tagged articles from the Wall Street Journal, concatenated together in groups of about 100 per file. This is comparable to a typical e-mail or database application, where many documents (or records) are stored in a small number of storage files. For applications in which each document is a separate file, greater file I/O will result in substantially decreased performance.

Index Size vs. Speed

It takes much longer, proportionately, to update a very large index than to update several smaller indexes. As the size of the collection grows and memory is held constant, indexing speed will gradually decrease. For example, in one test indexing a 1 gigabyte collection, performance was about 75 MB/hr. For this reason, you may wish to build smaller indexes on partitions of your collection, and search them simultaneously.

Index Category

Alternatively, you may wish to use the Merge function to consolidate several small indexes into one.

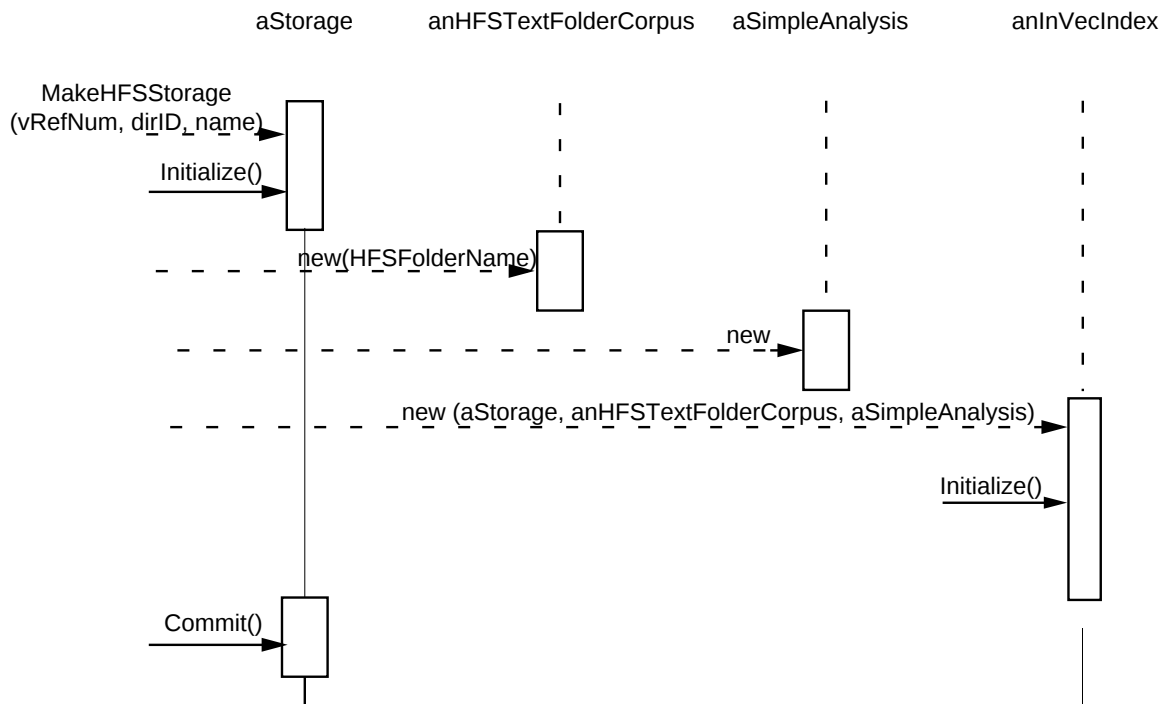
Common Operations

Creating an Index

Creating a new index requires that all items upon which it depends be created first. You must create:

- a storage in which the index will reside. See “Creating New Storage” on page 9-5.
- a corpus to organize a collection of documents and extract their text.
- an analysis for locating the terms in the documents.

This example just creates the index framework; to load the index, see “Updating an Index” on page 5-14. The example creates an index for an HFSTextFolderCorpus using a Simple Analysis.

Figure 5-5 Interaction diagram for index creation**Listing 5-1** Creating an index

```

// include Mac types for HFS-related items
#include <Types.h>

// choose a storage implementation
#include "HFSStorage.h"

// choose a corpus implementation
#include "HFSTextFolderCorpus.h"

// choose an analysis implementation
#include "SimpleAnalysis.h"

// choose an index implementation
#include "InVecIndex.h"

// get the user information (using constants for the sake of this example)
StringPtr      name = "\\precipes.index";
StringPtr      HFSFolderName = "\\pMacintosh HD:Corpora:recipes";

```

Index Category

```

short          vRefNum = 0;
long           dirID = 0;

// create storage for the index
IAStorage * aStorage = MakeHFSStorage(vRefNum, dirID, name);
IADeleteOnUnwind delInxStorage(aStorage);
aStorage->Initialize();

// create index for folder (creates corpus and analysis)
InVecIndex anInVecIndex(aStorage,
    new HFSTextFolderCorpus(HFSFolderName), new SimpleAnalysis());
anInVecIndex.Initialize();

// commit the storage to disk
aStorage->Commit ();

```

Naming the Index Root Block

You can establish a block of storage and ensure the index root block is stored in this block. This allows access to the index block at another time. This may be necessary if an application needs to open this index and does not know which type of index it is. See “Establishing an Index Whose Type is Unknown” on page 5-13.

When storage is allocated, you must allocate a named IO block. Then this block id can be provided to the index constructor. See “Allocating and Deallocating Blocks of Storage” on page 9-6 for more information.

InVecIndexType is a constant indicating this is an inverted vector index.

Listing 5-2 Differences when creating an index with a named block

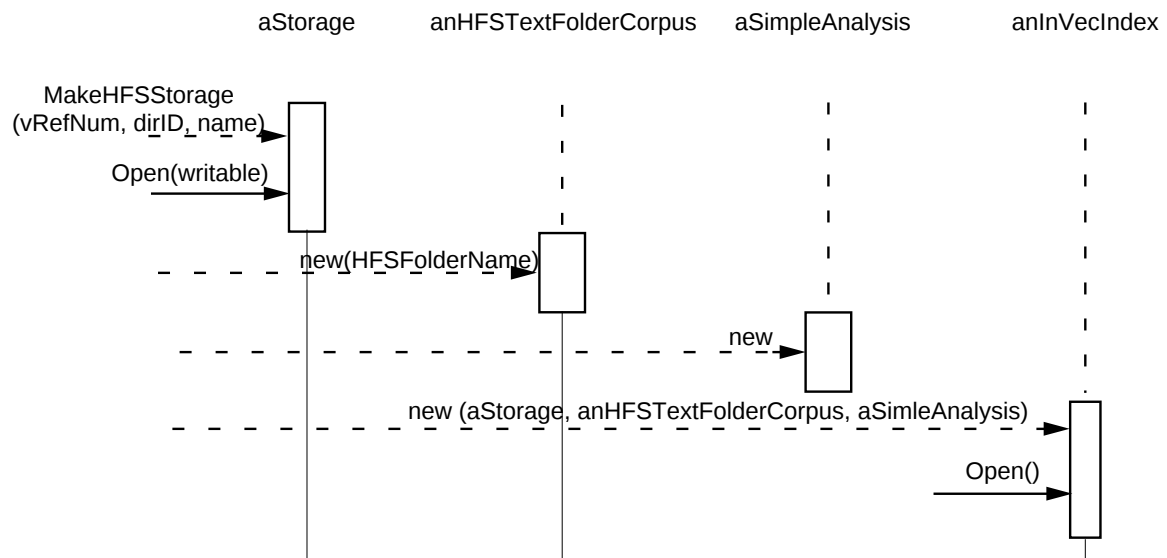
```
// create storage for the index
IAStorage* aStorage = MakeHFSSStorage(vRefNum, dirID, name);
IADeleteOnUnwind delInxStorage(aStorage);
aStorage ->Initialize();
const char* aBlockName="INDEXROOT";
IABlockID anIABlockID=aStorage->AllocateNamedBlock(aBlockName);

// create index for folder (creates corpus and analysis)
InVecIndex anInVecIndex(aStorage,
    new HFSTextFolderCorpus(HFSFolderName), new SimpleAnalysis(),
    InVecIndexType, anIABlockID);
anInVecIndex.Initialize();
```

Establishing an Existing Index

To reuse a stored index, create a new index object using the same type of corpus and analysis that is in the original object and the name of the storage where the index was established. Then `Open()`, rather than `Initialize()`, this index to restore its contents. The storage must be reestablished (and opened) first. See “Opening Existing Storage” on page 9-6 for more on reestablishing storage.

This example establishes an inverted vector index with an `HFSTextFileCorpus` and a `SimpleAnalysis`. The index is established as writable, but no update has begun. To establish an index as read-only, open its storage as read only.

Figure 5-6 Interaction diagram for establishing an existing index

Listing 5-3 Establish an existing index

```
// get the user information (using constants for the same of this example)
    StringPtr      name = "\\precipes.index";
    StringPtr      HFSFolderName = "\\pMacintosh HD:Corpora:recipes";
    short          vRefNum = 0;
    long           dirID = 0;
    Boolean        writable = true;

// reestablish storage for the index
    IASStorage * aStorage = MakeHFSStorage(vRefNum, dirID, name);
    IADeleteOnUnwind delInxStorage(aStorage);
    aStorage ->Open(writable);

// reestablish index for folder (reestablishes corpus and analysis)
    InVecIndex anInVecIndex(aStorage,
        new HFSTextFolderCorpus(HFSFolderName), new SimpleAnalysis());
    anInVecIndex.Open();
```

Establishing an Index Whose Type is Unknown

If you don't know which type an index is, but you do know its index root name, analysis and corpus types, you can test the root to determine the index type. See "Naming the Index Root Block," above, to create an index to a named root block.

The following code would replace the index construction in the "Establish an existing index" code, above.

Listing 5-4 Determining type of index

```
// get the pre-defined root block ID
const char* aBlockName = "INDEXROOT";
IABlockID anIABlockID = aStorage->TOC_Get(aBlockName);

// determine which index type it is
IAIndexTypes indexTypes;
IAIndex* index;
IAReadIndexTypes(aStorage, anIABlockID, &indexTypes);
switch(indexTypes.indexType) {
    case InVecIndexType:
        index = new InVecIndex(aStorage,
                               new HFSTextFolderCorpus(HFSFolderName),
                               new SimpleAnalysis(),
                               indexTypes.indexType, anIABlockID);
        break;
    case InvertedIndexType:
        index = new InvertedIndex(aStorage,
                                   new HFSTextFolderCorpus(HFSFolderName),
                                   new SimpleAnalysis(),
                                   indexTypes.indexType, anIABlockID);
        break;
    case VectorIndexType:
        index = new VectorIndex(aStorage,
                                 new HFSTextFolderCorpus(HFSFolderName),
                                 new SimpleAnalysis(),
                                 indexTypes.indexType, anIABlockID);
        break;
    default:
        //throw exception
        IAAssertion(false, "index type invalid", 'VIIV');
}
```

Updating an Index

There are two main ways to update an index:

- ensure it is synchronized with its corpus by using the Update() function to apply any changes to its corpus
- individually add or delete documents

Index Category

No matter which means of updating you use, you must first ensure the index is established and writable. This means it must be created and initialized (see “Creating an Index” on page 5-8) or re-established from storage that has been opened as “writable” (see “Establishing an Existing Index” on page 5-11). Following the update, you must commit the storage to ensure the changes are stored in persistent storage.

Synchronizing an Index to the Corpus

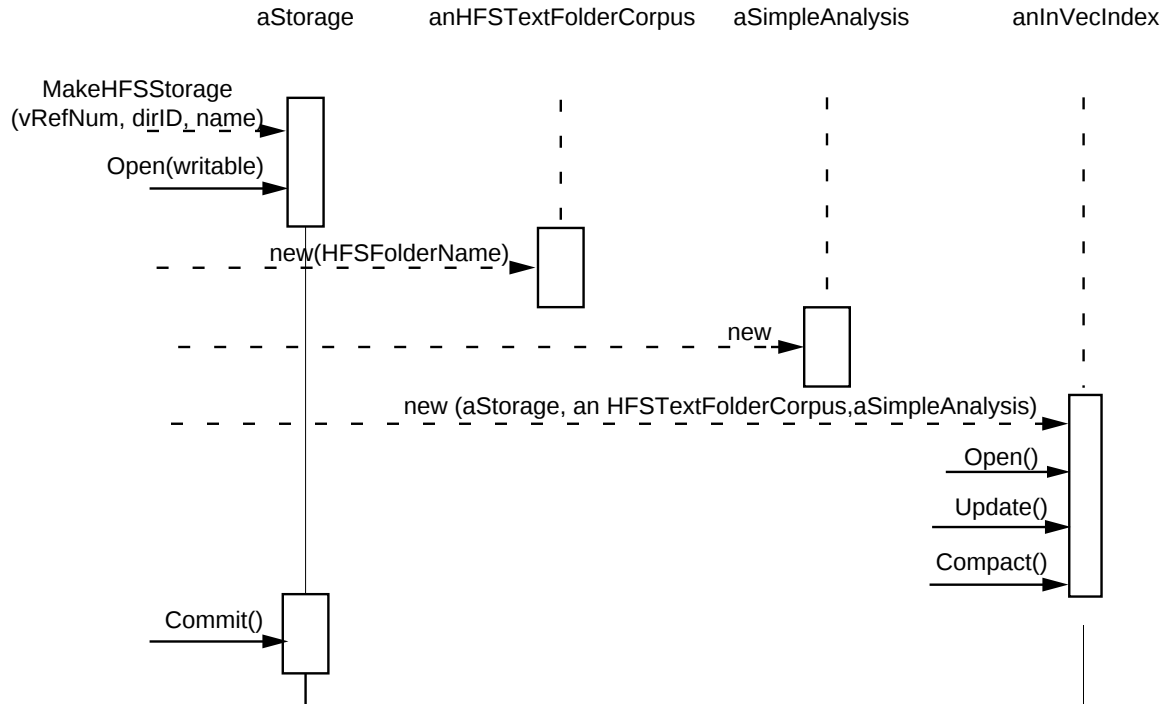
The corpus maintains the collection of documents that is indexed in the index. If changes have been made to this collection, the index may no longer reflect the corpus. For example, if the index was for an `HFSTextFolderCorpus`, documents may have been added or deleted from the corpus, or a document's text may have changed, without any change to the index. The index would no longer be synchronized with its corpus.

You can ensure an index matches its corpus by using the `Update()` function of the index. This function depends on having a corpus with an iterator, that is, one which can provide a list of each document in the corpus.

This function will:

- remove any documents from the index that are no longer found in the corpus
- add any documents to the index that are in the corpus but not in the index
- re-analyze any documents that have been modified since the last index update.

If a new filter has been added to an analysis (for example, more stop words) this update will ensure every document has been reanalyzed to match that filter.

Figure 5-7 Interaction diagram of an update to match the corpus**Listing 5-5** An example of updating an index to match its corpus

```

// establish the index in storage (see above)
// update index to match corpus and re-analyze all docs
    anInVecIndex.Update();

// take care of changes caused by possible deletions
    anInVecIndex.Compact();

// commit the changed storage to disk
    aStorage->Commit();
  
```

Updating by Individual Document

You may wish to update an index without completely matching a corpus. For example, a user may “touch” just those documents to be added, deleted, or re-analyzed.

Index Category

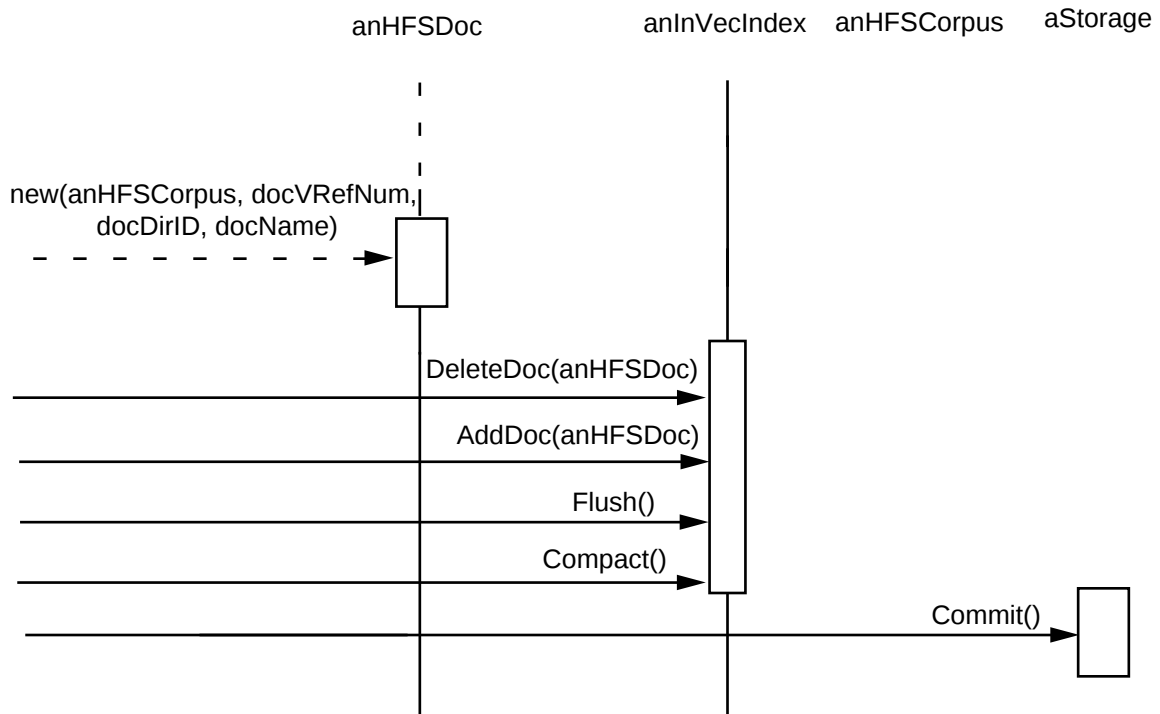
Note

If your corpus class has a document iterator, and you add or delete a document from the index, you must also, separately, add or delete the document from the corpus.

AddDoc(IADoc*) and DeleteDoc(IADoc*) allow individual updates. After a call to AddDoc, the index takes responsibility for the IADoc object passed in and will delete it at destruct time. (This is not the case for DeleteDoc.)

After a number of insertions and deletions, Flush() must be called to make the changes permanent in the index.

Figure 5-8 Interaction diagram for individual update



Listing 5-6 Updating individual documents

```
// get the user information (using constants for the same of this example)
StringPtr  name = "\\precipes.index";
StringPtr  HFSFolderName = "\\pMacintosh HD:Corpora:recipes";
short      vRefNum = 0;
long       dirID = 0;
Boolean    writable = true;
```

Index Category

```
// get the new document information
StringPtr  docName = "\pInsertMe";
short      docVRefNum = 0;
long       docDirID = 0;

// create storage for the index
IAStorage * aStorage = MakeHFSSStorage(vRefNum, dirID, name);
IADeleteOnUnwind delInxStorage(aStorage);
aStorage->Open(writable);

// create the corpus
HFSCorpusanHFSCorpus(HFSCorpusType);

// create the HFS Doc
HFSDoc *anHFSDoc =
    new HFSDoc(&anHFSCorpus, docVRefNum, docDirID, docName);

// create index for folder (creates analysis)
InVecIndex anInVecIndex(aStorage, &anHFSCorpus, new SimpleAnalysis());
anInVecIndex.Open();

// do individual updates (iterate if multiple documents)

// add or delete it
anInVecIndex.AddDoc(anHFSDoc);

// complete the update
anInVecIndex.Flush();

// commit the storage to disk
aStorage->Commit ();
printf ("Successful Completion\n");
```

Functions for Updating

These functions exist for individual updates. See the reference for these functions in “IAIndex” on page 5-31 for more information.

AddDoc

DeleteDoc

IsDocIndexed

RenameDoc

Index Category

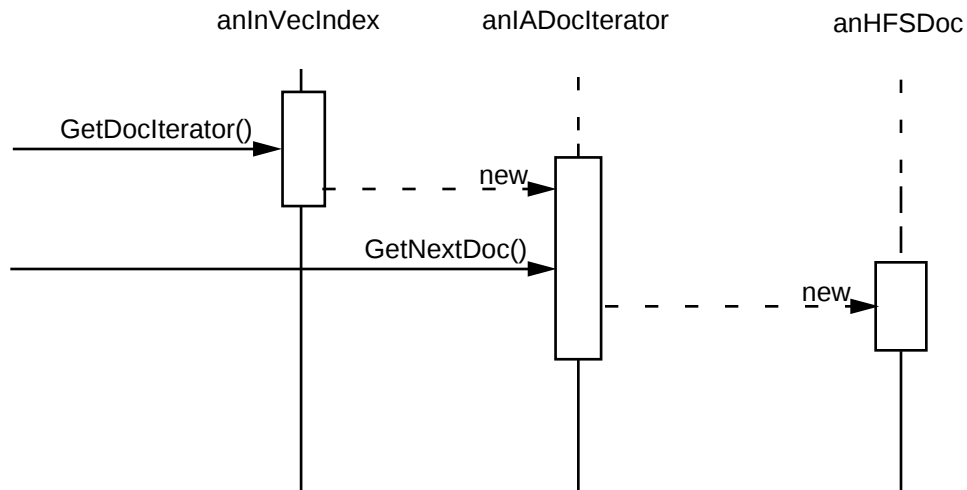
Note

If you want to re-analyze a document that is in the index (perhaps because it has changed), you should first delete it from the index and then add it back. The document will be re-analyzed and the index updated.

Iterating Through the Documents in an Index

There may be a need to list all documents found in an index, or to provide each document to another task. This can be done with an index iterator.

Figure 5-9 Interaction diagram for iterating through an index



Listing 5-7 Iterating through an index

```

// establish the index

// establish the iterator
IADocIterator* anIADocIterator=anInVecIndex.GetDocIterator();
HFSDoc*        anHFSDoc

// loop through the index // NULL returned at end
while (anHFSDoc = (HFSDoc*)anIADocIterator->GetNextDoc()) {
    PrintDocName(anHFSDoc); // application provides
}
  
```

Merging Indexes

Two or more indexes may be merged into a single index using the merge member function. This function requires several 100k/index and needs twice the disk space during the merge.

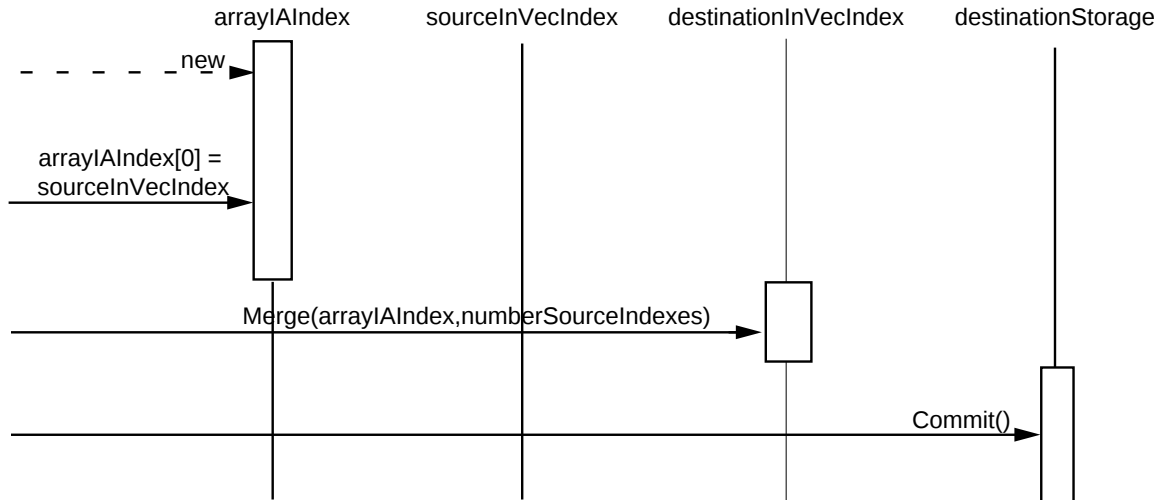
Merges are about ten times faster than building an index, and as noted before, building large indexes takes proportionally more time than building small ones. Because of this, you may wish to build several small indexes and then merge them rather than build one very large one.

Ensure these things before you merge indexes:

- the indexes have the same type of corpus and analysis
- no document is indexed in more than one of the indexes
- there is sufficient disk space to do the merge
- the indexes are not currently being updated.

Indexes must be in storage and opened before they can be merged. You may wish to open the storage of the source indexes as read-only to save memory.

If a document is present in more than one of the indexes, the merge operation will throw an exception with code `IndexDocAlreadyIndexed` ('VIAI').

Figure 5-10 Interaction diagram for a merge**Listing 5-8** Merge a source index to a destination index

```

// establish the indexes as writable
// delete duplicates from source

// make an iterator
IADocIterator* anIADocIterator = sourceInVecIndex.GetDocIterator();
uint32 docCount = sourceInVecIndex.GetDocCount();
printf ("%lu documents in the source index before \n", docCount);
HFSTextFolderDoc* anHFSDoc;

// find dupes and delete them until source exhausted
while(anHFSDoc = (HFSTextFolderDoc*)anIADocIterator->GetNextDoc()) {

    if (destinationInVecIndex.IsDocIndexed(anHFSDoc)) {
        sourceInVecIndex.DeleteDoc(anHFSDoc);
        PrintDocName(anHFSDoc);
        printf ("is duplicated in destination index\n");
    }
}

docCount = sourceInVecIndex.GetDocCount();
printf ("%lu documents in purged source\n", docCount);

// flush the changes
  
```

Index Category

```

        sourceInVecIndex.Flush();
        sourceInVecIndex.Compact();

// do the actual merge
docCount = destinationInVecIndex.GetDocCount();
printf ("%lu documents in the index before\n", docCount);

// create the array of indexes
const uint32 numberSourceIndexes = 1;
IAIndex* anIAIndexArray[numberSourceIndexes];
anIAIndexArray[0] = &sourceInVecIndex;

// do the merger
printf("Merging\n");
destinationInVecIndex.Merge(anIAIndexArray, numberSourceIndexes);
docCount = destinationInVecIndex.GetDocCount();
printf("%lu documents in the index after merging\n", docCount);

```

Compacting an Index

When a document is deleted from an InvertedIndex using DeleteDoc, the function marks the document as deleted and prevents the access to the document; the function does not actually delete the references to the documents and those terms it uses exclusively. Because of this, after many deletions, the index may contain unused information. You should periodically compact the index to remove this unused information. The recommended procedure is to compact the index just before committing the storage.

▲ **WARNING**

You **must** compact the index at least once before committing the storage after doing a number of deletes.

Compacting an index does not compact its storage. If you wish to regain the storage used by the deleted documents, use the storage class Compact function following the index compaction. See “Compacting Storage” on page 9-8 for more information.

Listing 5-9 Compact an index

```

// establish the index in storage
anInvertedIndex.Open();
anInvertedIndex.Compact();
aStorage->Commit();

```

Index Class Category Reference

Header Files in the Index Category

FreqPosting.h

FreqPosting

HighFreqTerms.h

FreqTerm

IAIndex.h

IAIndex
IAIndexTypes
IAReadIndexTypes

InVecIndex.h

InVecIndex

InvertedIndex.h

FreqPS
InvertedIndex

TermIndex.h

DocInfo
IDDoc
IDTerm
TermIndex
TermInfo

TFVector.h

TFComponent
TFVector

VectorIndex.h

VectorDocInfo
VectorIndex

Class Specifications

DocInfo

Class

Header: TermIndex.h

Hierarchy

Public subclass of IAOOrderedStorable. See “IAOrderedStorable” on page 10-14.

Description

DocInfo is the relationship between the index and a document within the index.

Relationships

DocInfo contains IADoc

One doc info contains one IADoc.

Clients

See “FreqPosting maps to DocInfo” on page 5-28.

Public Member Functions

constructor()

constructor(IADoc* document, DocID docID)

Input

IADoc*	The document.
DocID	The ID for the document.

destructor

Deletes the document.

DeepCopy

See “IAStorable.DeepCopy” on page 10-28.

Equal

See “IAOrderedStorable.Equal” on page 10-15. DocInfo equals another DocInfo if the doc equals the other doc.

GetDocID

Access method for DocInfo member data.

Output

DocID	id
	The identification number of the document within the index.

GetDocument

Access method for DocInfo member data.

Output

IADoc *	doc
	A pointer to the indexed document.

GetDocumentLength

Access method for DocInfo member data.

Index Category

Output

DocLength **length**
The total number of indexed terms in the document.

LessThan

See “IAOrderedStorable.LessThan” on page 10-16. DocInfo is sequenced by its doc member data.

Restore

See “IAStorable.Restore” on page 10-28.

SetDocument

Access method for DocInfo member data.

Input

IADoc* **doc**
The document object.

Store

See “IAStorable.Store” on page 10-30.

StoreSize

See “IAStorable.StoreSize” on page 10-29.

FreqPosting

Struct
Header: FreqPosting.h

Struct

Index Category

Description

Represents an occurrence of a term in a document. FreqPosting is the relationship between a term and an indexed document in which it occurs. See Figure 5-13 on page 5-47.

Relationships

FreqPosting maps to DocInfo

One frequency posting maps to one doc info.

This is done by carrying the DocID, a unique identifier of DocInfo.

Clients

See “FreqPS contains FreqPosting” on page 5-30.

Public Functions

constructor

constructor(DocID docID, DocLength numberTerms)

Input

DocID	The ID of the document
DocLength	The number of terms in the document

GetDocID

Access method for FreqPS member data.

Index Category

Output

DocID **id**
 The identification number of the document in which the term occurred.

GetFreq

Access method for FreqPS member data.

Output

DocLength **freq**
 The number of times the term occurred in the document.

StoreSize()

Output

IABlockSize
 The size of the blocks used to store postings

StoreSize(FreqPosting* previous)

Input

FreqPosting* previous
 The last posting stored

Output

IABlockSize
 The size of the output block used.

FreqPS

Class

Header: InvertedIndex.h

Hierarchy

Base Class.

Index Category

Description

FreqPS accesses the postings for a term in an inverted index from the storage provided. It provides a stream from which FreqPostings can be retrieved.

Relationships

FreqPS contains FreqPosting

One FreqPS may contain many frequency postings.

Clients

See “InvertedIndex gets (by Term) FreqPS” on page 5-48.

Public Member Functions

constructor

Input

InvertedTermInfo*

A pointer to the termInfo for the term to be posted.

BitArray* An array of deleted documents. Used to ensure those not physically deleted yet are not given as postings.

IStorage* The storage in which to place the postings. Storage must be open.

destructor

Next

Input

FreqPosting* Returns the address of the next posting in this slot.

Index Category

Output

bool True if one returned, false if not.

Description

Copies the next FreqPosting from the stream into the provided FreqPosting. Returns NULL at the end of the stream.

Usage

```
for (bool go = fps->Next(&posting); go ;
      go = fps->Next(&posting))
```

FreqTerm

Struct

Struct
Header: HighFreqTerms.h

Data

uint32	freq
	The number of times the term appears.
IATerm*	term
	The term.

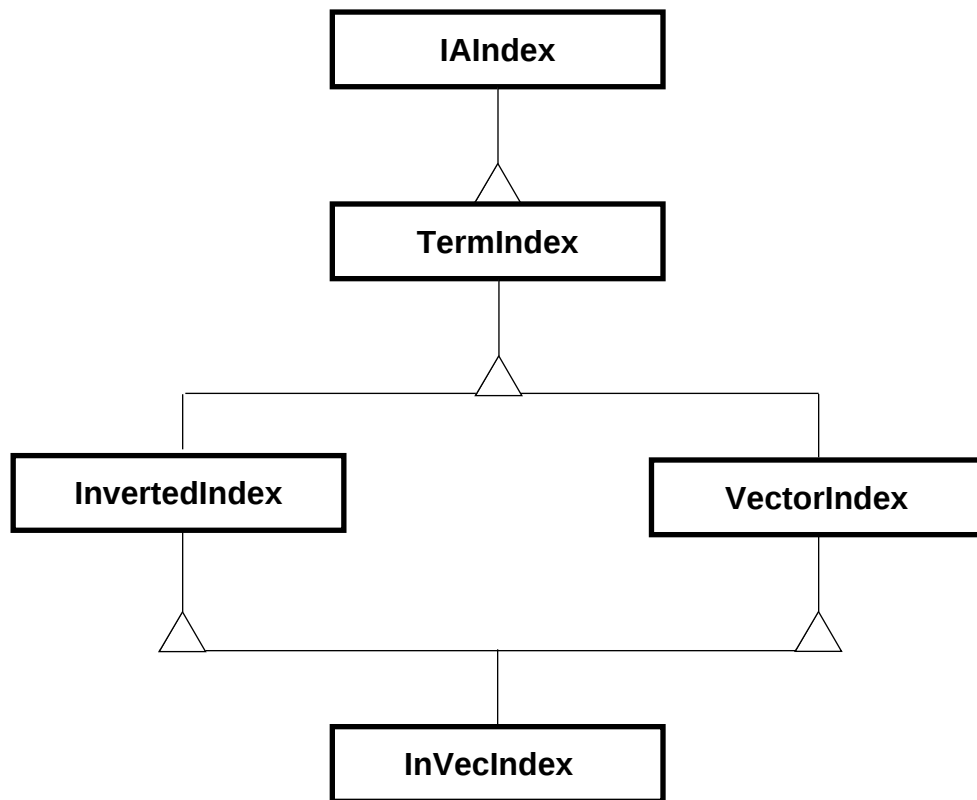
IAIndex

Class

Heading: IAIndex.h

Hierarchy

Abstract base class.

Figure 5-11 Index hierarchy

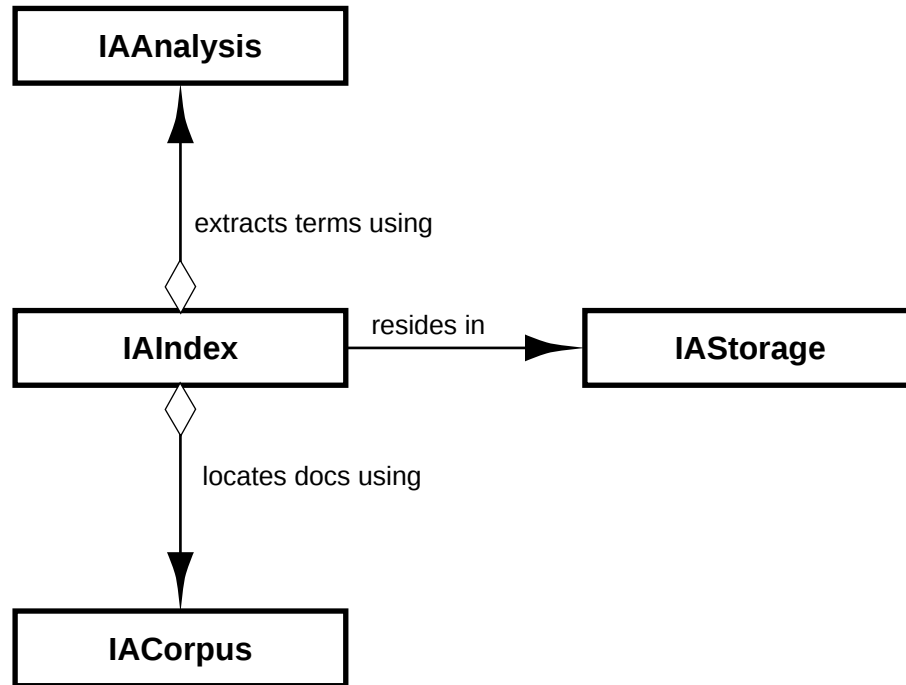
Description

IAIndex is the base class of all the index classes. It controls the establishment of a corpus, and the location of terms through analysis. It manages the storage for the index, corpus, and analysis used.

The relationships with the analysis and corpora are stored in the index root block. This block is stored upon Initialize() and FinishUpdate(). It is restored on an open. Subclasses can add information to this root block by implementing the protected functions RootSize(), StoreRoot() and RestoreRoot().

Relationships

Figure 5-12 Overview of the index relationships



IAIndex locates docs using IACorpus

One index contains one and only one corpus.

IAIndex locates terms using IAAnalysis

One index contains one and only one analysis.

IAIndex is stored in IAStorage

One index is stored in one storage for its root, but allocates and stores items in many storages internally.

Public Member Functions

constructor

Input

IStorage* storage
A pointer to the storage in which to place the index.

IACorpus* corpus
A pointer to the associated corpus.

IAAnalysis* analysis
A pointer to the analysis to be used to extract terms.

uint32 indexType
The index type constant.

IABlockID indexRoot
The block id of the root. Default is nil; the root will be allocated if not defined.

destructor

Virtual
Deletes corpus and analysis.

AddDoc

Pure virtual.

Input

IADoc* document
A pointer to the IADoc for the document that is to be added to the index.

Description

Adds a document to the index. Also passes control of the IADoc object to the index. The IADoc will be deleted automatically when the index is deleted.

Index Category

▲ **WARNING**

AddDoc assumes the document does not already exist in the index. If you are not sure if the document has been indexed, use IsDocIndexed() to check. If you wish to replace the previous index information for a document that has changed, you must delete the document and then re-add it.

Usage

```
anIAIndex.AddDoc(&anIADoc);
```

Compact

Virtual.

Attempts to compact the index; removes deleted items.

▲ **WARNING**

If documents have been deleted, the index must be compacted before storage is committed.

Usage

```
anIAIndex.Compact();
```

DeleteDoc

Pure virtual.

Input

IADoc*	document A pointer to the IADoc for the document that is to be removed from the index.
--------	---

Description

Marks a document as deleted. Prevents reporting of postings to this document.

Usage

```
anIAIndex.DeleteDoc(&anIADoc);
```

Index Category

Notes

Does not delete the caller's IADoc from memory.

Flush

Virtual.

Flushes all changes and disables further changes. Typically called just before `aStorage->Commit()`.

Usage

```
anIIndex.Flush();
```

GetAnalysis

Access method for IADoc member data.

Output

`IAAnalysis*analysis`
A pointer to the analysis used to extract terms.

GetCorpus

Access method for IADoc member data.

Output

`IACorpus* corpus`
A pointer to the corpus used to interface with the physical documents.

GetDocCount

Pure virtual.

Index Category

Output

uint32 numberDocuments
The total number of documents indexed.

Usage

```
for (uint32 i = 0; i < indexCount; i++)
    docCount += indices[i]->GetDocCount();
```

GetDocIterator

Pure virtual.

Output

IADocIterator*
A pointer to an iterator over the documents indexed.

Description

Returns an iterator over all the documents indexed. See IADocIterator.

Usage

```
IADocIterator* anIterator= anIIndex.GetDocIterator();
```

GetDocIterator(IADoc* start)

Pure virtual.

Input

IADoc* start
A pointer to the IADoc of the document that you wish to be the first in the series accessed by the iterator.

Output

IADocIterator*
A pointer to an iterator over the documents indexed. Iterator is positioned at IADoc if it exists in the index. If not, it is positioned at the document which would logically follow that doc should it exist.

Index Category

Usage

```
IADocIterator* anIterator = index.GetDocIterator(startDoc);
```

GetIndexRoot

Access method for IAIndex member data.

Output

IABlockID **indexRoot**
The block ID of the index root storage block.

GetIndexType

Access method for IAIndex member data.

Output

uint32 **indexType**
A constant that indicates which type (e.g., inverted) of index this is.

GetIndexTypes

Input

IAIndexTypes* types
The struct of the type codes for the index.

Description

Accesses the types (storage, corpus, etc.) of an index. May be called at any time. See “IAIndexTypes” on page 5-43.

Usage

```
IAIndexTypes types;  
GetIndexTypes(&types);
```

GetMaxDocumentSize

Access method for IAIndex member data.

Output

uint32	maxDocSize
	The current maximum document size.

Notes

See SetMaxDocumentSize().

GetQueryAnalysis

Virtual.

Description

Gets the IAAnalysis to be used for processing queries on this index. If a preferred analysis has been set (by SetPreferredAnalysis), then it will be returned. If a preferred analysis has not been set, then GetQueryAnalysis will default to whatever analysis was specified at index construction.

Output

IAAnalysis*analysis
A pointer to the analysis to be used for processing queries.

GetPreferredAnalysis

Access method for IAIndex member data.

Output

IAAnalysis*analysis
A pointer to a preferred analysis.

GetStorage

Access method for IASIndex member data.

Output

IASStorage* storage

A pointer to the storage for the index, corpus, and analysis blocks.

Initialize

Virtual.

Description

Initializes a new empty index in a new empty storage.

Usage

```
anIASIndex.Initialize();
```

IsDocIndexed

Pure virtual.

Input

IADoc* doc

A pointer to the IADoc of the document that might be indexed.

Output

bool

True if the document is indexed; False if the document is not indexed.

Usage

```
anIASIndex.isDocIndexed(&doc);
```

Merge

Pure virtual.

Input

IAIndex**	indexes An array containing the indexes to be merged into this index.
uint32	indexCount The number of indexes in the array.

Description

Merges an array of indexes into an index. The index, corpus and analysis classes must be the same for all indices. The indices must be disjoint — no documents may be indexed in more than one index. If a document is in more than one index, Merge will throw an exception with code `IndexDocAlreadyIndexed` ('VIAI').

Usage

```
destIndex.Merge(indexes, indexCount);
```

Open

Virtual.

Description

Opens an existing index. By default, Calls `Open()` on the storage, corpus and analysis. The index must have been constructed with the exact same types as that in the storage.

Usage

```
anIAIndex.Open();
```

RenameDoc

Pure virtual.

Index Category

Input

IADoc*	oldName
	A pointer to the IADoc containing the old name.
IADoc*	newName
	A pointer to a new IADoc containing the new name.

Description

Updates the indexes references to an (unchanged) document. Only the names need be present in the IADocs. The new name must not already exist in the index. The index must be opened for update.

Usage

```
anIAIndex.RenameDoc(oldDoc, newDoc);
```

SetMaxDocumentSize

Access method for IAIndex member data.

Input

uint32	maxDocSize
	The number of unique words to be used as the maximum document size.

Notes

In order to prevent the potential for unbounded memory usage, indexes stop processing documents after this number of unique index terms has been reached. (Note that “unique index terms” is not the same as “unique words.” For example, if a stemmer is being used, then all forms of a word with the same stem will be treated as a single unique index term.) The default is 2000. If your application will be working with very large documents, you should set this higher.

SetPreferredAnalysis

Access method for IAIndex member data.

Input

IAAnalysis*	analysis
	An analysis to be used for processing queries.

Index Category

Description

Sets the analysis module that will be used to process new queries on the index. See `GetQueryAnalysis()`.

Usage

```
IAAnalysis *myNewAnalysis = new SimpleAnalysis();  
anIAIndex.SetPreferredAnalysis(myNewAnalysis);
```

Update

Virtual.

Description

Uses the corpus iterator to add new documents and delete expired documents. Simple applications should be able to maintain an index with just this method; complex applications will need the more fine-grained control of other methods. See “Updating an Index” on page 5-14.

The index must be open, but no update may be started.

▲ WARNING

It is the responsibility of the corpus iterator to return documents in the correct order. If documents are out of order, Update may either miss some documents that require adding, or reindex unchanged documents.

Usage

```
anIAIndex.Update();
```

IAIndexTypes

Struct

Struct

Header: `IAIndex.h`

Functions

constructor

Usage

```
IAIndexTypes types;
```

constructor(uint32 s, uint32 c, uint32 a, uint32 i)

Input

uint32	s	The storage type.
uint32	c	The corpus type.
uint32	a	The analysis type.
uint32	i	The index type.

Usage

```
IAIndexTypes types(storageType, corpusType,  
                    analysisType, indexType);
```

Equal

Input

IAIndexTypes*	other	The structure of types to which this might be equal.
---------------	-------	--

Output

bool	True if equal; false if not.
------	------------------------------

Index Category

Note
Null types are not considered equal.

Usage

```
if (myTypes.Equal(theirTypes)) printf("Match\n");
```

Data

```
uint32    analysisType;  
uint32    corpusType;  
uint32    indexType;  
uint32    osSetType;  
uint32    storageType;
```

InVecIndex

Class

Header: InVecIndex.h

Hierarchy

Public subclass of both InvertedIndex and VectorIndex.

Description

This combines both inverted and vector index.

Public Member Functions

constructor

Input

```
IAStorage* storage  
           A pointer to the storage in which to place the index.  
IACorpus*  corpus  
           A pointer to the associated corpus.  
IAAnalysis* analysis  
           A pointer to the analysis to be used to extract terms.
```

Index Category

```
uint32      type = InVecIndexType
            The index type. InVecIndexType is a constant 'I&V2'.
IABlockID   indexRoot = NULL
            The block id of the root. Will create one if not supplied.
```

Class InvertedIndex

Header: InvertedIndex.h

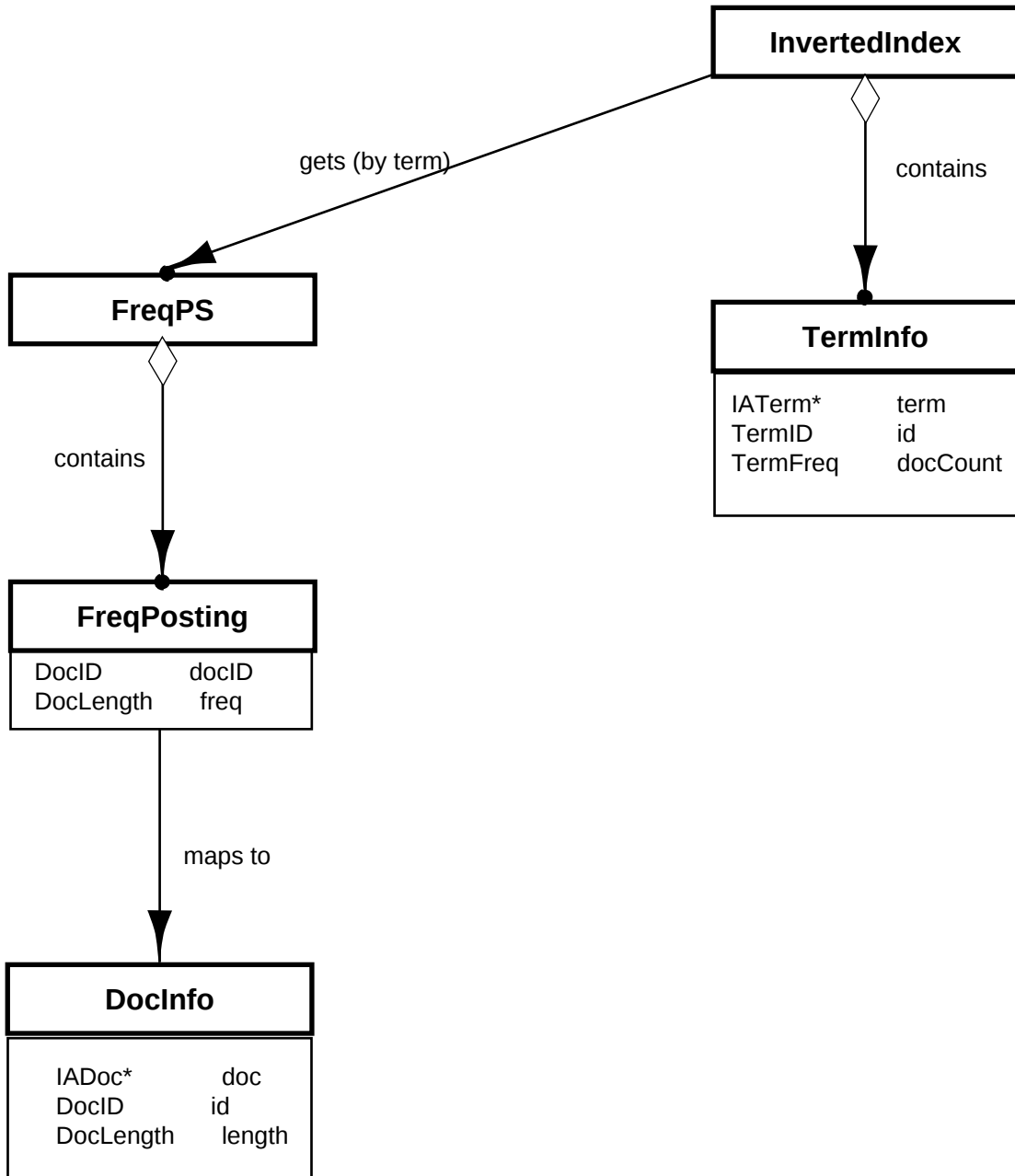
Hierarchy

Public subclass of TermIndex. Virtual.

Description

An inverted index keeps tracks of terms and points to which documents they are in.

Relationships

Figure 5-13 Inverted index overview

InvertedIndex contains TermInfo

One InvertedIndex contains many TermInfo

InvertedIndex gets (by Term) FreqPS

One Inverted Index creates and gets many FreqPS, one per term.

Public Member Functions

constructor

Input

IAStorage* storage
A pointer to the storage in which to place the index.

IACorpus* corpus
A pointer to the associated corpus.

IAAnalysis* analysis
A pointer to the analysis to be used to extract terms.

uint32 type = InvertedIndexType
The index type. Constant is 'Inv6'.

IABlockID indexRoot = NULL
The block id of the root. Will create one if not supplied.

destructor

Compact

See "IAIndex.Compact" on page 5-48.

GetDeletedDocCount

Output

`uint32 numberDeletedDocs`
The number of deleted documents since the last Compact.

Usage

```
uint32 numberDeletedDocs = GetDeletedDocCount();
```

GetFreqPostings

Input

`TermInfo* termInfo`
Pointer to the `termInfo` for the term.

Output

`FreqPS *`
A pointer to the frequency postings.

Usage

```
FreqPS* fps = index.GetFreqPostings(ti);
```

Initialize

See “`IAIndex.Initialize`” on page 5-49.

Open

See “`IAIndex.Open`” on page 5-49.

TermIndex

Header: `TermIndex.h`

Class

Index Category

Hierarchy

Public subclass of IIndex.

Description

A term index is a general abstraction of any index which maintains a relationships between terms and documents. It contains the general structures and functions for creating and maintaining these indexes. TermIndex, although instantiable, serves as the base class for InvertedIndex and VectorIndex.

Public Member Functions

constructor

Input

IAStorage*	storage	A pointer to the storage in which to place the index.
IACorpus*	corpus	A pointer to the associated corpus.
IAAnalysis*	analysis	A pointer to the analysis to be used to extract terms.
uint32	type = TermIndexType	The index type. Constant is 'Ter2'.
IABlockID	indexRoot = NULL	The block id of the root. Will create one if not supplied.

destructor

AddDoc

See "IIndex.AddDoc" on page 5-34.

AddDoc(IADoc* document, DocID* returnID);

Input

IADoc* document
A pointer to document to add.

Input/Output

DocID* returnID
The document id; AddDoc assigns this and returns its address here.

Description

The same as AddDoc(IADoc* document), except the document ID is returned. Must have a StartUpdate before calling.

DeleteDoc

See “IAIndex.DeleteDoc” on page 5-35.

Flush

See “IAIndex.Flush” on page 5-36.

GetDocCount

Output

DocID
The number of documents in the index.

Usage

```
totalDocCount += i[j]->GetDocCount();
```

GetDocInfo

Input

IADoc* **document**
The pointer to the document whose information is needed.

bool **ignoreError = false**
If false, will throw an Invalid document exception if the doc info is not found. If true, DocInfo will be nil if no info found.

Output

DocInfo*
Pointer to the document information.

Usage

```
DocInfo* info = GetDocInfo(doc, true);
```

GetDocInfoIterator

Output

IAOrderedStorableIterator*
Pointer to an iterator over the set of document information.

Usage

```
IAOrderedStorableIterator* docs = index.GetDocInfoIterator();
```

GetDocInfoIterator(IADoc* start);

Input

IADoc* **start**
Pointer to an IADoc containing the document name at which you wish this iterator to start.

Output

IAOrderedStorableIterator*
Pointer to an iterator over the set of document information.

Index Category

Description

Same as `GetDocInfoIterator`, only the iterator will be positioned at the `DocInfo` for the input document and continue from there. If the document is not found in the set, the iterator will be positioned at the document which would logically follow this one.

GetDocIterator

See “`IAIndex.GetDocIterator`” on page 5-37.

GetDocIterator(IADoc* start)

See “`IAIndex.GetDocIterator(IADoc* start)`” on page 5-37.

GetFlushProgressData

Access method for `TermIndex` member data.

Output

`void*` `pdata`
A pointer to a the item whose progress is being reported.

GetFlushProgressFn

Access method for `TermIndex` member data.

Output

`FlushProgressFn*flushProgressFn`
A pointer to the function used for progress callbacks.

GetFlushProgressFreq

Access method for `TermIndex` member data.

Index Category

Output

clock_t	flushProgressFreq The number of clock ticks between progress reports. Uses ANSI clocks_per_sec.
---------	---

GetIDDoc

Input

DocID id	The id of the document.
----------	-------------------------

Output

IADoc*	The IADoc for the document.
--------	-----------------------------

Notes

The index must be open. This function will fail with an Invalid Doc ID exception if the document ID does not exist.

Usage

```
StringDoc* doc = (StringDoc*)index.GetIDDoc(posting.docID);
```

GetIDTerm

Input

TermID id	The id of the term.
-----------	---------------------

Output

IATerm*	The IATerm for the term.
---------	--------------------------

Notes

The index must be open. This function will fail with an Invalid Term ID exception if the term ID does not exist.

Index Category

Usage

```
IATerm* term = index.GetIDTerm(component->termID);
```

GetMaxDocID

Output

DOCID

The next available document ID.

Notes

This is also used as the maximum count; that is the largest number of documents including those which have been deleted but not actually physically purged.

Usage

```
DocID max = index->GetMaxDocID();
```

GetMaxTermID

Output

TermID

The next available term ID.

Notes

This is also used as the maximum count; that is the largest number of terms including those which have been deleted but not purged.

Usage

```
TermID maxTermID = index->GetMaxTermID();
```

GetTermCount

Output

TermFreq The number of terms in the index.

GetTermInfo

Input

IATerm* **term**
The pointer to the term whose information is needed.

Output

TermInfo*
Pointer to the term information.

Notes

The index must be open. This will fail with an invalid term exception if the term does not exist.

Usage

```
TermInfo* i = indices[j]->GetTermInfo(entry->term);
```

GetTermInfoIterator

Output

IAOrderedStorableIterator* **iterator**
Pointer to an iterator over the set of term information.

Usage

```
IAOrderedStorableIterator* terms =  
    index->GetTermInfoIterator();
```

GetTermInfoIterator(IATerm* start);

Input**IATerm***

The term at which the iterator should be positioned.

Output**IAOrderedStorableIterator***

Pointer to an iterator over the set of term information.

Description

Same as GetTermInfoIterator() except the iterator will be positioned at the input term. If this term is not in the set, the iterator will be positioned at the term which would logically follow.

Initialize

See "IAIndex.Initialize" on page 5-40.

IsDocIndexed

See "IAIndex.IsDocIndexed" on page 5-40.

Merge

See "IAIndex.Merge" on page 5-41.

Open

See "IAIndex.Open" on page 5-41.

RenameDoc

See “IAIndex.RenameDoc” on page 5-41.

SetFlushProgressData

Access method for TermIndex member data.

Input

`void*` `pdata`
The item whose progress is to be reported.

SetFlushProgressFn

Access method for TermIndex member data.

Input

`FlushProgressFn*fn`
The function to be called for progress status during `AddDoc()`, `DeleteDoc()`, and `Flush()`.

SetFlushProgressFreq

Access method for TermIndex member data.

Input

`clock_t` `freq`
The number of clock ticks between progress reports. Uses ANSI `clocks_per_sec`.

Protected Member Functions

GetBytesForUpdate

Access method for TermIndex member data.

Output

uint32 **bytesForUpdate**
 Number of bytes allocated for certain indexing data structures.

Notes

BytesForUpdate is an internal value used as a hint to help allocate data structures efficiently for indexing. See SetBytesForUpdate().

SetBytesForUpdate

Access method for TermIndex member data.

Input

uint32 **bytesForUpdate**
 Number of bytes to allocate for certain indexing data structures.

Notes

BytesForUpdate is an internal value used as a hint to help allocate data structures efficiently for indexing. The default is 1,000,000. Larger values will cause the indexing application to use more memory, but it will process changes to the index in larger chunks and therefore increase its performance.

TermInfo

Header: TermIndex.h

Class

Hierarchy

Public subclass of IAOrderedStorable. See “IAOrderedStorable” on page 10-14.

Index Category

Description

Term Info is the basic information about a term as it relates to this index.

Relationships

TermInfo contains IATerm

One termInfo contains one and only one term.

Clients

See "InvertedIndex contains TermInfo" on page 5-48.

See "TFComponent maps to TermInfo" on page 5-63.

Public Member Functions

constructor()

constructor(IATerm* term, TermID termID)

Input

IATerm*	term
	The term.
TermID	termID
	The ID for the term.

destructor

Deletes the term.

DeepCopy

See “IAStorable.DeepCopy” on page 10-28.

Equal

See “IAOrderedStorable.Equal” on page 10-15. TermInfo equals another TermInfo if the term equals the other term.

GetDocumentCount

Access method for TermInfo member data.

Output

TermFreq	docCount
	The number of documents in which the term occurs.

GetTerm

Access method for TermInfo member data.

Output

IATerm*	term
	The term in question.

GetTermID

Access method for TermInfo member data.

Output

TermID	id
	The ID of the term.

LessThan

See “IAOrderedStorable.LessThan” on page 10-16. TermInfo is sequenced by term.

Restore

See “IAStorable.Restore” on page 10-28.

SetDocumentCount

Access method for TermIndex member data.

Input

TermFreq	docCount
	The number of documents in which the term occurs.

SetTerm

Access method for TermIndex member data.

Input

IATerm*	term
	The term in question.

Store

See “IAStorable.Store” on page 10-30.

StoreSize

See “IAStorable.StoreSize” on page 10-29.

TFComponent

Class

Header: TFVector.h

Description

A TFComponent is the relationship between a document and a term it contains. See Figure 5-14 on page 5-69.

Relationships

TFComponent maps to TermInfo

One TF component maps to one and only one term info.

This mapping is indirect; TFComponent contains a TermID, which uniquely points to a single TermInfo.

Clients

See “TFVector contains TFComponent” on page 5-64.

Data

TermID	termID The TermID
TermFreq	freq The frequency of that term.

TFVector

Class

Header: TFVector.h

Hierarchy

Base Class.

Description

The stream of TFComponents; the vehicle for obtaining the components of a document.

Relationships

TFVector contains TFComponent

One TFVector contains many TFComponents.

Clients

See “VectorIndex gets (by doc) TFVector” on page 5-69.

Public Member Functions

constructor(DocLength length)

Input

DocLength length

The number of components in the vector.

destructor

ComponentsRead

Input

IAInputBlock* input

The allocated and opened input block for the components

ComponentsSize

Output

IABlockSize

The block size used for component storage

ComponentsWrite

Input

`IAOutputBlock* output`
The allocated output block for the storage

GetComponents

Access method for TFVector member data.

Output

`TFComponent* components`
An array of TFComponents.

GetDocumentLength

Access method for TFVector member data.

Output

`DocLength length`
The number of components in the vector (i.e. the number of unique indexed terms in the document).

SetComponents

Access method for TFVector member data.

Input

`TFComponent* components`
An array of TFComponents.

SetDocumentLength

Access method for TFVector member data.

Index Category

Input

DocLength length

The number of components in the vector (i.e. the number of unique indexed terms in the document).

Class VectorDocInfo

Header: VectorIndex.h

Hierarchy

Public subclass of DocInfo. See “DocInfo” on page 5-25.

Description

DocInfo for a vector index. This allows the storage of DocInfo as a block.

Client

See “VectorIndex contains VectorDocInfo” on page 5-70.

Public Member Functions

GetVectorBlockID

Access method for VectorDocInfo member data.

Output

IABlockID vectorBlock

The BlockID of the block where the vector is stored.

SetVectorBlockID

Access method for VectorDocInfo member data.

Index Category

Input

IABlockID **vectorBlock**
 The BlockID of the block where the vector is stored.

constructor()

constructor(IADoc* document, DocID docID)

Input

IADoc* The document.
DocID The ID for the document.

DeepCopy

See “IAStorable.DeepCopy” on page 10-28.

Restore

See “IAStorable.Restore” on page 10-28.

Store

See “IAStorable.Store” on page 10-30.

StoreSize

See “IAStorable.StoreSize” on page 10-29.

VectorIndex

Class

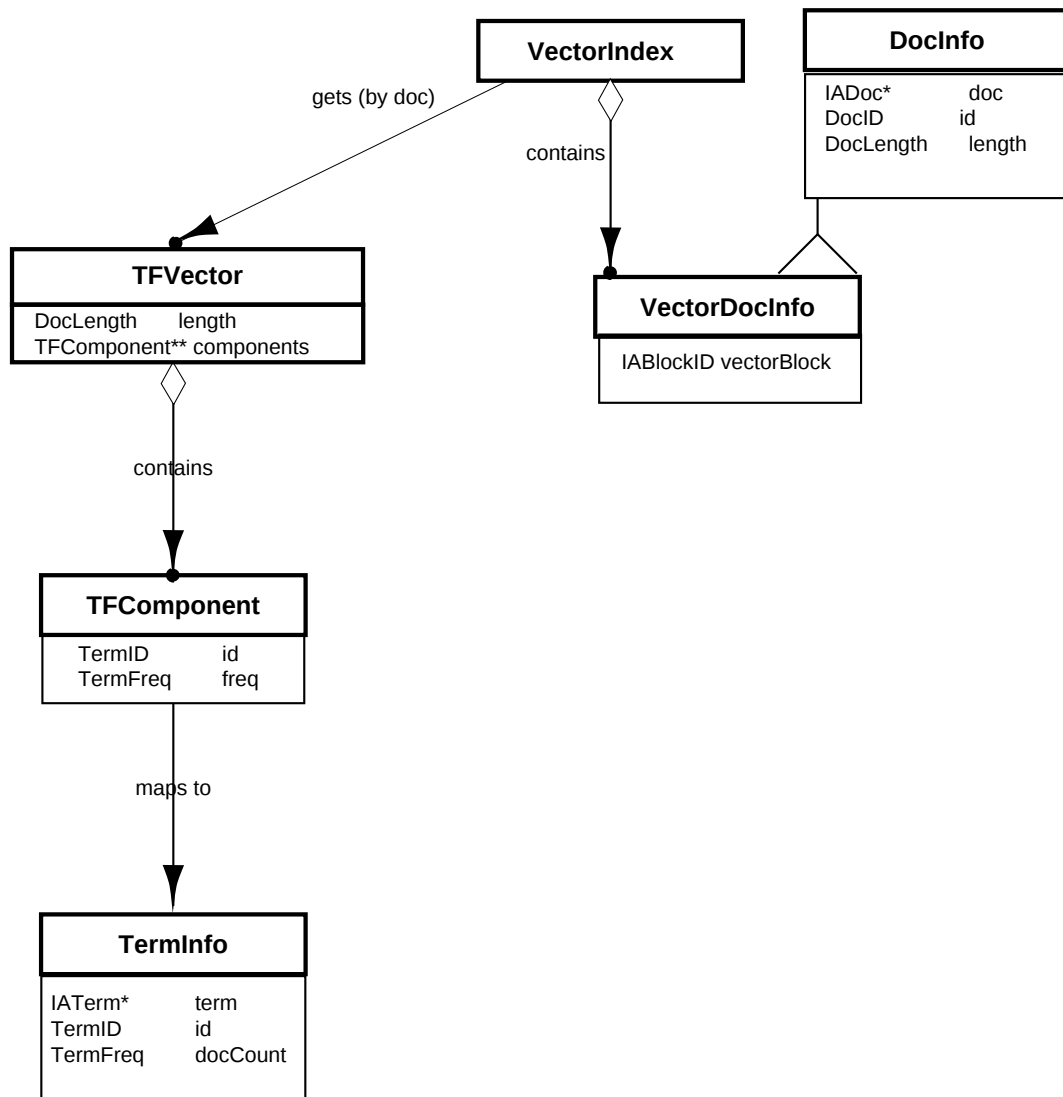
Header: VectorIndex.h

Index Category

Hierarchy

Public subclass of TermIndex. Virtual. See “TermIndex” on page 5-49.

Relationships

Figure 5-14 Vector index overview**VectorIndex gets (by doc) TFVector**

A vector index create and gets many TFVectors, one per document.

VectorIndex contains VectorDocInfo

One vector index contains many VectorDocInfo, one per document.

Public Member Functions

constructor

Input

IAStorage* storage
A pointer to the storage in which to place the index.

IACorpus* corpus
A pointer to the associated corpus.

IAAnalysis* analysis
A pointer to the analysis to be used to extract terms.

uint32 indexType=VectorIndexType
The index type constant.

IABlockID indexRoot
The block id of the root. Default is nil; the root will be allocated if not defined.

destructor

GetTFVector

Virtual.

Input

IADoc* doc
Pointer to the document.

Output

TFVector*
The vector.

Index Category

Usage

```
TFVector* vector = aVectorIndex.GetTFVector(di->doc);
```

Class Utilities

GetHighFreqTerms

Header: HighFreqTerms.h

Input

TermIndex* index
A pointer to the index.

uint32* nTerms
The desired number of terms. Returns the actual number (n or less) found.

Output

FreqTerm*
A pointer to an array of freqTerms.

Notes

Results should be freed with `IAFreeArray()`.

Usage

```
FreqTerm* results = GetHighFreqTerms(&index, &resultCount);
```

IAReadIndexTypes

Input

IStorage* storage
A pointer to the storage in which to place the index.

IABlockID indexRoot
The block id of the root.

IAIndexTypes* types
A pointer to the initialized index types structure. This will be returned with the types read.

Index Category

Usage

```
IAIndexTypes indexTypes;  
IAReadIndexTypes(storage, indexRoot, &indexTypes);
```

Typedefs

DocID

A unique identifier for a document.

Type

TermID

Header

TermIndex.h

DocLength

The number of terms in a document

Type

TermFreq

Header

TermIndex.h

FlushProgressFn

FlushProgressFn(float percent void* data); the progress function to be used when building an index.

Type

void

Header

TermIndex.h

TermFreq

The number of times a term appears in a document.

Type

uint32

Header

TermIndex.h

TermID

A unique identifier for a term.

Type

uint32

Header

TermIndex.h

Index Category

Extern Data

extern "C"

Order function so that arrays of TFComponent can be sorted by qsort.

extern uint32TMaxDocSize

The maximum number of tokens indexed per doc.

Documents longer than this are currently truncated. Default, set in TermIndex, is 2000.

Index Category

Constants

```
const uint32 InvertedIndexType='Inv6'  
            InvertedIndex.h  
const uint32 InVecIndexType='I&V2'  
            InVecIndex.h  
const uint32 TermIndexType='Ter2'  
            TermIndex.h  
const uint32 VectorIndexType='Vec4'  
            VectorIndex.h
```

Index Exceptions and Error Handling

Errors That May Occur when Working with Indexes

These are errors that sometimes occur when working with indexes. IAT reports errors as exceptions. The explanations suggest possible causes of the exception in the context of working with indexes. See the exception code under its category for more detailed information.

You can tell the category of an exception by its prefix: VA: accessor, VC: corpus, VS, storage, VI, index. VTWN is a general exception code.

VCHE Validation of File Names (HFS Error)

There is no validation of the input storage and folder names. You must ensure they exist or could exist under that name in the path specified.

VIIV Incompatible Index Type

You can get this exception when you try to establish an existing index as a corpus or analysis type different from the one used in its creation.

VSPB Incompatible Corpus Type (Store Past Block End)

One way to read past the end of a block is to update an existing index that was created with a text folder corpus with a text file not in that corpus. The update will work; however, when you try to access the index after updating, you may get this error. You may only have documents within the folder for a text folder corpus.

VTWN Incomplete Index

If the index was being built under the same storage name, and that build failed, you get this exception.

Exceptions Thrown by Index Classes

VIAI

IndexDocAlreadyIndexed. Something has been renamed to a document already in the index, or there is an attempt to merge an index with one which already contains the document name.

Header

IAIndex.h

VIAO

IndexAlreadyOpen. The Initialize or Open functions were called when the index was already initialized or open.

Header

IAIndex.h

VIDN

IndexDocNotIndexed. The document is not found.

Header

IAIndex.h

VIIV

Index Invalid. The types of an index opened from storage do not equal the types of the constructed index.

Header

IAIndex.h

Index Category

VINO

IndexNotOpen. One of these functions have been called without opening or initializing the index first.

IAIndex:

- Compact
- StartUpdate

Inverted Index

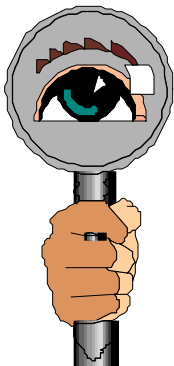
- Get Deleted Doc Count

TermIndex

- GetDocCount
- GetDocInfo
- GetDocInfoIterator
- GetIDDoc
- GetIDTerm
- GetMaxDocID
- GetMaxTermID
- GetTermCount
- GetTermInfo
- GetTermInfoIterator

Header

IAIndex.h



Accessor Category

Choosing an Accessor Type	3
Query Logic	4
Query Analysis	5
Common Operations	6
Building an Accessor	6
Answering Queries	7
Answering a Simple Ranked Query	8
Answering a Query by Example	11
Answering a Boolean Query	14
Describing a Document	16
Finding Related Words	18
Accessor Class Category Reference	20
Header Files in the Accessor Category	20
Class Specifications	22
IAAccessor	22
IAHit	26
IAProgressReport	29
InVecAccessor	32
InvertedAccessor	33

Accessor Category

RankedAccessor	36
RankedHit	43
RankedProgress	45
RankedQueryDoc	46
TWComponent	47
TWVector	48
VectorAccessor	51
Typedefs	53
Constants	53
Accessor Exceptions and Error Handling	54

Accessor Category

This accessor category contains the classes required to access IAT indexes. Accessors can be used to search for documents in the index and get information about them. An accessor provides the means to locate documents by query or to determine which documents are similar to each other. All searches are done through an accessor.

Searches may vary based on the accessor type (Vector, Inverted, or InVec), the item used to search (a text query or one or more sample documents), and the way the query is to be interpreted (ranked or Boolean).

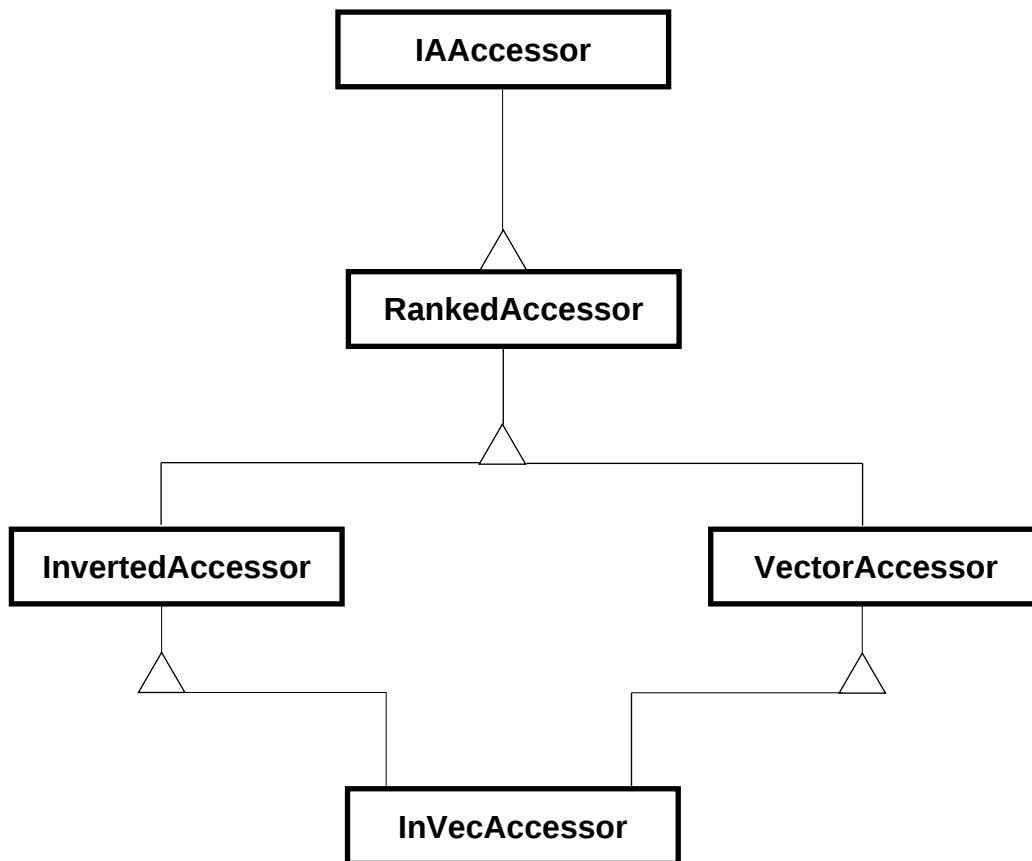
With ranked searching, the user describes his or her information need with an arbitrary list of words (which may be a sample of natural language text, or even a question), and the system determines which documents best satisfy that need. Because there is no single “right answer,” the system computes a score for each potentially matching document which represents its estimated relevance. The documents are then returned with the scores, sorted from highest relevance to lowest.

With Boolean searching, users describe their information need with a logical expression consisting of words connected by the Boolean operators AND, OR, and NOT.* While Boolean searching is useful for some specialized tasks, studies consistently show that users get better search results with ranked searching.

Choosing an Accessor Type

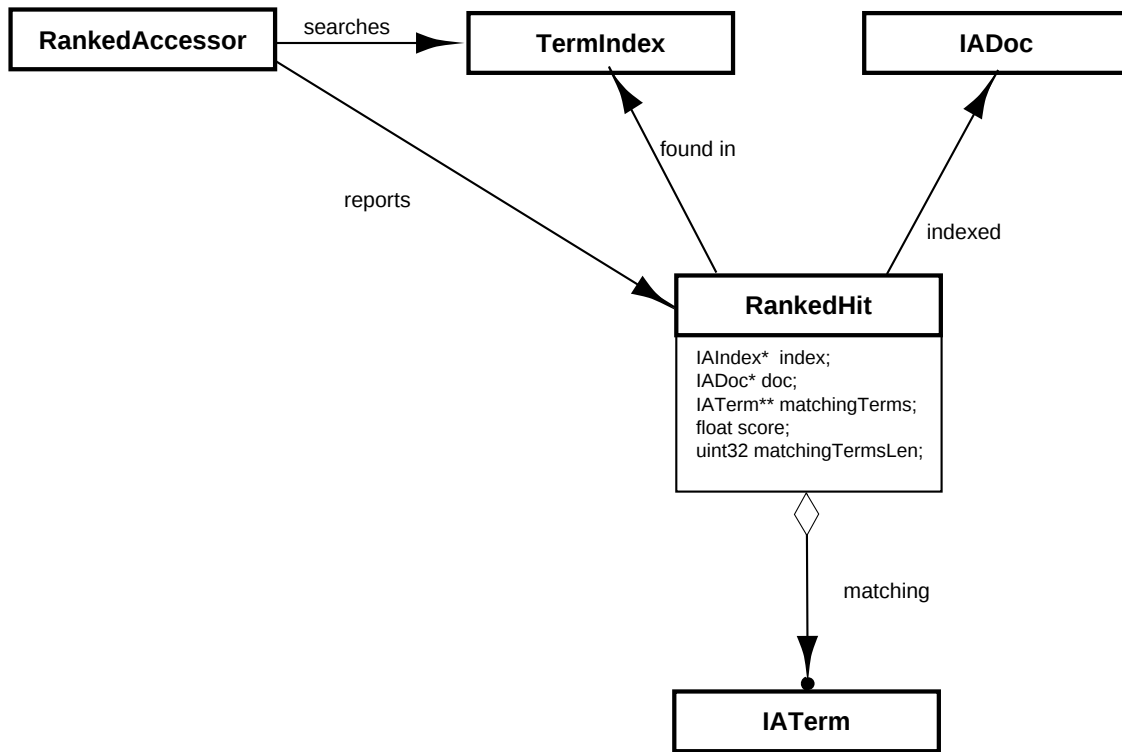
As seen in Figure 6-1, the inheritance tree for an accessor parallels that of an index. Generally you will wish to use the accessor that matches the index type used. Although you may use an inverted accessor or a vector accessor with an InVecIndex, the InVec accessor takes most advantage of the InVecIndex features.

* In information retrieval, the Boolean NOT operator is shorthand for BUT NOT. For example, the Boolean query “dog NOT beagle” would find all items containing the word “dog” except those also containing the word “beagle.”

Figure 6-1 Accessor inheritance tree

Query Logic

The primary work of the accessor is to search the index in answer to queries. When it does this it reports hits, which match terms to documents. Figure 6-2 shows the abstract classes used in a query.

Figure 6-2 Query logic

One accessor may access many indexes. It reports an array of RankedHits in answer to a query.

Query Analysis

Queries, like documents, must be analyzed by an analysis module in order to extract the terms to be searched. By default, the query is processed using the same analysis as was used when indexing the documents. However, there may be cases where developers may want to allow their applications to offer different analysis options at search time.

For example, a collection of documents may be indexed using an analysis that uses all the words in the text. An application may then offer the users the option of automatically removing stop words (like “the”) from the query text. This would require the use of a different analysis for queries.

To use a different analysis for queries, call the `IAIndex` function `SetPreferredAnalysis` (page 5-42) after opening the index to be search with the accessor.

Common Operations

Building an Accessor

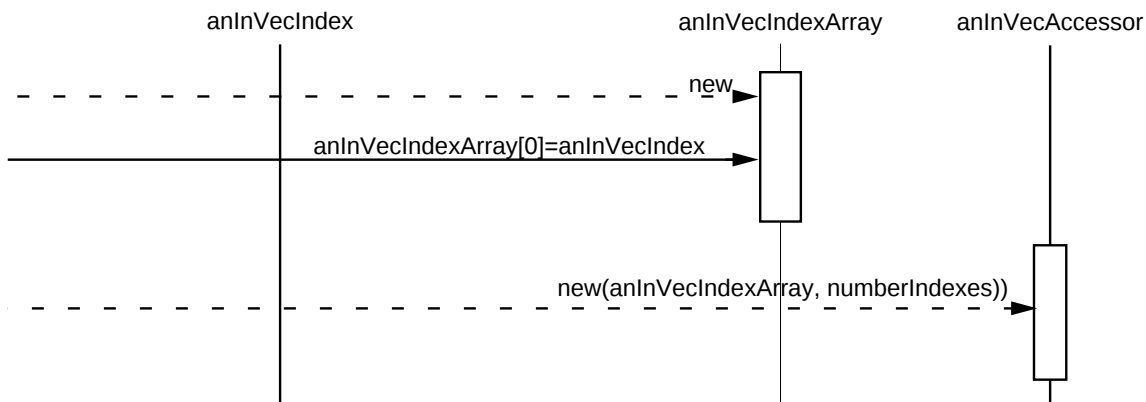
An accessor is built for a set of indexes. The indexes should be established in storage and opened (generally opened for read only access unless there is some other use than accessing them).

Note

Changing the index (by adding or deleting documents) after opening an accessor will cause the open accessor to become invalid.

The example below builds an accessor for a single index.

Figure 6-3 Interaction diagram to build an accessor



Listing 6-1 Build an inverted vector accessor

```

// establish an index in storage
// See ("Establish an existing index" on page 5-13)

// make an array of indexes to use (can be > one)
const uint32 numberIndexes = 1;
InvertedVectorIndex* anInVecIndexArray[numberIndexes]; // make indexes
anInVecIndexArray[0] = &anInVecIndex;

// create the accessor
InVecAccessor anInVecAccessor(anInVecIndexArray, numberIndexes);
  
```

Answering Queries

Accessors are the means by which indexes can be searched. The search request takes the form of a query. There are two general types of queries against an index:

- a simple ranked query, which, given a string of text, locates individual terms in that text. The accessor then finds documents which have those terms.
- a Boolean query, which, given a Boolean expression, locates documents that satisfy that expression.
- a query by example, which, given one or more documents that have been indexed and their index, will locate the most similar documents.

Preparing for a Query

To prepare for a query, you must establish these items:

- the maximum number of documents to retrieve(`numberDocs`). RecipeSwap, for example, chooses to limit the number of recipes to give the patron to ten, so the number of Ranked Hits is limited to ten
- the maximum number of matching terms (`numberTermsPerDoc`) to show per query. Those terms which contribute most to the document being retrieved are used.

You can then establish an array for the storage of the resulting RankedHits. This array will have as many members as the maximum number of documents to list.

Reporting Progress

You may wish to have a report of progress as the search goes on. You may establish a progress reporting function and pass that address to the search. The function you develop will use the RankedProgress type as an input parameter. See “RankedProgress” on page 6-45.

You provide the frequency of progress to the query; this parameter is the number of `clock_t` between reports.

Note

Set a frequency of progress greater than 0. If you set the frequency of progress to zero, you will have very frequent (about 1 ms) progress reporting. This will make the accessor intolerably slow.

Listing 6-2 is an example of a primitive progress reporting function.

Listing 6-2 Report search progress

```
bool ReportQueryProgress(RankedProgress* progress, void* data) {
    #pragma unused (data)
    printf("Percent Searched: %4.1f \n", progress->percent);
    return false;
}
```

Answering a Simple Ranked Query

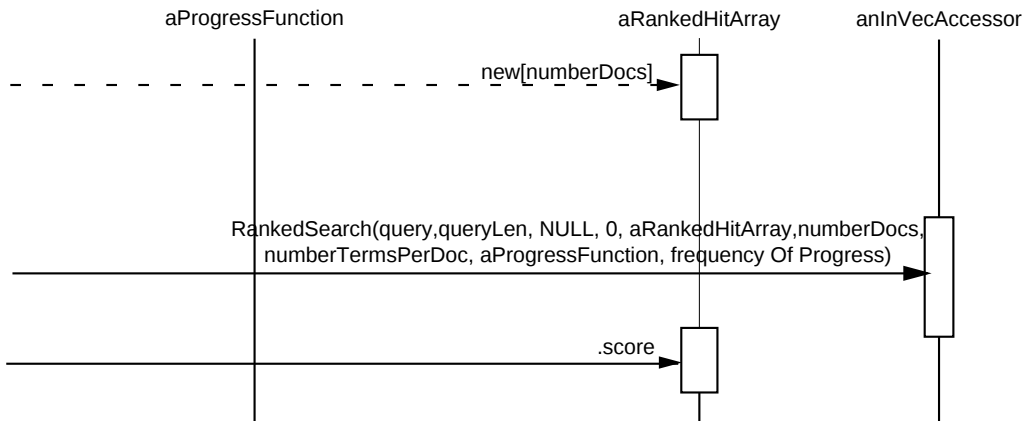
Listing 6-3 is an example of a program which parses a simple string of terms and matches that string against the terms in an inverted index. The results show which documents have any of the terms, what their score is, and which terms they contain. The terms are sorted according to how much they contributed to this document being retrieved. Figure 6-4 shows an output display of this query.

Figure 6-4 Output from a simple ranked query

```

Query: Swiss, spinach, onion
searching: 0.0
searching: 100.0
search time: 00 hours, 00 minutes and 00 seconds.
5 hits
1.00 : spinach-pizza [ spinach]
0.89 : quiche05 [ spinach onion]
0.82 : quiche10 [ spinach]
0.65 : quiche11 [ spinach]
0.63 : quiche09 [ swiss onion]
  
```

Figure 6-5 Interaction diagram for a simple ranked query



Listing 6-3 Answer a simple ranked query

```
// create the accessor
// (see "Build an inverted vector accessor" on page 6-6)
InVecAccessor anInVecAccessor(anInVecIndexArray, numberIndexes);
anInVecAccessor.Initialize();

// set up display of results
const    numberTermsPerDoc = 4; // max terms to show/doc
const    numberDocs = 5; // max docs to list
RankedHit* aRankedHitArray[NumberDocs];
clock_t  frequencyOfProgress = clocks_per_sec / 2; // tics btwn

// get the query and display it
char* query = GetQuery(); // application provided function
printf("Query: %s\n", query);

// do the search
uint32 numberHitsFound = accessor.RankedSearch(
    (byte*)query, strlen(query), // query string and length
    NULL, 0, // no query by example doc parameters
    aRankedHitArray, numberDocs, numberTermsPerDoc, // final results
    &ReportQueryProgress, frequencyOfProgress, NULL);

// report the results
DisplayResults(aRankedHitArray, numberHitsFound); // see Listing 6-4
```

Listing 6-4 Display search results

```

void DisplayResults (RankedHit** aRankedHitArray,
                    uint32 numberHitsFound) {
    // display number of hits
    printf("%lu hits\n", numberHitsFound);
    // show the documents that hit and their relevance score
    for (uint32 i = 0; i < numberHitsFound; i++) {
        // show document name and relevance score
        RankedHit* aRankedHit= aRankedHitArray[i];
        printf("%5.2f : ", aRankedHit->GetScore());
        // see Listing 6-5 for PrintDocName.
        PrintDocName(aRankedHit->GetDocument());

        // show the top n terms/document(unless none)
        if (aRankedHit->GetMatchingTermsLen()){
            printf(" [");
            for (uint32 j = 0;
                j < aRankedHit->GetMatchingTermsLen(); j++) {
                printf(" %s",
                    aRankedHit->GetMatchingTerms()[j]->GetData());
            }
            printf("]");
        }
        printf("\n");
        delete aRankedHit;
    }
    return;
}

```

Listing 6-5 Get and print a document name

```

void PrintDocName(IADoc* doc) {
    uint32 docNameLength;
    char* docName = (char*)doc->GetName(&docNameLength);
    printf("%s", docName);
    delete[] docName;
}

```

Answering a Query by Example

This type of search uses one or more documents as the query. It locates other documents similar to the query documents, and scores their relevance to the sample document. The result, is a ranked list of documents found.

Similarity is measured based on matching the statistical distribution of terms in the example and hit documents. Roughly speaking, two documents will have high similarity scores if they use many of the same words.

When you have a new document and you wish to see if a similar document exists already, you must add the new document to the index before you can use it as a query. Once you have added the document to the index, you can use it to create a `RankedQueryDoc`. This is a pairing of the document with the index it is in. If you do not wish to keep the new document in the index, you can delete it after the search.

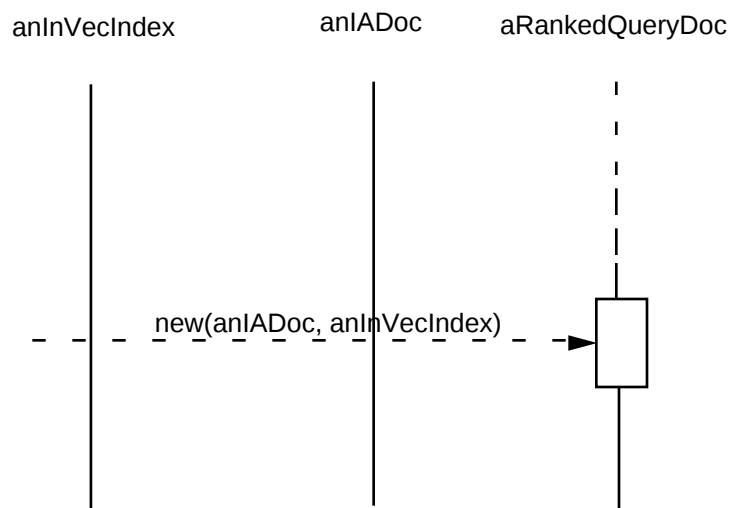
Figure 6-6 Sample output from a query by example

```
adding Mom's Chocolate Decadence
flushing: 0.00
flushing: 80.00
flushing: 100.00
Mom's Chocolate Decadence
searching: 0.0
searching: 50.0
searching: 121.9 into
searching: 150.0
5 hits
1.00 : Mom's Chocolate Decadence [ rasp choc genache decadance chambord]
0.94 : Chocolate Decadence [ rasp choc genache decadance chambord]
0.18 : Hazelnut Cheesecake [ paris gene min ken crust]
0.13 : Cheesecake Collection [ gelatin crust tbs oreos chocolate]
0.13 : White Choc Cheesecake02 [ gelatin mousse chocolate bittersweet pipe]
deleting Mom's Chocolate Decadence
```

In this example, the second item has almost the same score as the first, suggesting that this is probably the same recipe as the sample document.

Note

The sample document must be in an index that is contained in the array used when constructing the accessor. If you are matching to a different index, create the accessor with both indexes.

Figure 6-7 Interaction diagram for creating a RankedQueryDoc

Sample Code for Query-by-example

Listing 6-6 Find documents matching example document

```
// Set up the example doc with its index
StringPtr docName = "\pMom's Chocolate Decadence";
HFSTextFolderCorpus* anHFSTextFolderCorpus =
    (HFSTextFolderCorpus*)recipeIndex.GetCorpus();
HFSTextFolderDoc* anEmailDoc = new HFSTextFolderDoc(
    (HFSTextFolderCorpus*)anHFSTextFolderCorpus, 0, docName, 0);

recipeIndex.AddDoc((HFSTextFolderDoc*)anEmailDoc->DeepCopy());
recipeIndex.Flush();

// Convert example doc to a RankedQueryDoc
RankedQueryDoc aRankedQueryDoc(anEmailDoc, &recipeIndex);
PrintDocName(anEmailDoc); // see Listing 6-5
printf("\n");

// Set up query results
const    numberTermsPerDoc = 5; // amount to show/document
const    numberDocs = 5; // top n docs to show

// Create the query structure
RankedHit* aRankedHitArray[numberDocs]; // array of hits
clock_t frequencyOfProgress = 30; // time btwn progress rpts

// Do the search
InVecAccessor anInVecAccessor(anInVecIndexArray, numberIndexes);
anInVecAccessor.Initialize();

char* query = NULL; //Null the term search parmeters
uint32 numberOfExamples = 1; //There is one sample doc
uint32 numberOfHitsFound = anInVecAccessor.RankedSearch(
    (byte*)query, strlen(query), // query is nil
    &aRankedQueryDoc, numberOfExamples, // feedback doc params
    aRankedHitArray, numberDocs, numberTermsPerDoc, // results
    &DemoRankedProgress, frequencyOfProgress, NULL);

// Report the results and remove new doc
DisplayResults(aRankedHitArray, numberOfHitsFound); // Listing 6-4
recipeIndex.DeleteDoc(anEmailDoc);}
```

Answering a Boolean Query

This type of search uses a Boolean expression as the query. It locates documents that satisfy the Boolean expression. For example, the expression “cat AND dog” would be satisfied by only those documents containing *both* the word “cat” and the word “dog.” The result, as with a ranked query, is a ranked list of documents with relevance scores.

The actual characters to be interpreted as Boolean operators can be set by the application using the member functions `SetBooleanAndOperator` (default is ‘&’), `SetBooleanOrOperator` (default is ‘|’), and `SetBooleanNotOperator` (default is ‘!’). In addition, Boolean expressions can be nested, and the nesting operators can be set using the functions `SetBooleanLeftFence` and `SetBooleanRightFence`; left and right parentheses are the defaults.

Figure 6-8 Sample output from a Boolean query

```

accessor initialization: 00 hours, 00 minutes and 01 seconds.
Query: (chocolate & cinnamon) ! liqueur
searching: 0.0
searching: 100.0
search time: 00 hours, 00 minutes and 00 seconds.
5 hits
1.00 : Cinn Choc Chip Cookies
0.76 : Vegan Choc Pudding
0.75 : Chocolate Cheesecake06
0.69 : White Choc Fruitcake
0.68 : Chocolate Pecan Pudding

```

Sample Code for Boolean Query

Listing 6-7 Find documents satisfying Boolean expression

```
const uint32 kMaxDocuments = 5;

InVecAccessor accessor(indices, nIndices); // make appropriate accessor
accessor.Initialize();

RankedQueryDoc rqd1[kMaxDocuments];

printf("Query: %s\n", query); // display query

RankedHit* results[kMaxDocuments]; // allocate array for results
uint32 resultCount = 0;

resultCount = ((InvertedAccessor*)accessor)->RankedSearchBoolean
    ((byte*)query, strlen(query), // query string
    results, kMaxDocuments, // result array
    &DemoRankedProgress, 30, NULL); // progress args

DisplayResults(&results, resultCount); // see Listing 6-4
```

Describing a Document

In addition to searching, accessors can also provide a list of words that best describe the document. You may use the `GetDocTopic` function for this purpose.

In this context, “best describe” means “most differentiate from other documents in the index.” So, for example, if your collection of documents consists of 500 items all about Pizza, the word “pizza” would probably not be one of the words, since it does not help distinguish one document from another. Instead, you would expect to see words like “vegetarian” or “pepperoni.” The terms are sorted from most to least descriptive.

Figure 6-9 Sample output from describing a document

```
Terms Describing Document: Ice Cream Xmas Pudding
australian
cherry
ice
rising
cherries
marshmallows
christmas
```

Sample Code for Describing a Document

Listing 6-8 Find the words that best describe a document

```
DemoAccessor accessor(indices, nIndices); // make appropriate accessor
accessor.Initialize();

IATerm* results[MaxResultCount]; // allocate array for results

// For this example, get most recently added document
DocID maxID = index.GetMaxDocID();
IADoc* doc = index.GetIDDoc(maxID - 1);
IADeleteOnUnwind delDoc(doc);

RankedQueryDoc rqd(doc, &index);
uint32 resultCount = accessor.GetDocTopic(&rqd, results, MaxResultCount,
                                           &DemoRankedProgress, 30, NULL);
printf("Terms Describing Document: ");
PrintDocName(doc);
printf("\n");
for (uint32 i = 0; i < resultCount; i++) {
    IATerm* term = results[i];
    printf(" %s\n", term->GetData());
    delete term;
}
```

Finding Related Words

Accessors can also provide a list of words related to a given word.

In this context, “related to” means “commonly occurs in the same contexts.” In a collection of recipes, the word “pepperoni” might have “pizza,” “crust,” and “sausage” as some of its related terms. The terms are sorted from most to least related.

Figure 6-10 Sample output from finding related words

```
accessor initialization: 00 hours, 00 minutes and 02 seconds.  
searching: 0.0  
searching: 0.0  
searching: 100.0  
searching: 0.0  
searching: 100.0  
Terms related to: dijon  
mustard  
tblsp  
seed  
herb  
age  
honey  
grainy  
instructions  
vinegar
```

Sample Code for Finding Related Words

Listing 6-9 Find the words related to a given word

```
// Make accessor & storage, etc -- see 6-1

InVecIndex index(storage);
index.Open();

const uint32 nIndices = 1;
InVec* indices[nIndices]; // make indices
indices[0] = &index;

InVecAccessor accessor(indices, nIndices); // make appropriate accessor
accessor.Initialize();

IATerm* termResults[MaxResultCount]; // allocate array for results

uint32 resultCount = accessor.GetTermsRelated(
    (byte*)query, strlen(query), // query string
    termResults, MaxResultCount, // result array
    &DemoRankedProgress, 30, NULL); // progress args

printf ("Terms related to: %s\n", query);
printf ("related terms:");
for (uint32 i = 0; i < resultCount; i++) {
    IATerm* term = termResults[i];
    printf (" %s\n", term->GetData());
    delete term;
}
```

Accessor Class Category Reference

Header Files in the Accessor Category

IAAccessor

IAAccessor
IAHit
IAProgressReport

InVecAccessor

InVecAccessor

InvertedAccessor

InvertedAccessor

RankedAccessor

RankedAccessor
RankedHit
RankedProgress
RankedQueryDoc

TWVector

TWComponent
TWVector

VectorAccessor

VectorAccessor

IARound

Class Specifications

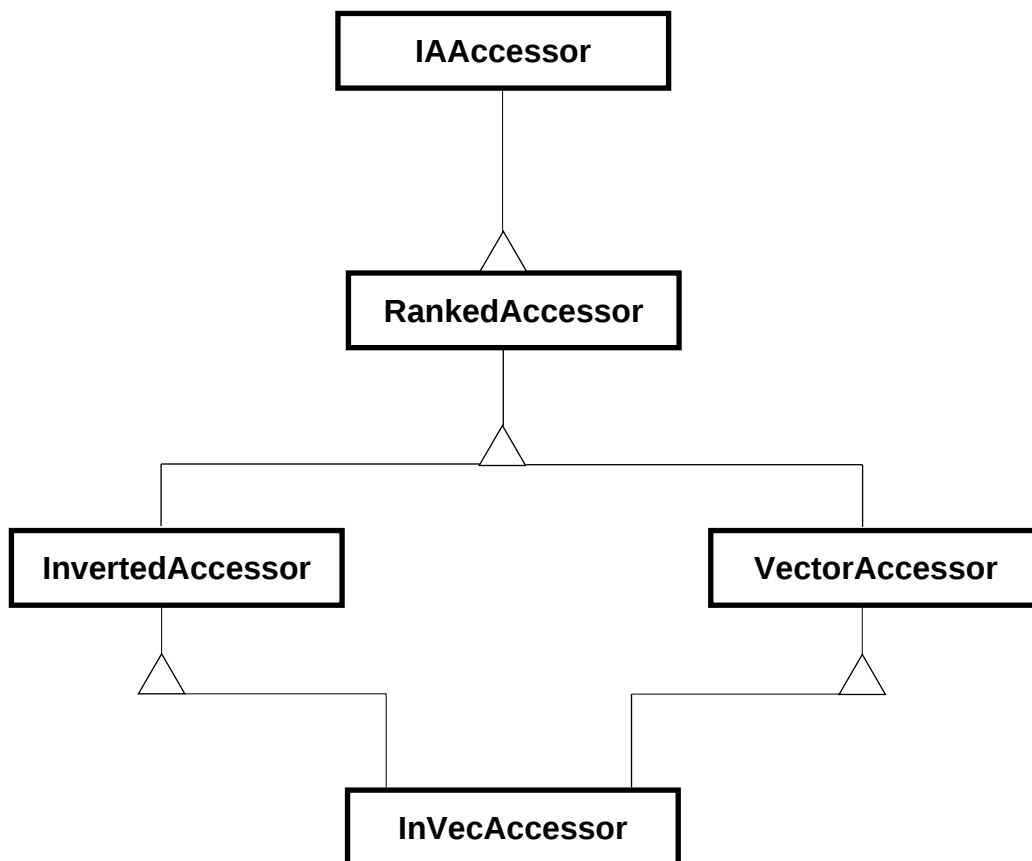
Class IAAccessor

Header: IAAccessor.h

Hierarchy

Abstract Base Class.

Figure 6-11 Accessor inheritance



Description

IAccessor is the base class for providing access (such as a search) to an index.

Relationships

IAAccessor accesses IAIndexes

One accessor may access many indexes.

IAAccessor reports an IAHit

One accessor may report many hits

IAAccessor reports status with IAProgressReport

One accessor uses one Progress Report

Public Member Functions

constructor

Input

IAIndex**	The array of indexes to search.
uint32	The number of indexes in the array.

destructor

Virtual.
Note that this destructor does not delete the indexes.

GetAccessorType

Access method for IAAccessor member data.

Output

uint32	accessorType
	The type of the accessor used.

GetIndexCount

Access method for IAAccessor member data.

Output

uint32	indexCount
	The number of indexes being accessed by the accessor.

GetIndices

Access method for IAAccessor member data.

Output

IAIndex**	indices
	An array of the indexes being accessed by the accessor.

Initialize

Virtual.

Input

IAStorage*	storage for the accessor; default is NULL; generally the named block in the index is used.
IABlockId	the blockID for the storage, if provided. Default is 0.

Accessor Category

Notes

Must be called after constructor but before any other methods. This is required because constructors cannot call virtual methods.

If the accessor initialization data was stored, it is restored. Otherwise it is computed from scratch, which may be slow for large indexes.

Usage

```
accessor->Initialize();
```

IsInitializationValid

Input

IStorage*

storage for the accessor; default is NULL; generally the named block in the index is used.

IBlockId

the blockID for the storage, if provided. Default is 0.

Output

bool

True if the initialization has been stored.

Notes

Checks to see if the accessor initialization data was stored. If not, Initialize() will be slow. Calling StoreInitialization will initialize and store so subsequent initializations will be faster.

SetAccessorType

Access method for IAAccessor member data.

Input

uint32

accessorType

The type of the accessor used.

SetIndexCount

Access method for IAAccessor member data.

Input

`uint32` `indexCount`
The number of indexes to be accessed by the accessor.

▲ **WARNING**

The index count must match the number of indexes in the array passed by the `SetIndices` method.

SetIndices

Access method for IAAccessor member data.

Input

`IAIndex**` `indices`
An array of indexes to be accessed by the accessor.

StoreInitialization

Input

`IASStorage*`
storage for the accessor; default is `Nil`; generally the named block in the index is used.

`IABlockId`
the blockID for the storage, if provided. Default is 0.

Notes

Both initializes and stores the accessor initialization data. Accessor should not be initialized when this is called.

Class IAHit

Header: `IAAccessor.h`

Hierarchy

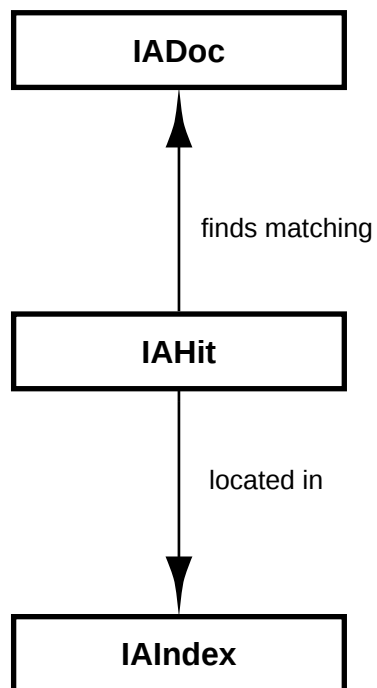
Base class.

Description

Base class for search results. A hit is the connection between a document that matches the query and its index.

Relationships

Figure 6-12 IAHit relationships



IAHit finds matching IADoc located in IAIndex

An IAHit identifies one doc in one index.

Clients

See “IAAccessor reports an IAHit” on page 6-23.

Public Member Functions

constructor

Input

IAIndex* index
Pointer to the index containing the matching document.

IADoc* doc
Pointer to the matching document.

destructor

Virtual.

GetDocument

Access method for IAHit member data.

Output

IADoc* doc
A pointer to the document found by the hit.

GetIndex

Access method for IAHit member data.

Output

IAIndex* index
A pointer to the index in which the hit document was found.

SetDocument

Access method for IAHit member data.

Input

IADoc*	doc
	A pointer a document to be used by a hit.

SetIndex

Access method for IAHit member data.

Input

IAIndex*	index
	A pointer to the index in the hit document resides.

IAProgressReport

Class

Header: IAAccessor.h

Hierarchy

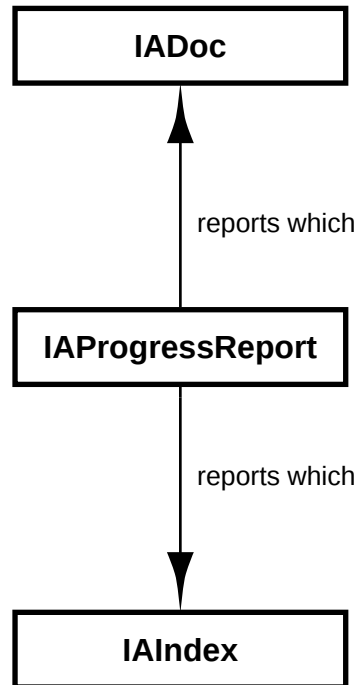
Base class.

Description

Base class for progress reports. Progress reports are used by user-provided progress functions.

Relationships

Figure 6-13 IAProgressReport relationships



IAProgressReport reports which IAIndex is being processed

One progress report reports on one index at a time.

IAProgressReport reports which IADoc is being processed

One progress report reports on one document.

Clients

See “IAAccessor reports status with IAProgressReport” on page 6-23.

Public Member Functions

GetDocument

Access method for IAProgressReport member data.

Output

IADoc*	doc
	A pointer to the document whose progress is being reported. NULL if not applicable.

GetIndex

Access method for IAProgressReport member data.

Output

IAIndex*	index
	A pointer to the index whose progress is being reported. NULL if not applicable.

GetPercent

Access method for IAProgressReport member data.

Output

float	percent
	A number between 0.0 and 100.0 inclusive, representing the percent of the search (or other access operation) completed.

SetDocument

Access method for IAProgressReport member data.

Accessor Category

Input

IADoc*	doc	A pointer to the document whose progress is being reported.
--------	-----	---

SetIndex

Access method for IAProgressReport member data.

Input

IAIndex*	index	A pointer to the index whose progress is being reported.
----------	-------	--

SetPercent

Access method for IAProgressReport member data.

Input

float	percent	A number between 0.0 and 100.0 inclusive, representing the percent of the search (or other access operation) completed.
-------	---------	---

Class InVecAccessor

Header: InVecAccessor.h

Hierarchy

Public subclass of InvertedAccessor and VectorAccessor. See “InvertedAccessor” on page 6-33 and “VectorAccessor” on page 6-51.

Description

Accelerates searches on InVec indexes.

Public Member Functions

constructor

Input

IAIndex** indexes
 The array of indexes to search.

uint32 indexCount
 The number of indexes in the array.

uint32 type = InVecAccessorType

destructor

RankedSearch

See “RankedAccessor.RankedSearch” on page 6-41.

InvertedAccessor

Class

Header: InvertedAccessor.h

Hierarchy

Public subclass of RankedAccessor. See “RankedAccessor” on page 6-36.

Description

An inverted accessor accesses inverted indexes for searches for terms.

Public Member Functions

constructor

Input

`InvertedIndex** indexes`
The array of indexes to search.

`uint32 indexCount`
The number of indexes in the array.

`uint32 InvertedAccessorType`
The type of accessor. Constant 'Inv0'.

destructor

RankedSearch

See “RankedAccessor.RankedSearch” on page 6-41.

RankedSearchBoolean

Input

`byte* booleanTextQuery`
The query text, in the form of a Boolean expression.

`uint32 textQueryLen`
The number of bytes in the query text.

`RankedHit** results`
An array in which to place the resulting hits.

`uint32 resultLen`
The maximum number of hits desired.

`RankedProgressFn* progressFn`
A pointer to the progress function to use.

`clock_t progressFreq`
The number of ticks between progress reports.

`void* appData`
A user-supplied parameter to the progress reporting function.

Accessor Category

Output

uint32

The number of hits found matching the Boolean expression.

Notes

The Boolean expression assumes the Boolean operators set by the accessor functions SetBooleanAndOperator, SetBooleanOrOperator, SetBooleanNotOperator, SetBooleanLeftFence, and SetBooleanRightFence.

Protected Member Functions

GetInvertedRankedQueryMaxTerms

Access method for InvertedAccessor member data.

Output

uint32

value

The maximum number of terms to be included in the query.

Notes

See notes for SetInvertedRankedQueryMaxTerms.

GetInvertedRankedQueryMinTerms

Access method for InvertedAccessor member data.

Output

uint32

value

The number of terms below which no query truncation will occur.

Notes

See notes for SetInvertedRankedQueryMinTerms.

SetInvertedRankedQueryMaxTerms

Access method for InvertedAccessor member data.

Input

uint32 value
The maximum number of terms to be included in the query.

Notes

InvertedAccessors optimize performance by truncating extremely long queries. All terms are used up to a certain minimum truncation threshold (set by SetInvertedRankedQueryMinTerms). Then the query is truncated by discarding terms whose weights are so low that they will have little or no effect on the results of the search. Finally, only the top N (highest weighted) remaining terms are kept. This function sets the value of N. The default is 50. For no query truncation, set both values to 0xFFFFFFFF.

SetInvertedRankedQueryMinTerms

Access method for InvertedAccessor member data.

Input

uint32 value
The number of terms below which no query truncation will occur.

Notes

InvertedAccessors optimize performance by truncating extremely long queries. All terms are used up to a certain minimum truncation threshold (set by this function). The default is 10. Then the query is truncated by discarding terms whose weights are so low that they will have little or no effect on the results of the search. Finally, only the top N (highest weighted) remaining terms are kept, where N is the value set by the function SetInvertedRankedQueryMaxTerms. For no query truncation, set both values to 0xFFFFFFFF.

Class RankedAccessor

Header: RankedAccessor.h

Hierarchy

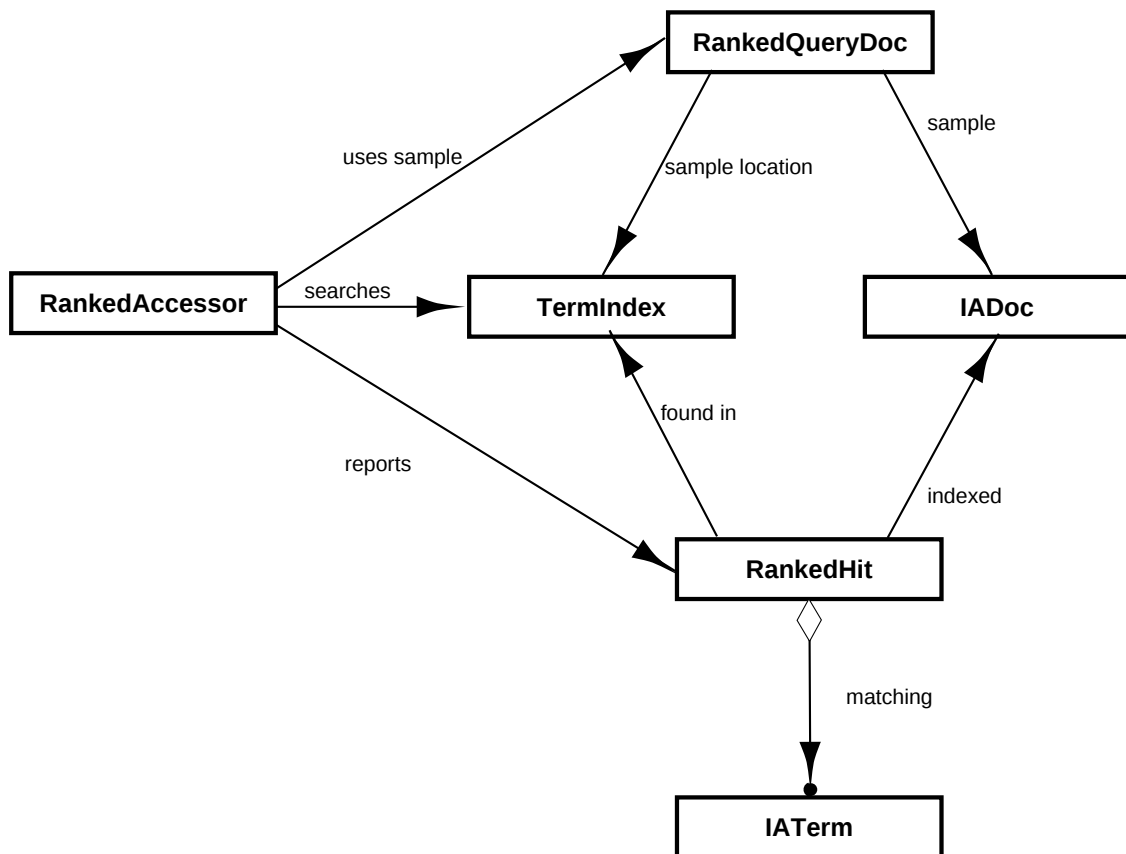
Public subclass of IAAccessor. See “IAAccessor” on page 6-22.

Description

An abstract class that searches any type of index and ranks the results.

Relationships

Figure 6-14 RankedAccessor relationships



RankedAccessor uses sample RankedQueryDoc

One Ranked Accessor uses one RankedQueryDoc per query by example, but may use many.

RankedAccessor searches TermIndex

OneRankedAccessor may search many TermIndexes.

RankedAccessor reports RankedHit

One RankedAccessor will report many RankedHits per query.

Public Member Functions

constructor

Input

`IAIndex** indexes`
A pointer to an array of indexes to be used in the search.

`uint32 indexCount`
The number of indexes.

`uint32 type`
the constant indicating the kind of accessor.

destructor

GetDocTopic

Virtual.

Input

`RankedQueryDoc* doc`
The sample document and its index.

`IATerm** results`
The terms that characterize the document.

`uint32 resultLen`
The maximum number of terms to report.

Accessor Category

RankedProgressFn* progressFn

A pointer to the progress reporting function to use.

clock_t progressFreq

The frequency of reporting progress.

void* appData

User-provided parameter to the progress reporting function.

Output

uint32

The number of terms found in the document (actual number of results).

Notes

Identifies the terms which best represent the document's content. Orders them by weights indicating their importance.

Usage

```
RankedQueryDoc rqd(doc, &index);
uint32 resultCount = accessor.GetDocTopic
    ( &rqd, results, MaxResultCount,
      &DemoRankedProgress, 30, NULL);
```

HitEqual

Virtual

Input

IAIndex* index1

The index containing the first hit.

const IADoc* doc1

The document containing the first hit.

IAIndex* index2

The index containing the second hit.

const IADoc* doc2

The document containing the second hit.

Output

bool

True, these are the same documents; false, they are not.

Accessor Category

Notes

Determines merging of hits; current implementation returns equal if IADocs are equal. Used to determine whether hits from two different indexes are actually the same document.

HitLessThan

Virtual

Input

IAIndex* index1
The index containingf the first hit.

const IADoc* doc1
The document containing the first hit.

IAIndex* index2
The index containing the second hit.

const IADoc* doc2
The document containing the second hit.

Output

bool
Returns True if the doc is less than the second doc.

IsHit

Virtual

Input

IAIndex* index
The index.

const IADoc* doc
The document in the index.

Output

bool
Always true .

Accessor Category

Notes

This is provided to allow subclasses to filter hits by other criteria, like date. It must be overridden to be useful.

MergeHits

Virtual

Input

`const RankedHit* hit1`

The first of two hits on the same document.

`const RankedHit* hit2`

The second of two hits on the same document.

Output

`RankedHit`

That hit with the highest score of the two.

Notes

Merges hits that are `HitEqual()` into one hit — default copies higher scoring. This may occur when a document is indexed in more than one index.

RankedSearch

Pure virtual.

Input

`byte* textQuery`

The query text.

`uint32 textQueryLen`

The number of bytes in the query text.

`RankedQueryDoc* docQuery`

A pointer to an array of sample documents and their index.

`uint32 nDocs`

The number of sample documents.

`RankedHit** results`

An array in which to place the resulting hits.

Accessor Category

uint32 resultLen
The maximum number of hits desired.

uint32 matchingTermsLen
The maximum number of terms to report in ranked hits.

RankedProgressFn* progressFn
A pointer to the progress function to use.

clock_t progressFreq
The number of ticks between progress reports.

void* appData
A user-supplied parameter to the progress reporting function.

Output

uint32
The number of hits found.

RankedSearch

Pure virtual.

Input

IADocText* query
The query object.

RankedQueryDoc* docQuery
A pointer to an array of sample documents and their index.

uint32 nDocs
The number of sample documents.

RankedHit** results
An array in which to place the resulting hits.

uint32 resultLen
The maximum number of hits desired.

uint32 matchingTermsLen
The maximum number of terms to report in ranked hits.

RankedProgressFn* progressFn
A pointer to the progress function to use.

clock_t progressFreq
The number of ticks between progress reports.

void* appData
A user-supplied parameter to the progress reporting function.

Accessor Category

Output

uint32

The number of hits found.

Notes

Applications may wish to use this variant of RankedSearch when they need the query text to be part of a real object. For example, they may require some metadata — such as a language code — to be passed along with the query text.

RankedHit

Class

Header: RankedAccessor.h

Hierarchy

Public subclass of IAHit. See “IAHit” on page 6-26.

Relationships

Ranked Hit contains matching IATerm

1 hit may match many terms.

Client

See “RankedAccessor reports RankedHit” on page 6-38.

Public Member Functions

constructor

Input

IAIndex* index

A pointer to the index containing the document

IADoc* doc

A pointer to the document.

Accessor Category

`float score`
The score of the hit.

`IATerm** terms`
The array of matching terms.

`uint32 termLen`
The number of matching terms.

destructor

Deletes matchingTerms.

DeepCopy

`const`

Output

`RankedHit*`
a copy of this ranked hit, including matching terms.

GetMatchingTerms

Access method for RankedHit member data.

Output

`IATerm** matchingTerms`
An array of the top scoring terms in the intersection of the document with the query.

GetMatchingTermsLen

Access method for RankedHit member data.

Output

`uint32 matchingTermsLen`
The number of matching terms in the intersection of the document with the query.

GetScore

Access method for RankedHit member data.

Output

float	score
	The relevance score of the hit. The relative strength of this document's match to the query. Scaled from 0.0 to 1.0.

SetScore

Access method for RankedHit member data.

Input

float	score
	A relevance score to be assigned to the hit.

RankedProgress

Class

Header: RankedAccessor.h

Hierarchy

Public subclass of IAProgressReport. See "IAProgressReport" on page 6-29.

Description

An extension of the progress report that adds reporting by the current term being processed.

Public Member Functions

constructor

GetTerm

Access method for RankedProgress member data.

Output

IATerm*	term
	When non-NULL, reports the term currently being processed.

SetTerm

Access method for RankedProgress member data.

Input

IATerm*	term
	Used to report the term currently being processed.

Struct RankedQueryDoc

Struct
Header: RankedAccessor.h

Description

The document that is used as an example for query by example. This document must reside in an index used by the accessor. This struct identifies that index.

Relationships

RankedQueryDoc connects a sample IADoc to its location in a TermIndex

1 query connects a single doc to a single index.

Client

See “RankedAccessor uses sample RankedQueryDoc” on page 6-37.

Public Member Data

IADoc* doc
TermIndex* index

Public Member Functions

constructor

constructor(IADoc* doc, TermIndex* index)

Input

IADoc* doc
A pointer to the document.
TermIndex* index
A pointer to the index it resides in.

TWComponent

Struct

Struct
Header: TWVector.h

Description

A term and its weight. Used in relationship to a document.

Relationships

TWComponent points to an IATerm

1 component points to one term.

Clients

See “TWVector contains TWComponent” on page 6-48.

Public Data

TermID	termID
	The unique ID of the term.
float	weight
	The normalized weight of the term.

Class TWVector

Header: TWVector.h

Hierarchy

Base Class.

Description

A collection of weighted terms associated with a document.

Relationships

TWVector contains TWComponent

One TWVector may contain many TWComponents.

Client

See “VectorAccessor contains TWVector” on page 6-51.

Public Member Functions

constructor

Input

DocLength **length**
 The number of components in the vector.

destructor

GetComponents

Access method for TWVector member data.

Output

TWComponent* **components**
 An array of TWComponents.

GetDocumentLength

Access method for TWVector member data.

Output

DocLength **length**
 The number of components in the vector.

Normalize

Adjusts the weights of the vector components so that the Euclidean length of the vector is 1.

Note

To compare vectors using the Similarity() function, normalize them first.

SetComponents

Access method for TWVector member data.

Input

TwComponent* components
An array of TWComponents.

SetDocumentLength

Access method for TWVector member data.

Input

DocLength length
The number of components in the vector.

Similarity

Input

TwVector* other
The vector to compare to this one.

Output

float score
The score of similarity. How similar the two docs are to each other.

Notes

The score is increased by the product of the weights of any terms which appear in both vectors. (In mathematical terms, the similarity score is the inner product of the two vectors.)

Sum

Input

TWVector* other
The vector to be added to this one.

Output

TWVector*
A new vector whose length is the sum of this vector and the other vector.

VectorAccessor

Class

Header: VectorAccessor.h

Hierarchy

Public subclass of RankedAccessor. See “RankedAccessor” on page 6-36.

Description

An accessor which allows a ranked search over a vector index.

Relationships

VectorAccessor contains TWVector

One Vector Accessor contains many TWVector, one per document.

Public Member Functions

constructor(VectorIndex indices, uint32 indexCount uint32 type);**

Input

VectorIndex** indexes
The array of indexes to search.

Accessor Category

uint32 indexCount

The number of indexes in the array.

uint32 type

The kind of accessor. Defaults to VectorAccessorType, 'Vec0'.

constructor(VectorIndex index_ptr_ptr, TermIndex* context);**

Input

VectorIndex** indexes

The array of indexes to search.

TermIndex* context

An example index to locate the most similar.

destructor

GetDocTopic

See “RankedAccessor.GetDocTopic” on page 6-38.

GetTWVector

Input

IADoc* doc

The document whose components are needed.

uint32 index

The number of the index (position in the indexes array).

Output

TWVector

A pointer to the container of the components.

Notes

Returns the vector for a doc from the Nth index of this accessor

RankedSearch

See “RankedAccessor.RankedSearch” on page 6-41.

Typedefs

RankedProgressFn

The function for reporting progress during a search,

Usage

```
RankedProgressFn
    (const RankedProgress* progress, void* data)
```

Type

```
bool
```

Header

```
RankedAccessor.h
```

Constants

```
const uint32 InVecAccessorType = 'I&V0'
const uint32 InvertedAccessorType = 'Inv0'
const uint32 VectorAccessorType = 'Vec0'
```

Accessor Exceptions and Error Handling

Errors are currently handled by throwing exceptions.

VAAI

`IAccessorAlreadyInitialized`

May mean that `initialize` has been called when the accessor has already been initialized.

VANI

`IAccessorNotInitialized`

May mean that `RankedSearch` or `GetDocTopic` has been called and the accessor has not been initialized.

VAIV

`IAccessorInitInvalid`

A saved accessor initialization is no longer valid (most likely due to the fact that documents have been added or deleted).



Analysis Category

Understanding Tokens and Terms	3
Understanding Tokenizers	4
Understanding Filters	6
Existing Filters	6
Filter Sequence	8
Creating Analysis Subclasses	9
Creating a SimpleAnalysis Subclass	9
Creating a Subclass of IAAAnalysis	10
Creating a Subclass of IATokenFilter	12
Creating a Subclass of IATerm	13
Creating a Text Utility	13
Analysis Class Category Reference	15
Header Files in the Analysis Class Category	15
Class Specifications	17
AlphaTokenizer	17
DocTextCharStream	21
DowncaseFilter	23
IAAnalysis	24
IATerm	27
IAToken	31

Analysis Category

IATokenFilter	33
IATokenStream	35
ICharStream	37
ShortWordFilter	42
SimpleAnalysis	43
StringTerm	45
Constants	47
Exceptions	48

Analysis Category

The analysis classes provided by IAT provides the abstract classes for the location of terms within text.

The classes within this category can do these functions:

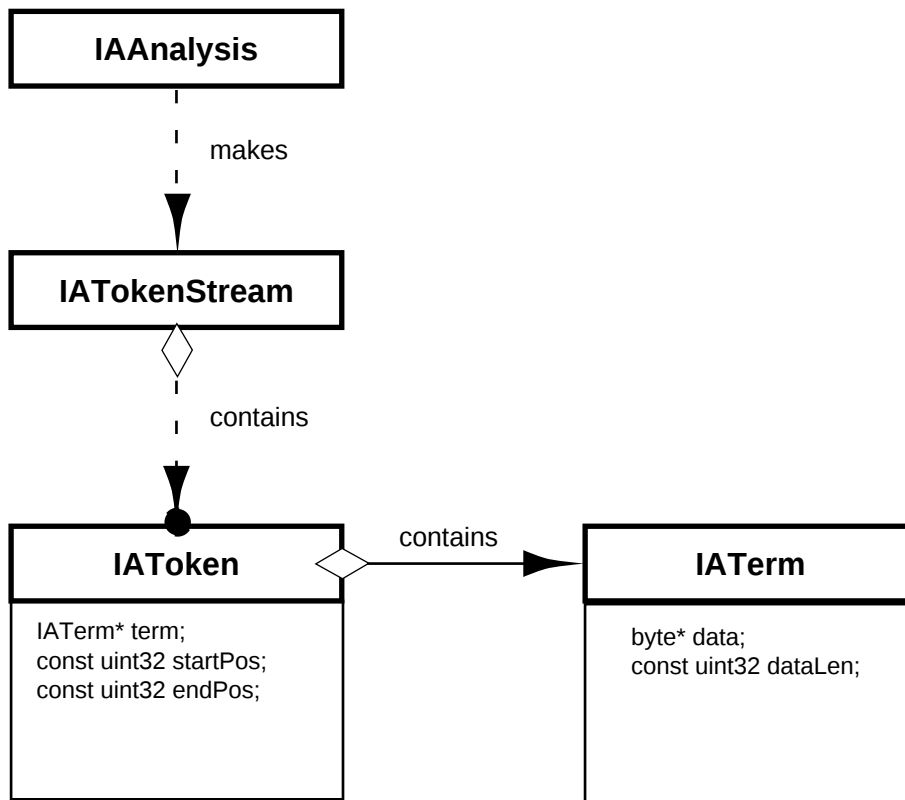
- scan text to locate tokens from which terms are extracted
- filter tokens to change them or remove them.

In this chapter we will refer to an example analysis module included with the IAT called “SimpleAnalysis.” More powerful analysis modules may also be available; these may be linked in as separate libraries.

Understanding Tokens and Terms

IAT distinguishes “token” and “term.” A “token” is a passage of text that might be a term. A “term” is a token that, after filtering, has been accepted in the index. A term is typically a word; it may be, however, the root of a word or a phrase.

The analysis provides a token stream. This stream contains many tokens, each of which generally corresponds to a single term. Figure 7-1 shows the abstract class relationships.

Figure 7-1 Class diagram of tokens and terms

Understanding Tokenizers

A tokenizer is a class that creates tokens. Tokenizers take input streams from text and, by applying logic for determining the logical beginning and end of a possible term, create tokens. Figure 7-2 illustrates one type of tokenizer, an alphabetic one.

A typical tokenizer might break the string

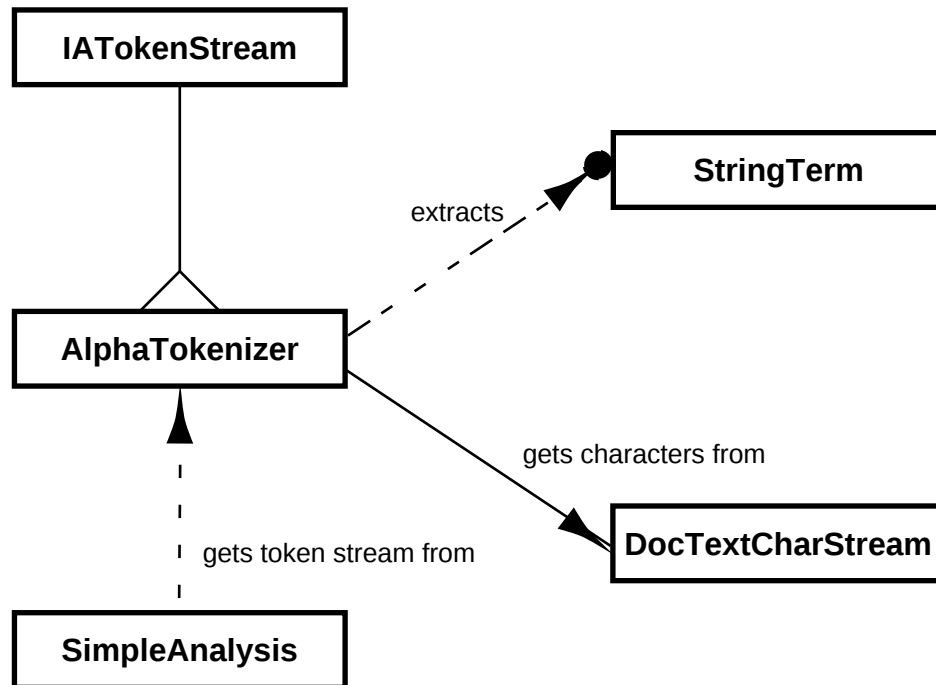
I'm going on a date with R2D2 to the Galaxy Restaurant
into the tokens:

I	date	the
M	with	Galaxy

Analysis Category

going	R	Restaurant
on	D	
a	to	

Figure 7-2 A tokenizer



AlphaTokenizer, the example tokenizer provided with **SimpleAnalysis**, creates a token stream from an input character stream. **AlphaTokenizer**, as any subclass of **IATokenStream**, provides a `GetNextToken` function. It creates tokens by selecting contiguous "chunks" of alphabetic characters under a maximum length. Non-alpha characters are skipped (however a new token is begun following non-alphabetic characters).

The **AlphaTokenizer** uses the ANSI function `isalpha()` to determine alphabetic characters.

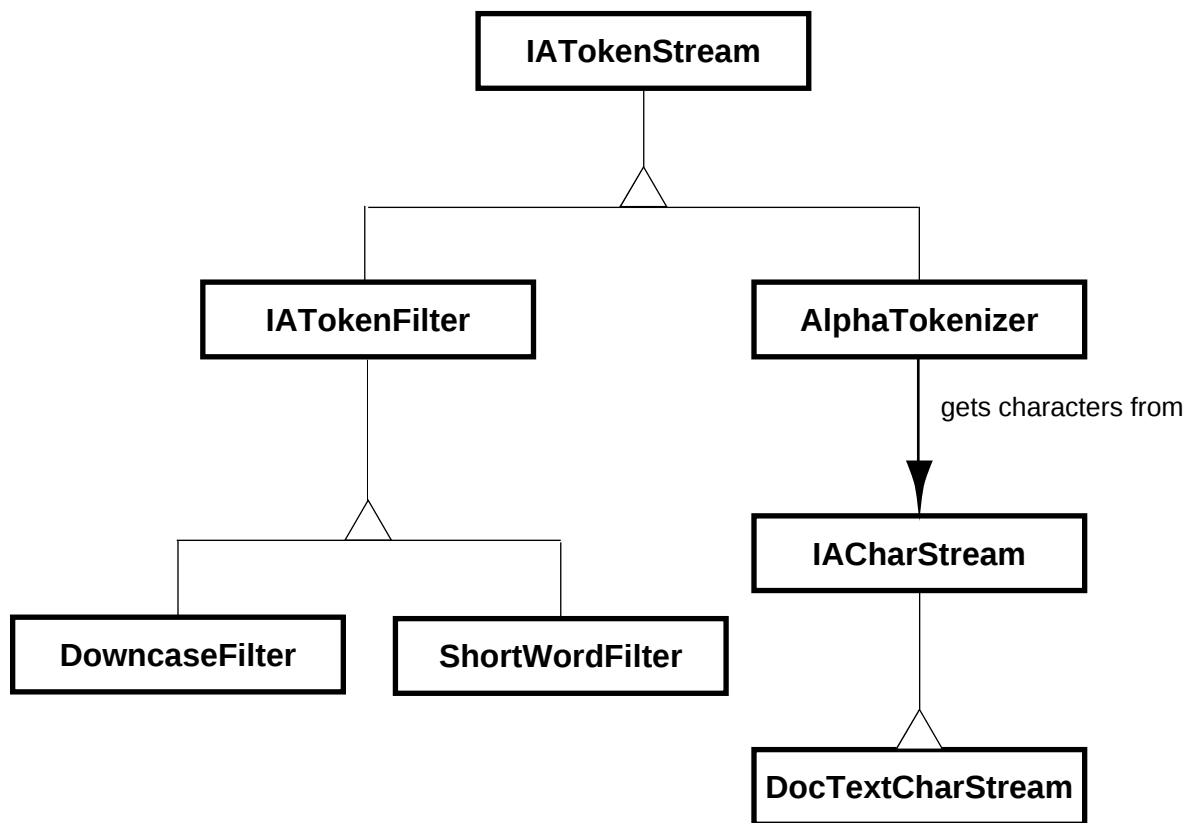
This tokenizer only works with 8-bit characters; if you are using a larger character such as UNICODE, you must provide another tokenizer.

Applications may wish to create their own tokenizer for the initial creation of tokens.

Understanding Filters

A filter is also a subclass of `IATokenStream`. Unlike tokenizers, however, they depend upon receiving a token stream as input. Filters also provide the “Get Next Token” facility; they only pass on tokens that are acceptable or that have been changed to be acceptable.

Figure 7-3 Token and filter classes



Existing Filters

Two sample filters are provided with IAT: Short Word Filter and Downcase Filter.

The Short Word Filter requires a source token stream. Its `GetNextToken` function will get the next token from that stream until it finds a token equal to or greater than its minimum length. The default minimum length (used by `SimpleAnalysis`) is three.

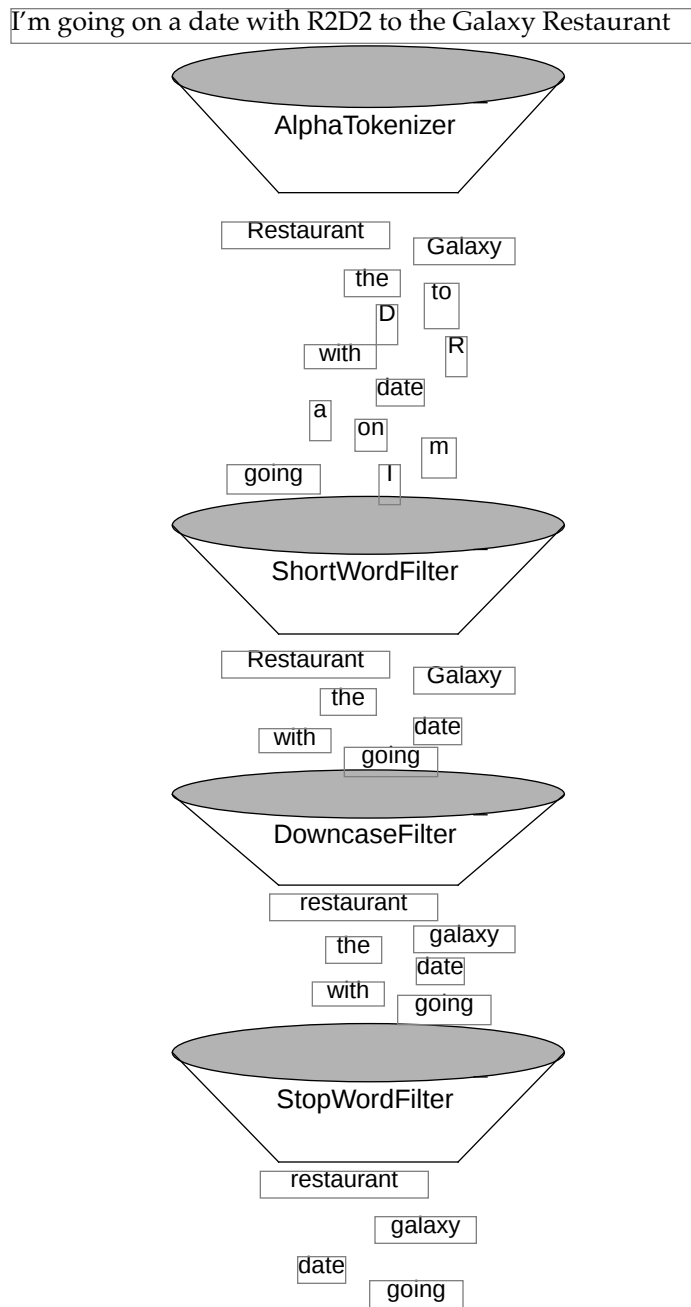
Analysis Category

The DowncaseFilter turns all uppercase characters to lowercase using the ANSI function `toLowerCase()`.

Other common types of filters, which applications will probably want to provide, include stop word filters and stemmers. Stop Word filters match the tokens against a list of tokens which are not desired for a given application. In the example in Figure 7-4, “with” and “the” are likely to be stopped by a common stop list. Stemmers remove affixes (which are generally suffixes in English) to recognize common variations of a term. For example, “going” would be reduced to “go.” This results in a normalization of terms.

Successive Filtering

A token stream is successively passed through a series of filters to achieve the desired effect. Figure 7-4 illustrates one such sequence.

Figure 7-4 Illustration of sequential filtering

Filter Sequence

The sequence of filtering is important. For example, a stemmer would have to filter

Analysis Category

following the short word filter. Otherwise the shortened forms produced by the stemmer might be filtered out as too short. A stop word list may have to be filtered to match the filtered term against it. If a stop word list includes words with upper case letters, for example, it would have to be matched to terms before they have their letters converted to lower case. If, however, the stop word list itself was filtered to be in lower case, it should not be matched until after the downcase filter.

Creating Analysis Subclasses

Creating a SimpleAnalysis Subclass

IAT provides one instantiable analysis class, SimpleAnalysis. This uses an alphabetic tokenizer, converts all terms to lower case, and filters out any token less than three characters.

You can create an stronger analysis than Simple Analysis by creating a subclass of SimpleAnalysis and adding additional filters.

Required Functions

GetProtoTerm

MakeTokenStream

Example

This example adds a stop list filter to simple analysis by creating a subclass, StopWordAnalysis.

Figure 7-5 Analysis subclass**Listing 7-1** SimpleAnalysis subclass header

```

class StopWordAnalysis : public SimpleAnalysis {
public:
    StopWordAnalysis() : SimpleAnalysis() {}
    StopWordAnalysis(StopWordAnalysis& sa) :
        SimpleAnalysis(sa) {}
    IATokenStream*   MakeTokenStream(IADocText* text);
    IATerm*          GetProtoTerm();

    IAOrderedStorableSet* stopset;
};

```

Listing 7-2 SimpleAnalysis subclass body

```

IATokenStream* StopWordAnalysis::MakeTokenStream
    (IADocText* text) {
    IATokenStream* dncase = SimpleAnalysis::MakeTokenStream(text);
    IATokenStream* stopwd = new StopWordFilter(dncase);
    return stopwd;
}

```

Creating a Subclass of IAAnalysis

If you wish to use a different tokenizer or omit one of the filters in SimpleAnalysis, you may wish to create a subclass of IAAnalysis. See “IAAnalysis” on page 7-24 for detailed information on its contents.

Analysis Category

The code in Listing 7-3 and Listing 7-4 demonstrates how to subclass IAnalysis — in this case, to build SimpleAnalysis.

Required Functions

GetProtoTerm

MakeTokenStream

Example

Listing 7-3 IAnalysis subclass header

```
#include "IAnalysis.h"
const uint32SimpleAnalysisType = 'Sim1';

class SimpleAnalysis : public IAnalysis {

public:
    SimpleAnalysis() : IAnalysis(SimpleAnalysisType) {}

    SimpleAnalysis(SimpleAnalysis& sa) : IAnalysis(sa) {}

    ITokenStream*    MakeTokenStream(IADocText* text);

    ITerm*           GetProtoTerm();
```

Listing 7-4 IAnalysis subclass body

```
#include "SimpleAnalysis.h"
#include "DocTextCharStream.h"
#include "AlphaTokenizer.h"
#include "DowncaseFilter.h"
#include "ShortWordFilter.h"
#include "StringTerm.h"

ITokenStream* SimpleAnalysis::MakeTokenStream(IADocText* text) {
    return new DowncaseFilter
        (new ShortWordFilter
            (new AlphaTokenizer
                (new DocTextCharStream(text))));
```

```

}

IATerm* SimpleAnalysis::GetProtoTerm() {
    return new StringTerm("");
}

```

Creating a Subclass of IATokenFilter

The core of a new filter is the implementation of `GetNextToken`, which takes the next token offered by the source `IATokenStream`, and weeds it out or alters it before passing it on.

See “`IATokenFilter`” on page 7-33 for detailed information on the abstract base class.

The `StopWordFilter` created in this example establishes the ordered storable set of stop words when it is constructed and places its input `IATokenStream` into `source`.

Required Functions

`GetNextToken`

Listing 7-5 StopWordFilter header

```

#include "SimpleAnalysis.h"
#include "HFSStorage.h"
#include "IAStorable.h"

class StopWordFilter : public IATokenFilter {
public:
    StopWordFilter(IATokenStream* s);
    virtual IAToken* GetNextToken();
protected:
    IAOrderedStorableSet* stopset;
}

```

Listing 7-6 StopWordFilter implementation of GetNextToken

```

IAToken* StopWordFilter::GetNextToken() {
for (IAToken* token = source->GetNextToken(); token;
    token = source->GetNextToken()) {
    IATerm* stopTerm = (IATerm*)stopset->Get(token->term);
    if (!stopTerm) {
        return token;
    } else {
        delete token;
    }
}
return Nil;
}

```

Creating a Subclass of IATerm

You may create a subclass of IATerm if you would like to create a custom constructor, or if you need to provide additional type conversions such as `char*` to `byte*`. You may not, however, change the implementation of its order, `LessThan` or `Equal`, or of its format, `Store` or `Restore`, as the current index logic is dependent upon the existing order and format.

See “IATerm” on page 7-27 for detailed information on the abstract class.

Required Functions

None

Creating a Text Utility

`DocTextCharStream` is an IAT-provided utility class that reads buffers of text from the `HFSDoc`.

You may need to provide another implementation for your documents.

Required Functions

`GetNextBuffer`

Example

Listing 7-7 DocTextCharStream header

```
#include "ICharStream.h"
#include "IACorpus.h"

class DocTextCharStream : public ICharStream {
public:
    DocTextCharStream() : ICharStream(), docText(NULL) {}
    DocTextCharStream(IADocText* text) :
        ICharStream(), docText(text) {}
    ~DocTextCharStream();

    uint32GetNextBuffer(char* buffer, uint32 bufferLen);
private:
    IADocText*docText;

};
```

Listing 7-8 DocTextCharStream body

```
#include "DocTextCharStream.h"

DocTextCharStream::~DocTextCharStream() {
    delete docText;
}

uint32 DocTextCharStream::GetNextBuffer(char* buffer, uint32 bufferLen) {
    return docText->GetNextBuffer((byte*)buffer, bufferLen);
}
```

Analysis Class Category Reference

Header Files in the Analysis Class Category

AlphaTokenizer.h

AlphaTokenizer

DocTextCharStream.h

DocTextCharStream

DowncaseFilter.h

DowncaseFilter

IAAnalysis.h

IAAnalysis
IATerm
IAToken
IATokenStream
IATokenFilter

IACCharStream.h

IACCharStream

ShortWordFilter.h

ShortWordFilter

SimpleAnalysis.h

SimpleAnalysis

StringTerm.h

StringTerm

Class Specifications

AlphaTokenizer

Class

Header: AlphaTokenizer.h

Hierarchy

Public subclass of IATokenStream. See “IATokenStream” on page 7-35.

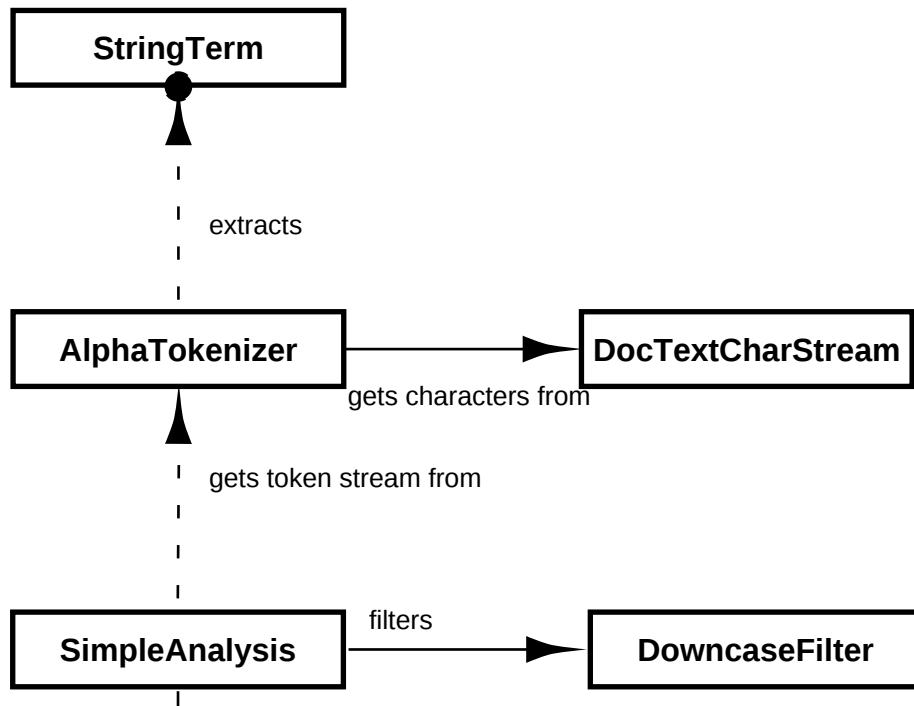
Description

AlphaTokenizer breaks a stream of characters into tokens. These tokens are contiguous alphabetic characters (as determined by the ANSI function `isalpha`). Non-alphabetic characters cause the end of a token and are removed from the stream.

Tokens longer than 63 characters are broken into smaller tokens. This number may be changed by altering the constant `AlphaTokenizerMaxTokenLen`.

Relationships

Figure 7-6 AlphaTokenizer relationships



AlphaTokenizer gets characters from DocTextCharStream

An AlphaTokenizer gets its input from a single DocTextCharStream, given to it at the time of construction.

AlphaTokenizer extracts IAToken

An AlphaTokenizer finds many IATokens in the text stream.

Clients

See “Simple Analysis gets tokens from AlphaTokenizer” on page 7-44.

Public Member Functions

constructor(IACharStream* stream)

Input

IACharStream* stream

The character stream that will provide characters to the tokenizer.

Usage

```

IATokenStream* SubclassAnalysis::
    MakeTokenStream(IADocText* text) {
    return new DowncaseFilter(new ShortWordFilter(new
        AlphaTokenizer(new DocTextCharStream(text))));
}

```

destructor

Notes

Deletes charStream.

GetNextToken

Output

IAToken*

The next alphabetic token found in the stream of characters. Returns NULL at end of stream.

Usage

```

for (IAToken* token = ts->GetNextToken(); token;
    token = ts->GetNextToken()) {
    posting.term = token->term;
}

```

GetTextSpan

See “IATokenStream.GetTextSpan” on page 7-37.

Protected Member Functions

GetCharStream

Access method for AlphaTokenizer member data.

Output

IACharStream*charStream

A pointer to the input character stream. This stream is deleted upon destruction.

GetStreamBuffer

Access method for AlphaTokenizer member data.

Output

char* buffer

Holds the token in progress.

SetCharStream

Access method for AlphaTokenizer member data.

Input

IACharStream*charStream

A pointer to the input character stream. This stream is deleted upon destruction.

SetStreamBuffer

Access method for AlphaTokenizer member data.

Input

char *	buffer
	Holds the token in progress.

DocTextCharStream

Class

Header: DocTextCharStream.h

Hierarchy

Subclass of ICharStream.

Description

The DocTextCharStream is a utility that selects text from IADocs. SimpleAnalysis provides DocTextCharStream as an input to the creation of the AlphaTokenizer.

Relationships

DocTextCharStream gets chars from IADocText

One DocTextCharStream gets its characters from IADocText.

AlphaTokenizer gets char stream from DocTextCharStream

In the implementation of Simple Analysis, AlphaTokenizer is created with a DocTextCharStream. See AlphaTokenizer for more information.

Public Member Functions

constructor

constructor(IADocText* text)

Input

IADocText* text
The text of the document to be analyzed.

Usage

```
IATokenStream* SubclassAnalysis::
    MakeTokenStream(IADocText* text) {
    return new DowncaseFilter(new ShortWordFilter(new
        AlphaTokenizer(new DocTextCharStream(text))));
}
```

destructor

Deletes the input text.

GetNextBuffer

Input

char* buffer
The pointer to the buffer.

uint32 bufferLen
The size of the buffer to read.

Output

uint32 charsRead
The number of characters read. 0 if no more buffers.

Analysis Category

Notes

Returns a character pointer to the buffered data.

Usage

```
uint32 charsRead = GetNextBuffer(buffer, IADiskBlockSize);
```

DowncaseFilter

Class

Header: DowncaseFilter.h

Hierarchy

Public subclass of IATokenFilter. See “IATokenFilter” on page 7-33.

Description

Downcase filter is an available filter for analysis. It changes any tokens in the token stream to be all lower case. DowncaseFilter uses the ANSI function `tolower`.

All terms provided to the Downcase Filter must be StringTerms.

Clients

See “Simple Analysis filters tokens through DowncaseFilter” on page 7-44.

Public Member Functions

constructor(IATokenStream* stream)

Input

IATokenStream* stream

The stream of tokens to be filtered.

Analysis Category

Usage

```

IATokenStream* SubclassAnalysis::
    MakeTokenStream(IADocText* text) {
    return new DowncaseFilter(new ShortWordFilter(new
        AlphaTokenizer(new DocTextCharStream(text))));
}

```

GetNextToken

See “IATokenStream.GetNextToken” on page 7-43.

Class IAnalysis

Header: IAnalysis.h

Hierarchy

Abstract Base Class.

Description

IAnalysis is the base class for the provision of terms from given text. It is used by the index class to locate terms in text provided by the corpus.

Relationships

IAnalysis makes IATokenStream

An analysis makes one token stream.

Public Member Functions

constructor(uint32 type)

Input

uint32 **type**
 A constant that indicates which type of analysis was built. This allows easier reconstruction of existing indexes.

Usage

(SimpleAnalysis is a subclass of IAAalysis)

```
InvertedIndex index(storage,
                     new HFSTextFolderCorpus(folderName),
                     new SimpleAnalysis(SimpleAnalysisType));
```

Initialize

Virtual.

Input

IStorage* **storage**
 Open or initialized storage.

IABlockID **analysisRoot**
 A root block allocated to store analysis items.

Notes

Initializes persistent state, writing analysis parameters to storage.

Usage:

```
analysisRoot = storage->Allocate();
analysis->Initialize(storage, analysisRoot)
```

GetProtoTerm

Pure virtual.

Output

IATerm*

The type of term produced by this analysis

Notes

Returns a prototype term, for bootstrapping sets of terms.

Usage

```
termInfoSet = IAMakeOrderedStorableSet
              (MakeTermInfo(analysis->GetProtoTerm(), 0));
```

MakeTokenStream

Pure virtual.

Input

IADocText* docText

Document text as received from the corpus.

Output

IATokenStream*

A stream handler for the tokens found in the text.

Notes

Builds and returns a tokenizer. The resulting token stream may be filtered through other IATokenFilters.

Usage

```
IATokenStream* ts = index->analysis->MakeTokenStream
                    (index->corpus->GetDocText(doc));
```

Listing 7-9 Sample implementation of filtered MakeTokenStream

```

IATokenStream* SimpleAnalysis::MakeTokenStream(IADocText* text) {
    return new DowncaseFilter
        (new ShortWordFilter
            (new AlphaTokenizer
                (new DocTextCharStream(text))));
}

```

Open

Virtual.

Input

IAStorage* storage
Allocated and opened storage.

IABlockID analysisRoot
Allocated block for the analysis.

Notes

Reads persistent state, checking that it's consistent with current parameters.

Usage

```

analysisRoot = input->ReadUInt32();// reading from index root
analysis->Open(storage, analysisRoot);

```

IATerm**Class**

Header: IAnalysis.h

Hierarchy

Public subclass of IAOrderedStorable. See “IAOrderedStorable” on page 10-14.

Description

An IATerm is the unit of indexing used in IAT.

Analysis Category

Clients

See “IAToken contains IATerm” on page 7-31.

See “TermInfo contains IATerm” on page 5-60.

Public Member Functions

constructor (const byte* buffer, uint32 length);

Input

const byte* buffer
Pointer to the term.

uint32 length
The length of the term.

Usage

```
new IAToken(new StringTerm(buffer, i), start,  
            charStream->CurrentPos() - 1);
```

destructor

Frees data.

DeepCopy

See “IAStorable.DeepCopy” on page 10-28.

Equal

See “IAOrderedStorable.Equal” on page 10-15. Subclasses should not override this function.

EqualNonVirtual

const

Input

const IAOrderedStorable* neighbor

Output

bool
True if equal, false if not.

Notes

Non-virtual implementation of Equal() for use by performance-critical code. Implemented version tests equality by word rather than byte.

GetData

Access method for IATerm member data.

Output

byte* data
The contents of the term. Allocated with IAMallocArraySized. This is stored in a uint32-aligned array created by AllocData.

GetDataLength

Access method for IATerm member data.

Output

uint32 dataLen
The length of the term.

LessThan

See “IAOrderedStorable.LessThan” on page 10-16. Subclasses should not override this function.

LessThanNonVirtual

const

Input

const IAOrderedStorable* neighbor

Output

bool

True if less than, false if not.

Notes

Non-virtual implementation of LessThan() for use by performance-critical code.
Implemented version tests by word rather than by byte.

Restore

See "IAStorable.Restore" on page 10-28.

Store

See "IAStorable.Store" on page 10-30.

StoreSize

See "IAStorable.StoreSize" on page 10-29.

Private Member Functions

AllocData

const

Analysis Category

Input

uint32 dataLenTerm

The number of bytes in the term.

Output

byte* uint32Data

A pointer to a uint32 array allocated to hold the term.

Notes

Allocates an array of uint32 corresponding to the length of the buffer. Does not load the array.

IAToken

Class

Header: IAAanalysis.h

Hierarchy

Base Class.

Description

An IAToken is a relationship between a term and a character stream. It represents a series of characters which may be a term.

In the given implementation, IATokens are created with the AlphaTokenizer.

Relationships

IAToken contains IATerm

An IAToken contains one and only one term. A term may be in more than one token, or no tokens once constructed.

IAToken is a portion of DocTextCharStream

An IAToken points to a start and end position within one character stream. A character stream may have many IATokens.

Clients

See “IATokenStream contains IAToken” on page 7-36.

Public Member Functions

constructor (IATerm* term, uint32 start, uint32 end)

Input

IATerm* term
Term
uint32 start
Start position in the character stream.
uint32 end
End position in the character stream.

Usage

```
new IAToken(new StringTerm(buffer, i), start,
            charStream->CurrentPos() - 1);
```

destructor

Virtual
Deletes the term.

GetEndPosition

Access method for IAToken member data.

Analysis Category

Output

uint32	endPos One greater than the position of the last character corresponding to this token.
--------	--

GetStartPosition

Access method for IAToken member data.

Output

uint32	startPos The byte position of the first character in the text corresponding to this token.
--------	---

GetTerm

Access method for IAToken member data.

Output

IATerm*	term The term within the token.
---------	------------------------------------

IATokenFilter

Class

Header: IAAAnalysis.h

Hierarchy

Subclass of IATokenStream. See “IATokenStream” on page 7-35.

Description

An IATokenFilter is a specialized IATokenStream which depends upon an input stream to modify. The filter will examine this stream and return only those tokens which pass its filter, or, in some cases, return a modified token.

Relationships

IATokenFilter filters IATokenStream

One token filter filters one and only one token stream at a time. One token stream may be filtered by several filters, but is usually sent through each sequentially.

Public Member Functions

constructor(IATokenStream* sourceStream)

Input

IATokenStream* sourceStream
A token stream from which to extract unfiltered tokens.

Usage

(DowncaseFilter is a subclass of IATokenFilter)

```
IATokenStream* SubclassAnalysis::
    MakeTokenStream(IADocText* text) {
    return new DowncaseFilter(new ShortWordFilter(new
        AlphaTokenizer(new DocTextCharStream(text))));
}
```

destructor

Deletes source.

GetNextToken

See “IATokenStream.°GetNextToken” on page 7-36.

Filters may bypass tokens until one is allowed to filter through.

Listing 7-10 Sample Implementation of `GetNextToken` for an `IATokenFilter`

```

IAToken* DowncaseFilter::GetNextToken() {
    IAToken* token = source->GetNextToken();
    if (!token) return NULL;
    StringTerm* term = (StringTerm*)token->term;
    for (uint32 i = 0; i < term->TextLen(); i++)
        term->Text()[i] = tolower(term->Text()[i]);
    return token;
}

```

GetTextSpan

See “`GetTextSpan`” on page 7-37. `GetTextSpan()` on a filter delegates to its source by default.

Protected Member Data

```

IATokenStream* source
    The source of tokens to be filtered.

```

IATokenStream**Class**

Header: `IAAnalysis.h`

Hierarchy

Abstract Base Class.

Description

`IATokenStream` is typically used as the interface between a character stream and the index. It provides tokens from the text provided by the corpus.

There are generally two types of token streams, tokenizers or filters. Tokenizers are the original providers of tokens constructed from the text. Filters are successive token streams that modify or filter out the contained tokens.

Relationships

IATokenStream contains IAToken

One token stream may contain many tokens. One token resides in one token stream.

Clients

See “IAAnalysis makes IATokenStream” on page 7-24.

Public Member Functions

constructor

Only used for initialization. Operational TokenStreams are constructed through IAAnalysis.MakeTokenStream(). See page page 7-26.

GetNextToken

Pure virtual.

Output

IAToken* token

Next token in the stream, or Nil if at end of the stream.

Usage

```
for (IAToken* token = ts->GetNextToken(); token;
    token = ts->GetNextToken()) {
    posting.term = token->term;
}
```

GetTextSpan

Input

`byte* buffer`
Destination address for the span.

`uint32 startPos`
Start position in the character stream.

`uint32 endPos`
End position in the character stream.

Notes

Copies into the destination a span of bytes from the source text. The span must start less than a buffer before the end of the last token read, and it may not extend past the end of the last token read. If it starts more than a buffer before, `AnalysisSpanUnavailable` is signalled.

Used to create a `byte*` copy of the term contents.

Usage

```
for (IAToken* token = ts->GetNextToken();
     token; token = ts->GetNextToken()) {
    ts->GetTextSpan((byte*)buffer, token->startPos, token->endPos);
}
```

ICharStream

Class

Header: `ICharStream.h`

Hierarchy

Base class.

Description

An `ICharStream` supplies a stream of characters to a tokenizer.

To access a stream of characters from the text of a document, use the subclass `DocTextCharStream`.

Public Member Functions

constructor

destructor

Deletes the text buffers.

AdvanceTo

Virtual.

Input

`uint32 desiredPosition`
The desired position in the character stream.

Notes

Places the position in the character stream at the desired position. Will fail with a VTWN exception if the desired position is before the current position or after the end of the set.
Subclasses may wish to implement a specialized faster version of this function.

Usage

```
// read ahead 5
uint32 desiredPosition = currentPosition + 5;
cs->AdvanceTo(desiredPosition)
```

CurrentPos

Inline.

Output

`uint32 currentPos`
The current position in the character stream.

Analysis Category

Usage

```
// note start of token
uint32 start = charStream->CurrentPos() - 1;
```

GetBuffer

Access method for ICharStream member data.

Output

char* buffer
A pointer to the current buffer of characters.

GetBufferPos

Access method for ICharStream member data.

Output

uint32 bufferPos
Position of the first character in the buffer.

GetEndChar

Access method for ICharStream member data.

Output

char* endChar
A pointer to the end of the current buffer.

GetNextChar

Input

bool* eos
False upon input. Returns True if the read is past the end of the set.

Analysis Category

Output

char

The next character past the current position in the buffer; NULL if past end of buffer.

Notes

Eos is assumed to be false, and is only set when eos is reached (read past end of buffer). When eos is set, the return value should be ignored.

```
char c;
// skip non-alpha characters
do {
    c = charStream->GetNextChar(&eof);
    if (eof) return NULL;
} while (!isalpha(c));
```

GetNextCharInBuffer

Access method for ICharStream member data.

Output

char* nextChar
A pointer to the next character to read in buffer.

GetTextSpan

Input

char* buffer

uint32 startPos

uint32 endPos

Notes

This can be used by a client to report the range of bytes in which a matching term occurred (“key word in context”).

SetBuffer

Access method for ICharStream member data.

Input

char*	buffer
	A pointer to the current buffer of characters.

SetBufferPos

Access method for ICharStream member data.

Input

uint32	bufferPos
	Position of the first character in the buffer.

SetEndChar

Access method for ICharStream member data.

Input

char*	endChar
	A pointer to the end of the current buffer.

SetNextCharInBuffer

Access method for ICharStream member data.

Input

char*	nextChar
	A pointer to the next character to read in buffer.

Protected Member Functions

GetNextBuffer

Input

`char* buffer`
The pointer to the buffer.

`uint32 bufferLen`
The size of the buffer to read.

Output

`uint32 charsRead`
The number of characters read. 0 if no more buffers.

Notes

Returns a character pointer to the buffered data. Subclasses must implement only this one method.

Usage

```
uint32 charsRead = GetNextBuffer(buffer, IADiskBlockSize);
```

Class ShortWordFilter

Header: ShortWordFilter.h

Hierarchy

Public subclass of IATokenFilter. See “IATokenFilter” on page 7-33.

Description

An IATokenFilter that will not pass tokens over a minimum length. The default length is three characters.

Public Member Functions

constructor(IATokenStream* sourceStream, uint32 l = MinWordLength)

Input

IATokenStream* sourceStream
The input token stream.

uint32 l = MinWordLength
The smallest length of token to allow through.

Notes

MinWordLength is a constant defined in the header. Current value is 3.

Usage

```
IATokenStream* SubclassAnalysis::
    MakeTokenStream(IADocText* text) {
    return new DowncaseFilter(new ShortWordFilter(new
        AlphaTokenizer(new DocTextCharStream(text))));
}
```

GetNextToken

See “IATokenStream.GetNextToken” on page 7-36.

SimpleAnalysis

Header: SimpleAnalysis.h

Class

Hierarchy

Public subclass of IAAAnalysis. See “IAAnalysis” on page 7-24.

Description

A version of IAAAnalysis that provides lower-case alphabetic tokens over two characters long.

Relationships

Simple Analysis gets tokens from AlphaTokenizer

Simple Analysis gets tokens from one AlphaTokenizer.

Simple Analysis filters tokens through DowncaseFilter

Simple Analysis filters tokens through one downcase filter.

Simple Analysis filters tokens through ShortWordFilter

Simple Analysis filters tokens through one short word filter.

Constants

```
const uint32 SimpleAnalysisType = 'Sim1';
```

Public Member Functions

constructor

Usage

```
InvertedIndex index(storage,
                    new HFSTextFolderCorpus(folderName),
                    new SimpleAnalysis());
```

Notes

The type is constant and established with the default construction.

copy constructor(SimpleAnalysis& sa)

GetProtoTerm

See “IAAnalysis.GetProtoTerm” on page 7-26.

MakeTokenStream

See “IAAnalysis.MakeTokenStream” on page 7-26. SimpleAnalysis uses AlphaTokenizer, DowncaseFilter and ShortWordFilter. The result is terms of 3 or greater alphabetic characters in lower case.

StringTerm

Class

Header: StringTerm.h

Hierarchy

Public subclass of IATerm.

Description

String Term is the term produced by the AlphaTokenizer. It uses characters rather than bytes.

Public Member Functions

constructor(const char* text)

Input

char*	text
	The IATerm text converted to characters.

constructor(const char* text, uint32 length)

Input

char*	text
	The IATerm text converted to characters.
uint32	length
	the number of characters in the string

DeepCopy

See “IATerm.DeepCopy” on page 7-28.

Text

Output

char*	text
	The IATerm text converted to characters.

TextLen

Output

uint32	textLen
	The number of characters.

Constants

AlphaTokenizerMaxTokenLen=63

The maximum length of a token

uint32 MinWordLength = 3

The length of a token the Short Word Filter will allow through. Tokens with fewer characters than this are filtered out of the token stream.

uint32 SimpleAnalysisType = 'Sim1';

The identifier of the simple analysis type.

Exceptions

VASU

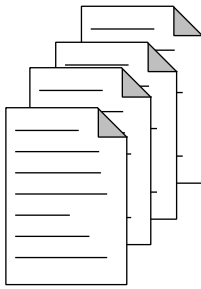
`AnalysisSpanUnavailable.`

Thrown by `IAAnalysis`.

VTSU

`TextSpanUnavailable.`

Thrown by `ICharStream`.



Corpus Category

Introduction	3
The HFS Implementation	4
HFS Corpus	4
HFSTextFolderCorpus	5
Common Procedures	5
Using a Corpus to Provide Documents	5
Creating a New Corpus	6
Establishing an Existing Corpus	7
Using an HFSCorpus to Locate a Document in HFS	7
Creating Corpus Subclasses	7
Creating a Subclass of IACorpus	8
Creating a Subclass of IADoc	9
Creating a Subclass of IADocIterator	10
Creating a Subclass of IADocText	11
Creating a Subclass of HFSIterator	13
Corpus Class Category Reference	16
Header Files in the Corpus Category	16
Class Specifications	17
DirectoryInfo	17
HFSCorpus	17
HFSDoc	21
HFSDocText	25
HFSVolumeInfo	29

Corpus Category

HFSIterator	32
HFSTextFolderCorpus	35
HFSTextFolderDoc	39
IACorpus	41
IADoc	47
IADocIterator	49
IADocText	50
Constants	51
Exceptions	51

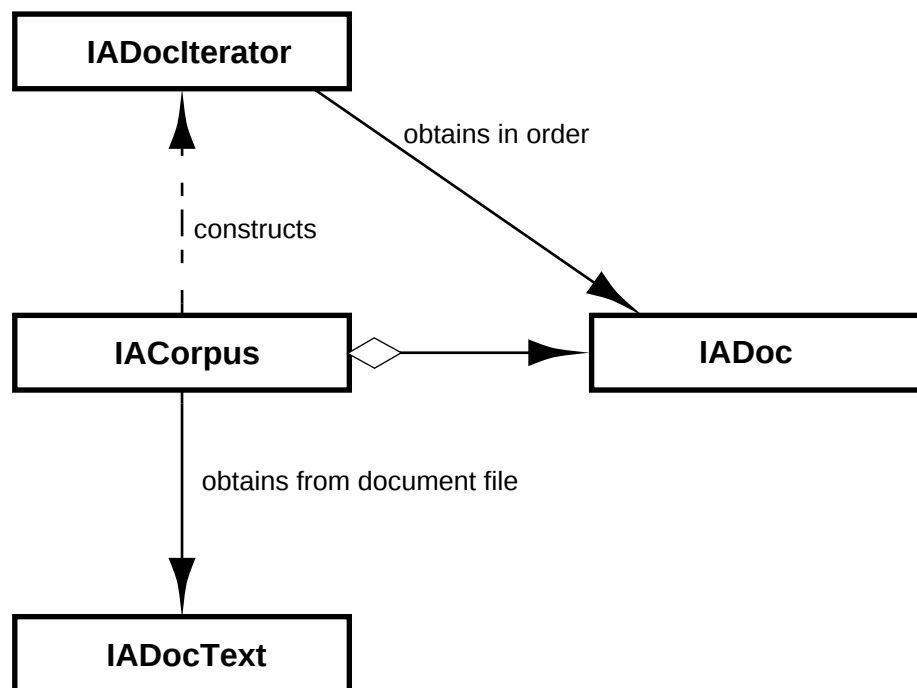
Introduction

In the field of information retrieval a “corpus” is a collection of documents being searched. In IAT the corpus class provides the tools for identifying a set of documents as a collection and providing text from these documents so they can be indexed.

The corpus is the interface between documents and the index. The corpus locates the document files and provides buffer text from these documents to the index and analysis objects. The corpus maintains the location of the collection of documents and, optionally, provides an iterator through them.

Figure 8-1 shows the relationships between the abstract classes.

Figure 8-1 Corpus relationships



Each index has a single corpus; that is, the documents within an index must be of the same type.

The actual use of a corpus is closely coupled to an index; the index classes are the major clients of the corpus classes. There is no given way to store a corpus except through an index. The index `Update` function uses the corpus iterator to review all documents

Corpus Category

within the corpus and update the index as required. The index locates terms by feeding the analysis text read from the documents through the corpus.

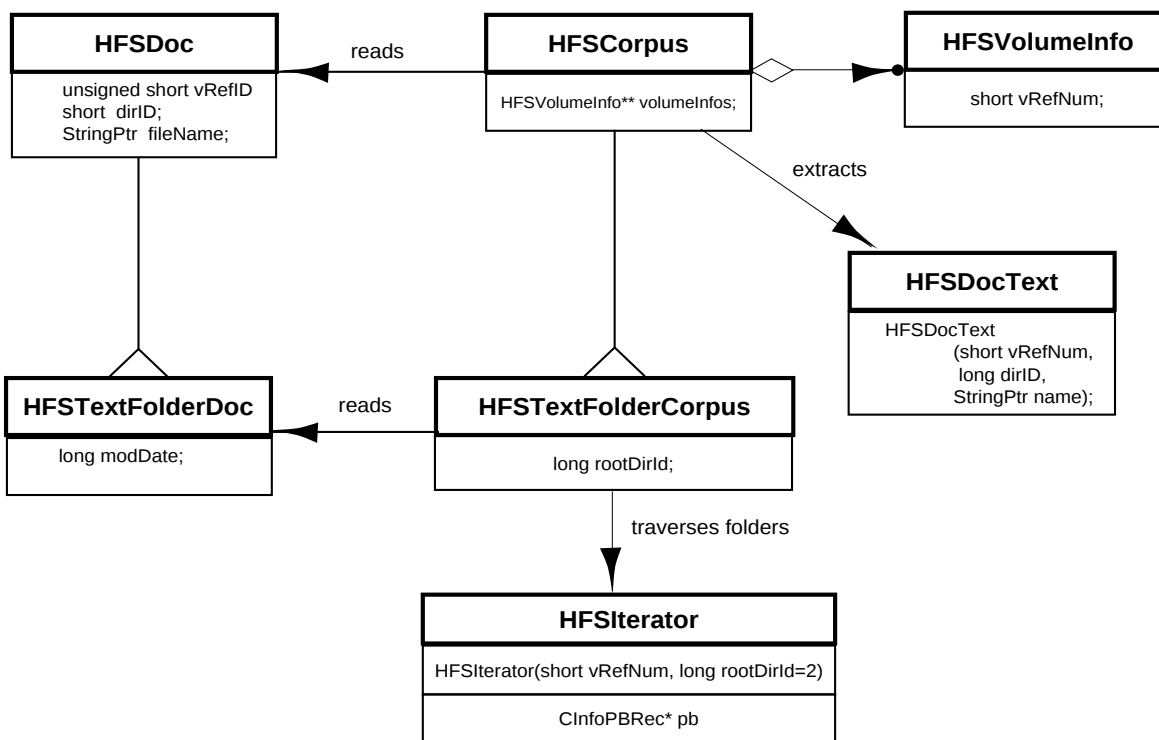
IAT provides an implementation that supports HFS files and interfaces to the collection of text files within an HFS Folder. If you require the ability to index documents of another file type, you must construct a corpus subclass for that type.

The HFS Implementation

There are two implementations of the corpus abstract classes. HFSCorpus provides access to the text in HFS files. HFSTextFolderCorpus provides, in addition, the ability to iterate through a folder and its subdirectories and select text documents.

Figure 8-2 shows these implementations.

Figure 8-2 HFS instantiation of corpus classes



HFS Corpus

The HFS Corpus characterizes the set of documents. It contains a mapping to which volumes the documents reside on. The HFS-provided vRefNum cannot be used as a persistent identifier of a document as it may change when the system is rebooted (it

Corpus Category

depends on the order in which devices are mounted). IAT has assigned its own persistent vRefID to each volume, and maintains a mapping of the vRefID to the vRefNum within HFSCorpus.

HFSDoc contains the information to locate a document: its vRefID, dirID, and fileName.

The HFS Corpus can be used to extract text from given HFS text documents. It has no iterator; that is, it may not be used, without further subclassing, with the Update() function of an index. Updates must be individually done.

HFSTextFolderCorpus

The HFSTextFolderCorpus is a subclass of the HFSCorpus. It maintains an iterator that chooses, from a given folder, any document with file type 'TEXT' within that folder or folders it contains.

HFSTextFolderDoc contains a modification date. Only those text documents modified since the last update are submitted for re-analysis.

The HFSIterator is a utility used within the private implementation of the HFSCorpus iterator. This utility will navigate through all the folders within a given root directory ID and return the next available document of any type.

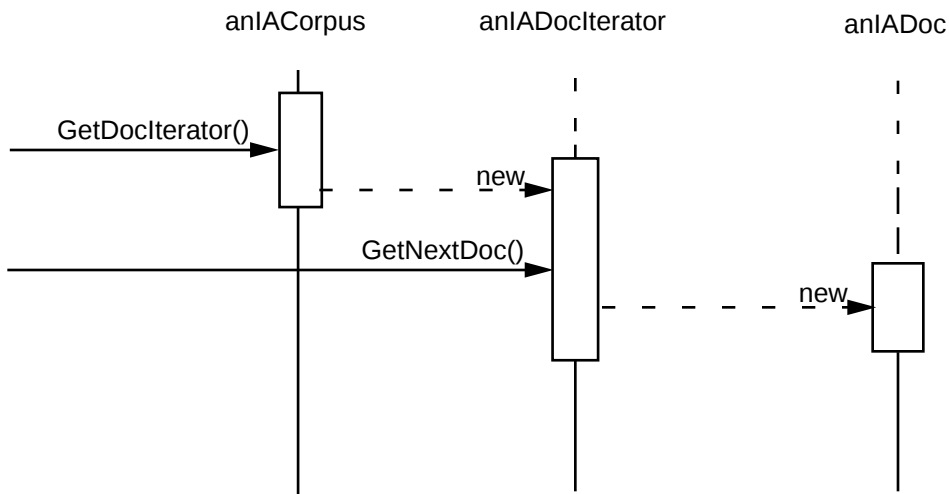
The HFSTextFolderCorpus will iterate through all folders and contained folders and select text documents from them.

Common Procedures

Using a Corpus to Provide Documents

Using the corpus document iterator can provide all documents currently in the corpus, whether or not indexed.

The example illustrates listing all the documents in an HFS Text Folder.

Figure 8-3 Interaction diagram for iterating through a corpus**Listing 8-1** List text files

```

// build the corpus
HFSTextFolderCorpus anHFSTextFolderCorpus(folderName);
printf ("%22.30s\n", folderName);

// get an iterator through the corpus
IADocIterator* anIADocIterator = anHFSTextFolderCorpus.GetDocIterator();
HFSTextFolderDoc* anHFSTextFolderDoc;
while (anHFSTextFolderDoc =
        (HFSTextFolderDoc*)anIADocIterator->GetNextDoc()) {
    // NULL when no more text docs in folder
    printf("\t");
    PrintDocName(anHFSTextFolderDoc);
    printf("\n");
}

```

Creating a New Corpus

A corpus is stored through its index. Generally a corpus is created at the same time an index is created. See “Creating an Index” beginning on page 5-8.

Establishing an Existing Corpus

The corpus is stored through its index. To establish an existing corpus, you must first establish its index (See “Establishing an Existing Index” on page 5-11) and then address the corpus data member. The corpus is stored in the index as an IACorpus.

Listing 8-2 Establishing an existing corpus

```
// establish the existing index containing the corpus
// (see “Establish an existing index” on page 5-13 for example)
anInVecIndex.Open();
```

Using an HFSCorpus to Locate a Document in HFS

The file information for an HFSDoc can be found by using public access methods.

```
Volume Reference Number
    anHFSCorpus->GetVRefNum(anHFSDoc->GetVolumeRefID())
DirectoryID
    anHFSDoc->GetDirID()
Filename
    anHFSDoc->GetFileName()
```

HFSTextFolderCorpus provides this information:

```
Volume Reference Number of root text folder
    anHFSTextFolderCorpus->GetVolumeRefNum()
Directory ID of the root folder
    anHFSTextFolderCorpus->GetRootDirID()
```

See “The HFS Implementation” on page 8-4 for more information on vRefID and vRefNum.

Creating Corpus Subclasses

If you need to create a corpus subclass, you generally need to create several subclasses:

One of IACorpus, to characterize the set of documents

One of IADoc, to provide information to uniquely identify and locate a single document

One of IADocText, to obtain a text string from the document.

You may also need to provide a subclass of IADocIterator if you wish to provide an index Update() function.

Corpus Category

The following examples use the HFS corpus implementation as an example. Specifically, we show how the HFSCorpus class is derived from IACorpus, and how its associated classes (such as HFSDoc) are derived from their base classes (in this case, IADoc).

Creating a Subclass of IACorpus

You may wish to create a subclass of IACorpus to access documents for an implementation other than the provided HFS Corpus.

See “IACorpus” beginning on page 8-41 for detailed information on this class.

Required Functions

- GetProtoDoc (establishes which type of document is accessed through this corpus)
- GetDocText (provides the text from the document).

Listing 8-3 Sample header file of an IACorpus subclass

```
class HFSCorpus : public IACorpus {
public:
    HFSCorpus(uint32 type = HFSCorpusType)
        : volumeInfos(NULL), volumeCount(0), IACorpus(type) {}
    ~HFSCorpus();

    // IACorpus methods
    IADoc*      GetProtoDoc();
    IADocText*  GetDocText(const IADoc* doc);

    // HFSCorpus-specific methods
    unsigned short  GetVRefID(short vRefNum);
    short          GetVRefNum(unsigned short vRefID);

protected:
    IABlockSize    InitialSize();
    void           Initializing(IAOutputBlock* output);
    void           Opening(IAInputBlock* input);
    IABlockSize    UpdateSize();
    void           Updating(IAOutputBlock* output);

    void           DeleteVolumeInfos();
    void           SetVolumeInfos (HFSVolumeInfo** vinfos)
        {volumeInfos = vinfos;}
```

Corpus Category

```

void          SetVolumeCount(short vCount) {volumeCount = vCount;}
HFSVolumeInfo** GetVolumeInfos () const {return volumeInfos;}
short         GetVolumeCount() const {return volumeCount;}

private:
    HFSCorpus(HFSCorpus&);// don't define a copy constructor

    HFSVolumeInfo** volumeInfos;// array mapping from vRefID to HFSVolumeInfo
    short          volumeCount;// length of the array
};

```

Listing 8-4 Sample implementation of GetProtoDoc

```

IADoc*  HFSCorpus::GetProtoDoc() {
    return new HFSDoc;
}

```

Listing 8-5 Sample implementation of GetDocText

```

IADocText* HFSCorpus::GetDocText(const IADoc* d) {
    HFSDoc* doc = (HFSDoc*)d;
    return new HFSDocText(GetVRefNum(doc->GetVolumeRefID()),
                          doc->GetDirID(), doc->GetFileName());
}

```

Creating a Subclass of IADoc

IADoc is the abstract class for the interface to the physical document. Any implementation must contain the data required to locate the actual document. Creating an implementation of IADoc requires a matching implementation of IADocText.

See “IADoc” beginning on page 8-47 for detailed information on this class.

An IADoc is an IAOrderedStorable. See “Creating a Subclass of IAOrderedStorable” on page 10-6 for more information.

Listing 8-6 Sample header of an IADoc subclass

```

class HFSDoc : public IADoc {
public:
    HFSDoc(HFSCorpus* corpus, short vRefNum,
           long dirID,
           const StringPtr name);
    HFSDoc() : fileName(NULL) {}
}

```

Corpus Category

```

virtual                ~HFSDoc();

IAStorable*           DeepCopy();
IABlockSize           StoreSize();
void                  Store(IAOutputBlock* output);
IAStorable*           Restore(IAInputBlock* input);

bool                  LessThan(IAOrderedStorable* neighbor);
bool                  Equal(IAOrderedStorable* neighbor);

// HFSDoc specific
byte*                 GetName(uint32 *length);

void                  SetVolumeRefID(unsigned short vrid)
                      {vRefID = vrid;}
void                  SetDirID(long dID)
                      {dirID = dID;}
void                  SetFileName(StringPtr name)
                      {fileName = name;}
unsigned short         GetVolumeRefID() const {return vRefID;}
long                  GetDirID() const {return dirID;}
StringPtr             GetFileName() const {return fileName;}

protected:
void                  DeepCopying(IAStorable* source);
void                  Restoring(IAInputBlock* input,
                                IAStorable* proto);

private:
                      HFSDoc(HFSDoc& fd);

unsigned short         vRefID;
long                  dirID;
StringPtr             fileName;

```

Creating a Subclass of IADocIterator

The IADocIterator will locate the documents in the corpus in sequence.

See “IADocIterator” beginning on page 8-49 for detailed information on this class.

Required Functions

- GetNextDoc()

Listing 8-7 Sample Header for an IADocIterator subclass

```

class HFSFolderCorpusIterator : public IADocIterator {
public:
    HFSFolderCorpusIterator(HFSTextFolderCorpus* c)
        : corpus(c), hfsIterator(new HFSIterator
            (c->GetVolumeRefNum(),
             c->GetRootDirId())) {}
    ~HFSFolderCorpusIterator() { delete hfsIterator; }
    IADoc*      GetNextDoc();
private:
    HFSTextFolderCorpus* corpus;
    HFSIterator*        hfsIterator;
};

```

Listing 8-8 Sample Implementation of GetNextDoc

```

IADoc* HFSFolderCorpusIterator::GetNextDoc() {
    while (hfsIterator->Increment()) {
        CInfoPBRec* info = hfsIterator->pb;
        if (info->hFileInfo.ioFlFndrInfo.fdType == 'TEXT') {
            return new HFSTextFolderDoc(corpus,
                info->hFileInfo.ioFlParID,
                info->hFileInfo.ioNamePtr,
                info->hFileInfo.ioFlMdDat);
        }
    }
    return NULL;
}

```

Creating a Subclass of IADocText

IADocText provides the text from the actual document. An implementation of this must be able to locate the document, read its content, and translate the content to text.

See “IADocText” beginning on page 8-50 for detailed information on this class.

Required Functions

■ GetNextBuffer()

Listing 8-9 Sample header of an IADocText subclass

```

class HFSDocText : public IADocText {
public:
    HFSDocText() : refNum(0) {}
    HFSDocText(short vRefNum, long dirID,
               const StringPtr name);
    ~HFSDocText();
    IADocText*    DeepCopy() const;

protected:
    void          SetRefNum (short rNum) {refNum = rNum;}
    void          SetTheVolumeRefNum(short vrnum) {theVRefNum = vrnum;}
    void          SetTheDirID(long did) {theDirID = did;}
    void          SetTheFileName(StringPtr name) {theFileName = name;}

    short         GetRefNum () const {return refNum;}
    short         GetTheVolumeRefNum() const {return theVRefNum;}
    long          GetTheDirID() const {return theDirID;}
    StringPtr     GetTheFileName() const {return theFileName;}

private:
    HFSDocText(HFSDocText&); // don't define a copy constructor

    short         refNum;
    short         theVRefNum;
    long          theDirID;
    StringPtr     theFileName;
};

```

Listing 8-10 Sample implementation of GetNextBuffer

```

uint32 HFSDocText::GetNextBuffer(byte* buffer, uint32 bufferLength) {
    long bytes = bufferLength;
    OSErr err = FSRead(refNum, &bytes, buffer);
    if (err && err != eofErr) {
        IAAssertion (false, "cannot read the next buffer", InvalidDocument);
    };
    return bytes;
}

```

Creating a Subclass of HFSIterator

HFSIterator is a utility used for the HFS implementation. It can be subclassed to quickly provide other HFS type corpora.

This iterator will, based on a volume reference number and root directory, locate all base files in that directory. The member function `Increment` will provide the `CBInfoPBRec` information for a file in its member data, `pb`. When there are no more files, the function returns `False`.

See “HFSIterator” beginning on page 8-32 for detailed information on the HFSIterator, and “HFSTextFolderCorpus” beginning on page 8-35 for detailed information on this corpus.

This example shows the use of that iterator with a custom filter to only return files that are text files with the proper suffix (we have chosen an iterator for “.h” header files). A subclass of `IADocIterator` is created to provide this custom iterator.

Listing 8-11 Creating a custom corpus iterator—header file

```
#pragma once
#include "HFSTextFolderCorpus.h"
class HdrCorpus : public HFSTextFolderCorpus {
public:
    HdrCorpus(uint32 type = HFSFolderCorpusType) :
        HFSTextFolderCorpus(type) {}
    HdrCorpus(short vRefNum, long rootDirId, uint32
        type = HFSFolderCorpusType) :
        HFSTextFolderCorpus( vRefNum, rootDirId, type) {}
    HdrCorpus(StringPtr rootDirPath, uint32 type =
        HFSFolderCorpusType) :
        HFSTextFolderCorpus( rootDirPath, type) {}
    // implementing the doc iterator function
    IADocIterator* GetDocIterator();
};
```

Listing 8-12 IADocIterator subclass header

```

#pragma once
#include "HFSIterator.h"
#include "HdrCorpus.h"
#include <string.h>
#include <Files.h>
#include <TextUtils.h> // for RelString
#include <Errors.h>
class HdrDocIterator : public IADocIterator {
public:
    HdrDocIterator(HdrCorpus* c) :
        corpus(c), hfsIterator
            (new HFSIterator(c->GetVolumeRefNum(),
                c->GetRootID())) {}
    ~HdrDocIterator() { delete hfsIterator; }
    IADoc*      GetNextDoc();
private:
    HFSTextFolderCorpus* anHFSTextFolderCorpus;
    HFSIterator*          anHFSIterator;
};

```

Listing 8-13 Corpus subclass body

```

IADocIterator* HdrCorpus::GetDocIterator() {
    return new HdrDocIterator(this);
}

```

Listing 8-14 IADocIterator subclass body

```

IADoc* HdrDocIterator::GetNextDoc() {
    while (anHFSIterator->Increment()) {
        CInfoPBRec* info = anHFSIterator->pb;
        if (info->hFileInfo.ioFlFndrInfo.fdType == 'TEXT') {
            Str255 name;
            uint32 nameLen =
                anHFSIterator->pb->hFileInfo.ioNamePtr[0];
            memcpy(name+1, anHFSIterator->pb->hFileInfo.ioNamePtr+1,
                    nameLen);

            name[0] = nameLen;
            if (name[nameLen] == 'h'
                && name[nameLen-1] == '.' ) {
                return new HFSTextFolderDoc(corpus,
                    info->hFileInfo.ioFlParID,
                    info->hFileInfo.ioNamePtr,
                    info->hFileInfo.ioFlMdDat);
            }
        }
    }
    return NULL;
}

```

Corpus Class Category Reference

Header Files in the Corpus Category

HFSCorpus

DirectoryInfo
HFSCorpus
HFSDoc
HFSDocText
HFSVolumeInfo

HFSIterator

HFSIterator

HFSTextFolderCorpus

HFSTextFolderCorpus
HFSTextFolderDoc

IACorpus

IACorpus
IADoc
IADocIterator
IADocText

Class Specifications

DirectoryInfo

Struct

Struct

Header: HFSCorpus.h

Data

long	id
	the id
short	length
	the number of files

HFSCorpus

Class

Header: HFSCorpus.h

Hierarchy

Public subtype of IACorpus. See “IACorpus” on page 8-41.

Description

A corpus implementation for Macintosh HFS files. HFSCorpus maintains a list of volumes used in the corpus. The volumes are assigned a unique volume ID that persists within IAT. The ID is mapped to the volume reference number. The associated class, HFSDoc, maintains the directory ID and file name.

Relationships

HFSCorpus reads HFSDoc

1 HFSCorpus reads many HFSDoc.

HFSCorpus extracts HFSDocText

HFSCorpus extracts many HFSDocText from each HFSDoc

HFSCorpus contains HFSVolumeInfo

An HFSCorpus contains an array of HFSVolumeInfo

Public Member Functions

constructor

Input

uint32 type = HFSCorpusType

destructor

Deletes volume array.

GetDocText

See “IACorpus.GetDocText” on page 8-44.

Usage

```
HFSDocText* bestTxt =
    (HFSDocText*)sindex.corpus->GetDocText(bestHFSDoc);
```

GetProtoDoc

See “IACorpus.GetProtoDoc” on page 8-44. HFSCorpus uses HFSDoc as its prototype.

GetVRefID

Input

short vRefNum
The HFS Volume reference number.

Output

unsigned short
The logical volume ID used in IAT.

Usage

```
unsigned short vRefId = corpus->GetVRefID(vRefNum);
```

GetVRefNum

Input

unsigned short vRefID
The logical reference ID assigned by IAT.

Output

short
The HFS volume reference number.

Usage

```
short vRefNum = corpus->GetVRefNum(doc->GetVolumeRefID());
```

Protected Member Functions

GetVolumeCount

Access method for HFSCorpus member data.

Corpus Category

Output

short volumeCount
 Length of the volume ID array.

GetVolumeInfos

Access method for HFSCorpus member data.

Output

HFSVolumeInfo**volumeInfos
 Array mapping from every vRefID to a HFSVolumeInfo.

Initializing

See “IACorpus.Initializing” on page 8-46. Establishes volume info array in storage.

InitialSize

See “IACorpus.InitialSize” on page 8-46. Computes size of volume info array.

Opening

See “IACorpus.Opening” on page 8-46. Reads volume array from storage.

SetVolumeCount

Access method for HFSCorpus member data.

Input

short volumeCount
 Length of the volume ID array.

SetVolumeInfos

Access method for HFSCorpus member data.

Input

HFSTVolumeInfo**volumeInfos

Array mapping from every vRefID to a HFSTVolumeInfo.

UpdateSize

See “IACorpus.UpdateSize” on page 8-47. Computes new size of volume array.

Updating

See “IACorpus.Updating” on page 8-47. Writes volume array to storage.

HFSDoc

Class

Header: HFSCorpus.h

Hierarchy

Public subclass of IADoc. See “IADoc” on page 8-47.

Client

See “HFSCorpus reads HFSDoc” on page 8-17.

Public Member Functions

constructor

constructor(HFSCorpus* corpus, short vRefNum, long dirID, const StringPtr name)

Input

HFSCorpus* corpus
The associated corpus.

short vRefNum
The HFS volume reference number of the volume where the file resides.

long dirID
The HFS directory ID of the file.

const StringPtr name
The HFS filename.

Usage

```
HFSDoc doc1(&corpus, vRefNum, dirID, name);
```

destructor

Virtual.

DeepCopy

See “IAStorable.DeepCopy” on page 10-28.

Equal

See “IAOrderedStorable.Equal” on page 10-15. HFSDocs are keyed and ordered by logical volume ID and directory ID, not by filename.

GetDirID

Access method for HFSDoc member data.

Output

long	dirID
	The HFS directory ID of the file.

GetFileName

Access method for HFSDoc member data.

Output

StringPtr	fileName
	The HFS file name (not the full path). Allocated with IAMallocArraySized. Use IAFreeArraySized to free.

GetName

See “IADoc.GetName” on page 8-48. Returns the file name, null terminated.

GetVolumeRefID

Access method for HFSDoc member data.

Output

unsigned short	tvRefID
	The logical volume reference ID assigned by IAT. Use the HFSCorpus GetVRefNum() function to get the HFS volume reference number.

LessThan

See “IAOrderedStorable.LessThan” on page 10-16. HFSDocs are ordered by volumeID and directoryID, not filename.

Restore

See “IAStorable.Restore” on page 10-28.

SetDirID

Access method for HFSDoc member data.

Input

long	dirID
	The HFS directory ID of the file.

SetFileName

Access method for HFSDoc member data.

Input

StringPtr	fileName
	The HFS file name (not the full path).

SetVolumeRefID

Access method for HFSDoc member data.

Input

unsigned short	tvRefID
	The logical volume reference ID assigned by IAT.

Store

See “IAStorable.Store” on page 10-30.

StoreSize

See “IAStorable.StoreSize” on page 10-29.

Protected Member Functions

DeepCopying

See “IAStorable.DeepCopying” on page 10-30.

Restoring

See “IAStorable.Restoring” on page 10-31.

HFSDocText

Class

Header: HFSCorpus.h

Hierarchy

Public subclass of IADocText. See “IADocText” on page 8-50.

Client

See “HFSCorpus extracts HFSDocText” on page 8-18.

Public Member Functions

constructor

constructor (short vRefNum, long dirID, const StringPtr name)

Input

short vRefNum
The HFS volume reference number of the volume on which the document file resides.

long dirID
The HFS directory ID of the file.

const StringPtr name
The HFS name of the file.

Notes

Opens the document file.

Usage

```
return new
HFSDocText(corpus->GetVRefNum(doc->GetVolumeRefID()),
            doc->GetDirID(), doc->GetFileName())
```

destructor

GetNextBuffer

See “IADocText.GetNextBuffer” on page 8-50. Reads the document file.

Protected Member Functions

GetRefNum

Access method for HFSDocText member data.

Output

short	refNum
	The path reference number returned when the access to the data fork was opened.

GetTheDirID

Access method for HFSDocText member data.

Output

long	theDirID
	The HFS directory ID.

GetTheFileName

Access method for HFSDocText member data.

Output

StringPtr	theFileName
	The HFS file name.

GetTheVolumeRefNum

Access method for HFSDocText member data.

Output

short	theVRefNum
	The HFS volume reference number.

SetRefNum

Access method for HFSDocText member data.

Input

short	refNum
	The path reference number returned when the access to the data fork was opened.

SetTheDirID

Access method for HFSDocText member data.

Input

long	theDirID
	The HFS directory ID.

SetTheFileName

Access method for HFSDocText member data.

Input

StringPtr	theFileName
	The HFS file name.

SetTheVolumeRefNum

Access method for HFSDocText member data.

Input

short	theVRefNum
	The HFS volume reference number.

HFSVolumeInfo

Class

Header: HFSCorpus.h

Hierarchy

Public subclass of IASTorable. See “IAStorable” on page 10-27.

Description

HFSVolumeInfo is used to a map of the volume reference numbers to the creationDate and Name of a volume. The creation date and name of the volume are persistent; the volume reference number may vary over time if the system has been rebooted.

The HFSCorpus maintains a map of the HFSVolumeInformation to the internally used vRefID.

When restored, HFSVolumeInfo locates the current volume reference number for the volume name and creation date.

Client

See “HFSCorpus contains HFSVolumeInfo” on page 8-18.

Public Member Functions

constructor

constructor(short vRefNum)

Input

short vRefNum

The HFS volume reference number.

Usage

```
newVolumeInfos[volumeCount] = new HFSVolumeInfo(vRefNum)
```

DeepCopy

See “IAStorable.DeepCopy” on page 10-28.

GetCreationDate

Access method for HFSVolumeInfo member data.

Output

long	creationDate
	Volume creation date (persistent).

GetVolumeName

Access method for HFSVolumeInfo member data.

Output

StringPtr	name
	Volume name (persistent).

GetVolumeRefNum

Access method for HFSVolumeInfo member data.

Output

short	vRefNum
	Volume reference number (persistent).

Restore

See “IAStorable.Restore” on page 10-28.

SetCreationDate

Access method for HFSVolumeInfo member data.

Input

long	creationDate
	Volume creation date (persistent).

SetVolumeName

Access method for HFSVolumeInfo member data.

Input

StringPtr	name
	Volume name (persistent).

SetVolumeRefNum

Access method for HFSVolumeInfo member data.

Input

short	vRefNum
	Volume reference number (persistent).

Store

See “IAStorable.Store” on page 10-30.

StoreSize

See “IAStorable.StoreSize” on page 10-29.

Used to restore HFS Volume Info from storage.

Class HFSIterator

Header: HFSIterator.h

Hierarchy

Base Class

Description

HFSIterator is built to return any file from a given volume and directory. It will recurse all folders to get to the actual files.

This can be used to determine which files, given a volume and directory, will be included in the corpus. HFSTextFolderCorpus, for example, uses this iterator to retrieve files then only includes text files.

Client

See “HFSTextFolderCorpus transverses folders using HFSIterator” on page 8-35.

Public Member Functions

constructor (short vRefNum, long rootDirId = 2)

Input

short vRefNum

The HFS volume reference number.

long rootDirId = 2

The directory ID of the highest level folder. Default is the volume root.

Usage

```
HFSIterator* hfsIterator =
    new HFSIterator(c->GetVolumeRefNum(),
        c->GetRootDirId());
```

destructor

GetPBRec

Access method for HFSIterator member data.

Output

CInfoPBRec* pb
Parameter block containing HFS file information. See *Inside Macintosh, Files*.

GetDir

Access method for HFSIterator member data.

Output

uint32 dir
Index into array of directory infos, representing the root-level directory being processed.

GetDirCount

Access method for HFSIterator member data.

Output

long dirCount
The number of root-level directories in the directory info array.

GetDirIndex

Access method for HFSIterator member data.

Output

short dirIndex
Index into directory being processed.

GetDirInfos

Access method for HFSIterator member data.

Output

DirectoryInfo* dirInfos
Sorted array of directory IDs for current volume.

Increment

Output

bool
True if a file has been found. File information will be in HFSIterator->pb.
False if there are no more files within the folders.

Notes

Locates the next available file within the structure and places it in member data pb.

Usage

```
while (hfsIterator->Increment())
```

Listing 8-15 Using HFSIterator

```
while (hfsIterator->Increment()) {
    CInfoPBRec* info = hfsIterator->pb;
    if (info->hFileInfo.ioFlFndrInfo.fdType == 'TEXT') {
        // filter out non-text documents
        return new HFSTextFolderDoc(corpus,
                                    info->hFileInfo.ioFlParID,
                                    info->hFileInfo.ioNamePtr,
                                    info->hFileInfo.ioFlMdDat);
    }
}
```

SetDirIndex

Access method for HFSIterator member data.

Input

short dirIndex
Index into directory being processed.

Protected Member Functions

CollectDirInfo

Builds a table of all the directory IDs in the named directory and all its subdirectories.

HFSFolderCorpus

Class

Header: HFSFolderCorpus.h

Hierarchy

Public subclass of HFSIterator. See “DirectoryInfo” on page 8-17.

Description

A corpus implementation for all the text files under a root HFS folder.

Relationships

HFSFolderCorpus reads HFSFolderDoc

1 HFSIterator reads many HFSFolderDoc.

HFSFolderCorpus transverses folders using HFSIterator

One corpus may use many iterators.

Constants

```
const uint32    HFSFolderCorpusType = 'HTF1'
```

Public Member Functions

constructor (uint32* type)

Input

```
uint32* type = HFSFolderCorpusType
    The type of corpus.
```

Notes

Initializes only.

constructor (short vRef, long rootDirID, uint32 type = HFSFolderCorpusType);

Input

```
short vRef
    The HFS volume reference number of the folder.
long rootDirID
    The HFS directoryID of the folder.
uint32* type = HFSFolderCorpusType
    The type of corpus.
```

Notes

Builds the corpus by iterating thorough the files in the folder represented by the reference number and directory ID.

Usage

```
HFSTextFolderCorpus* corpus =
    new HFSTextFolderCorpus(vrefNum, rootDirID);
```

constructor(StringPtr rootDirPath, uint32 type = HFSFolderCorpusType);

Input

StringPtr rootDirPath
The full path name to the folder. Do not end in a colon.

uint32*type = HFSFolderCorpusType
The corpus type.

Usage

```
StringPtr folderName="\pHD:docs";
HFSTextFolderCorpus* corpus =
    new HFSTextFolderCorpus(folderName)
```

GetDocIterator

See “IACorpus.GetDocIterator” on page 8-43. The HFS Text Folder doc iterator uses HFSIterator and only returns files of type “TEXT.”

GetProtoDoc

See “IACorpus.GetProtoDoc” on page 8-44. Uses HFSTextFolderDoc.

GetRootDirID

Access method for HFSTextFolderCorpus member data.

Output

long rootDirID
The HFS directory ID of the folder.

GetVolumeRefNum

Access method for HFSTextFolderCorpus member data.

Corpus Category

Output

short	vRefNum
	The HFS volume reference number of the volume where the folder resides.

Protected Member Functions

Initializing

See “IACorpus.Initializing” on page 8-46.

InitialSize

See “IACorpus.InitialSize” on page 8-46.

Opening

See “IACorpus.Opening” on page 8-46.

SetRootDirID

Access method for HFSTextFolderCorpus member data.

Input

long	rootDirID
	The HFS directory ID of the folder.

SetVolumeRefNum

Access method for HFSTextFolderCorpus member data.

Input

short	vRefNum
	The HFS volume reference number of the volume where the folder resides.

UpdateSize

See “IACorpus.UpdateSize” on page 8-47.

Updating

See “IACorpus.Updating” on page 8-47.

HFSTextFolderDoc

Class

Header: HFSTextFolderCorpus.h

Hierarchy

Public subclass of HFSDoc. See “HFSDoc” on page 8-21.

Client

See “HFSTextFolderCorpus reads HFSTextFolderDoc” on page 8-35.

Public Member Functions

constructor

constructor

Input

HFSTextFolderCorpus* corpus	The corpus controlling this document.
long dirID	The document file’s HFS directory ID (ioFLParID)

Corpus Category

```
const StringPtr name
    The HFS file name of the document.
long date
    The last modification date of the document.
```

DeepCopy

See “IAStorable.DeepCopy” on page 10-28.

Equal

See “IAOrderedStorable.LessThan” on page 10-16. This uses logical volume ID, directory ID, filename and modification date as the key information.

GetModDate

Access method for HFSTextFolderDoc member data.

Output

```
long      modDate
    The modification date of the document.
```

LessThan

See “IAOrderedStorable.LessThan” on page 10-16. This corpus uses logical volume ID, directory ID, filename and modification date as the key information.

Restore

See “IAStorable.Restore” on page 10-28.

SetModDate

Access method for HFSTextFolderDoc member data.

Corpus Category

Input

long **modDate**
 The modification date of the document.

Store

See “IAStorable.Store” on page 10-30.

StoreSize

See “IAStorable.StoreSize” on page 10-29.

Protected Member Functions

DeepCopying

See “IAStorable.DeepCopying” on page 10-30.

Restoring

See “IAStorable.Restoring” on page 10-31.

IACorpus

Class

Header: IACorpus.h

Hierarchy

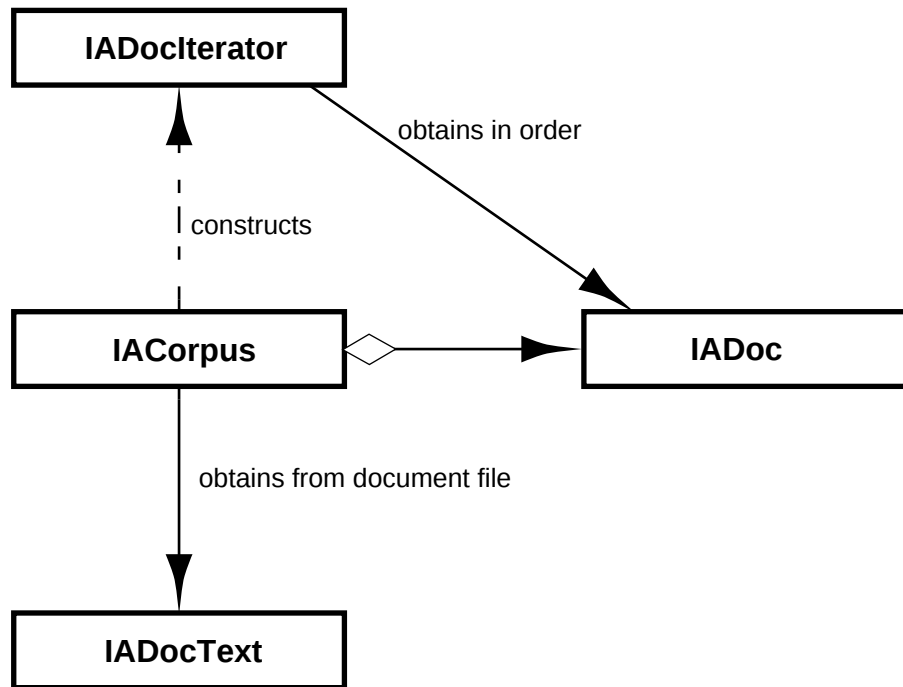
Abstract Base Class

Description

IACorpus serves as the major interface between the actual documents and the index. It characterizes a document collection. It locates the text in the documents.

Relationships

Figure 8-4 IACorpus relationships



IACorpus constructs IADocIterator

One Corpus may construct any number of iterators.

IACorpus obtains from document file IADocText

One corpus may obtain several IADocText.

IACorpus contains IADoc

One corpus may contain many IADoc

Public Member Functions

constructor(type)

Input

`uint32 type`
The constant for the corpus type being created.

Usage

(HFSCorpus is a subclass)

```
InvertedIndex index(storage,
                      new HFSCorpus(HFSCorpusType),
                      new SimpleAnalysis());
```

GetCorpusType

Access method for IACorpus member data.

Output

`uint32 corpusType`
The type of the corpus. This is maintained to allow the reconstruction of an already established corpus with the correct subclass.

GetDocIterator

Virtual.

Output

`IADocIterator*`
An object which obtains the documents of the corpus.

Notes

Determines set of documents to be indexed by the ones it chooses to locate.

Corpus Category

Usage

```
IADocIterator* corpusDocs = corpus->GetDocIterator();
```

GetDocText

Pure virtual.

Input

```
const IADoc* doc
```

A document contained in the corpus.

Notes

Accesses the text of a document.

Usage

```
IATokenStream* ts = index->analysis->
    MakeTokenStream(index->corpus->GetDocText(doc));
```

GetProtoDoc

Pure virtual.

Output

```
IADoc*
```

An initialized object of the type used in the corpus.

Notes

Used to establish sets based on the Doc type used in the corpus.

Usage

```
docInfoSet = IAMakeOrderedStorableSet
    (MakeDocInfo(corpus->GetProtoDoc(), 0));
```

Initialize

Input

IAStorage* storage
A pointer to the storage established and initialized for the corpus.

IABlockID corpusRoot
The root id for the corpus.

Usage

```
corpusRoot = storage->Allocate();
corpus->Initialize(storage, corpusRoot);
```

Open

Input

IAStorage*
A pointer to the storage established and opened for the corpus.

IABlockID
The root id for the corpus.

Notes

Restores corpus information from storage.

Usage

```
corpusRoot = input->ReadUInt32();
corpus->Open(storage, corpusRoot);
```

Update

Input

IAStorage* storage
A pointer to the storage established for the corpus. Storage must be open and writable.

IABlockID corpusRoot
The root id for the corpus.

Corpus Category

Notes

Writes changed corpus information to storage.

Usage

```
corpus->Update(storage, corpusRoot);
```

Protected Member Functions

Initializing

Virtual.

Input

IAOutputBlock output

Notes

Used to implement Initialize().

InitialSize

Virtual.

Output

IABlockSize

Notes

Used to implement Initialize().

Opening

Virtual.

Corpus Category

Input

IAInputBlock input

Notes

Used to implement Open().

UpdateSize

Virtual.

Output

IABlockSize

Notes

Used to implement Update().

Updating

Virtual.

Input

IAOutputBlock output

Notes

Used to implement Update().

IADoc

Class

Header: IACorpus.h

Hierarchy

Abstract Base Class, Subclass of IAOrderedStorable. See “IAOrderedStorable” on page 10-14.

Clients

See “IACorpus contains IADoc” on page 8-42.

See “IADocIterator obtains IADoc in order” on page 8-49.

See “IAHit finds matching IADoc located in IAIndex” on page 6-27.

See “IAProgressReport reports which IADoc is being processed” on page 6-30.

See “RankedQueryDoc connects a sample IADoc to its location in a TermIndex” on page 6-47.

Public Member Functions

constructor

GetName

const

Virtual.

Input

uint32* length

Returned length of the name.

Output

byte* name

Pointer to the name array.

Notes

Returns the name of a document. This will return NULL, and set its input parameter to 0, unless implemented by its subclass.

Returned array is allocated by IAMallocArray() and should be freed by IAFreeArray().

Name is null terminated.

Usage

```
uint32 length = 0;
byte* name = doc.GetName(length);
```

IADocIterator

Class

Header: IACorpus.h

Hierarchy

Abstract Base Class

Relationships

IADocIterator obtains IADoc in order

One iterator obtains many documents.

Client

See “IACorpus constructs IADocIterator” on page 8-42.

Public Member Functions

GetNextDoc

Pure virtual.

Output

IADoc* corpusDoc

The next document in the set. NULL if at the end of the set.

Notes

Advances the iterator to the next document in a set and returns it.
 The documents are returned in sequence, that is, the first document returned is the lowest in the set, the next the second lowest, and so on until all have been returned.
 IADoc* is NULL at the end of the set.
 Returns a new copy of the document. Clients must delete.

Corpus Category

Usage

```
IADoc* corpusDoc = CorpusDocs->GetNextDoc();
```

Class IADocText

Header: IACorpus.h

Hierarchy

Abstract Base Class

Public Member Functions

constructor

GetNextBuffer

Pure Virtual.

Input

```
byte* buffer
    Pointer to the text buffer.
uint32 bufferLen
    Buffer size.
```

Output

```
uint32
    Number of bytes placed in the buffer.
```

Notes

Extracts successive segments of the text of the document.
Returns number of bytes written into buffer.
Returns zero at end of document.

Corpus Category

Usage

```
uint32 bytesRead=docText->GetNextBuffer((byte*)buffer, bufferLen);
```

Constants

```
const uint32HFSCorpusType = 'HFS0'
const uint32HFSFolderCorpusType = 'HTF1'
```

Exceptions

VCHV

HFSVolumeNotFound

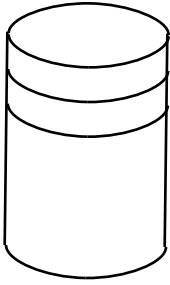
VCHE

HFSError

VCID

Invalid document.

Corpus Category



Storage Category

General Storage Logic	3
HFS Implementation	5
Creating New Storage	5
Sample Code to Create Storage	5
Opening Existing Storage	6
Sample Code for Establishing Existing Storage	6
Allocating and Deallocating Blocks of Storage	6
Reading and Writing Storage	8
Reporting on Storage	8
Compacting Storage	8
Using the Mutex Facility	9
Cloning Store Streams	10
Creating Storage Subclasses	10
Creating a Storage Construction Utility	10
Creating a Subclass of IASToreStream	11
Creating a Subclass of IAMutex	15
Storage Class Category Reference	17
Header Files in the Storage Class Category	17
Class Specifications	18
HFSSoreStream	18
IAInputBlock	22
IAOutputBlock	25
IALock	28

Storage Category

IAMutex	28
IAStorage	30
IAStoreStream	39
Storage Class Utilities	45
Typedefs	48
Storage Exceptions and Error Handling	50

Storage Category

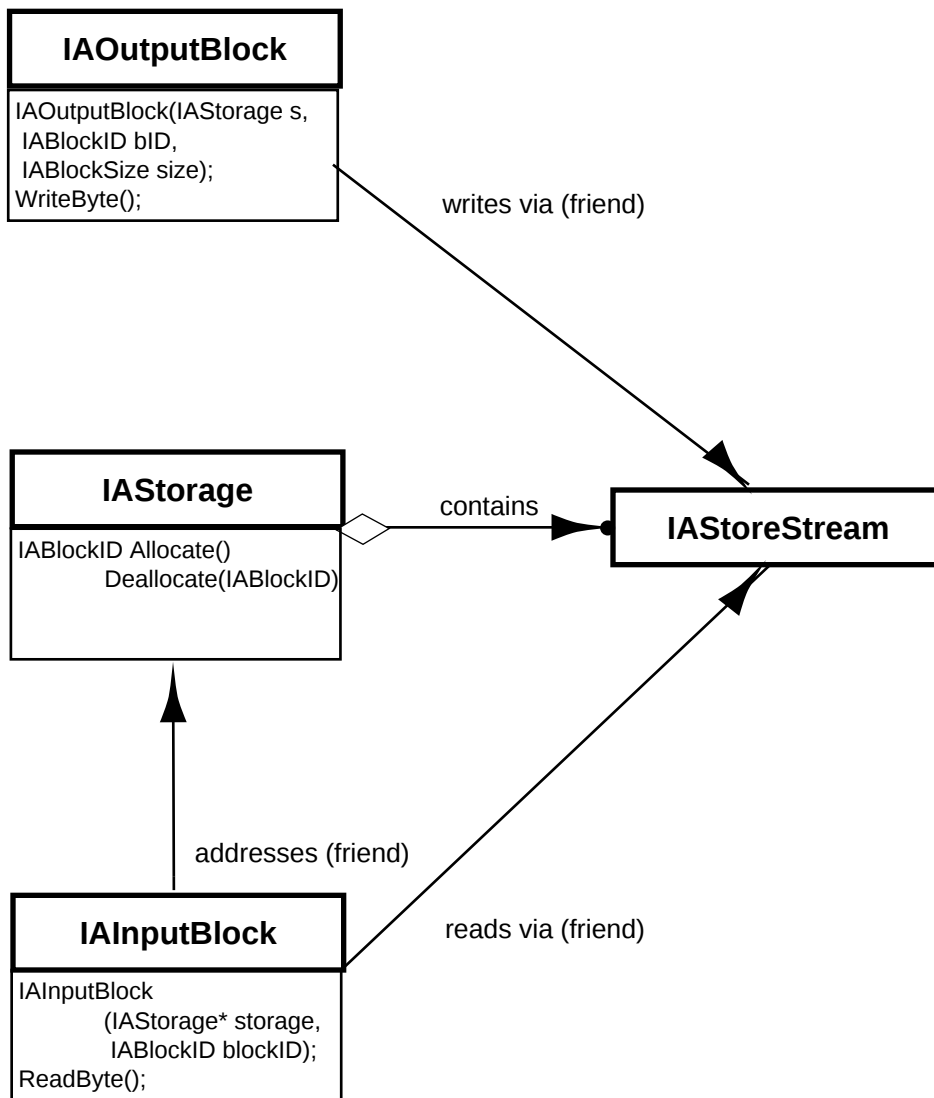
IAT provides classes to allow the storage of blocks of data into persistent storage. This storage is used by IAT to hold the information access indexes and structures. Indexes require persistent storage; this set of logical storage classes provides an interface to the storage media desired to hold the index information. Developers may also use these storage classes to store other data they wish to make persistent.

General Storage Logic

Figure 9-1 illustrates the relationships of the storage classes.

IStorage is managed in blocks. These blocks have ID Numbers which are stored within the storage class.

Items are written through the IAOutputBlock, which in turn uses the I/O functions of IASToreStream to write. Similarly, the IAInputBlock reads items through theIAStoreStream.

Figure 9-1 Logical storage classes

General use of IAT requires no internal knowledge of the storage. You create and open storage, then create information access classes to be stored in this storage. Updates to the IAT objects occur in memory. The storage is committed to disk after completion of the processing. This prevents damaged files due to incomplete processing.

IAT also provides member functions to allow you to see the amount of storage used for a file and to compact the file.

HFS Implementation

The IAT storage architecture is designed to be platform-independent. Platform-specific subclass implementations may be used to optimize performance. IAT provides a MacOS-specific implementation of storage that uses the Macintosh HFS file system. This implementation will be used for examples. Applications which use other storage types may create subclasses of the IAT abstract classes to interface to that storage.

Creating New Storage

Create storage with a utility rather than the direct use of a constructor. See “IAMakeStorage” on page 9-45 for more information.

MakeHFSStorage is an implementation of that utility which constructs storage for an HFS file. You must know the HFS volume, directory and file name before you can construct HFS storage.

Following creation, initialize the storage for use. This initialization creates the structures used to address blocks and opens the storage for writing.

Sample Code to Create Storage

Listing 9-1 Constructing storage

```
#pragma once
#include <Types.h>
#include "HFSStorage.h"

// Client must provide these values:
short      vRefNum = 0;
long       dirID = 0;
StringPtr  storageFileName = "\\pstorage.file";

// create storage
IAStorage* anIAStorage = MakeHFSStorage (vRefNum,
                                         dirID, storageFileName);
anIAStorage->Initialize();
```

Opening Existing Storage

Opening an existing storage requires a storage object and restores data from persistent storage to the object.

Storage may be opened as read-only or read and write access. `Open(True)` will allow writes.

Sample Code for Establishing Existing Storage

Listing 9-2 Establish existing storage

```
#pragma once
#include <Types.h>
#include "HFSStorage.h"

// Client must provide these values:
short      vRefNum = 0;
long       dirID = 0;
StringPtr  storageFileName = "\pstorage.file;

bool writable = true;

// create storage
IAStorage* anIAStorage = MakeHFSStorage (vRefNum,
                                         dirID, storageFileName);
anIAStorage->Open(writable);
```

Allocating and Deallocating Blocks of Storage

The base unit of storage is a block. A block is a contiguous set of data that is written or read from storage as a whole. Individual bytes, words, or strings are accessed in the block once it is in memory.

A block has a block ID that uniquely identifies it. This ID is of type `IABlockID`.

The storage object maintains a table of allocated blocks that maps each block to a specific location in physical storage. Objects using storage must know which block contains their desired data. They can do this by maintaining their own table of contents of storage, or they can request a named block in the internal storage table of contents and keep track of

Storage Category

that block name rather than its ID. In this case, the storage maintains an internal table, known as the TOC (for "Table Of Contents"), which maps the block names to block IDs.

The following example allocates new HFSStorage by a named block. When a block of storage is first created, it is always an output block, which will allow data to be written to the block.

Listing 9-3 Allocating a named block of storage

```
// create storage
IAStorage* anHFSStorage = MakeHFSStorage(vRefNum, dirID, fileName);
anHFSStorage ->Initialize();
const char* aBlockName = "MY NAMED BLOCK";

// ask for a new block to be labeled with the given name
IABlockID anIABlockID = anHFSStorage->AllocateNamedBlock(aBlockName);
IAOutputBlock anIAOutputBlock(anHFSStorage, anIABlockID,
                               anIABlockSize);
```

The sample listing below establishes a named block of storage.

Listing 9-4 Opening a named block of storage

```
// create storage object
bool writable = true;
IAStorage* anHFSStorage = MakeHFSStorage(vRefNum, dirID, fileName);
anHFSStorage ->Open(writable);

// get the pre-defined block ID
const char* aBlockName = "MY NAMED BLOCK";
IABlockID anIABlockID = anHFSStorage->TOC_Get(aBlockName);
IAInputBlock anIAInputBlock(anHFSStorage, anIABlockID);
```

Storage can be allocated directly without using a named block by the Allocate() function. This returns a block ID which the application must keep track of.

Storage is deleted by deallocating a block using the Deallocate(anIABlockID) function for unnamed blocks, or the RemoveNamedBlock(blockName) function for named blocks.

▲ **WARNING**

If you use Deallocate to delete a named block (instead of RemoveNamedBlock), you will leave the TOC entry for that name untouched. Unless you do a matching TOC_Remove, you will render that name unusable for the remaining life of the index.

Reading and Writing Storage

Blocks of storage are accessed through objects of the class `IAOutputBlock` or `IAInputBlock`. `IAOutputBlock` is a class to write the storage. It accesses the appropriate `IASoreStream` implementation for the class. `IAInputBlock` reads the storage through the store stream.

Note

No changes are made to persistent storage until the storage has been committed by the `Commit()` function of the `IStorage` class.

`IAInputBlock` read functions:

- `byte ReadByte()`
- `uint32 ReadUInt32()`
- `void ReadBuffer(void* aBuffer, uint32 length)`

`IAOutputBlock` write functions:

- `void WriteByte(byte b)`
- `void WriteUInt32(uint32 i)`
- `void WriteBuffer( void* aBuffer, uint32 length)`

Reporting on Storage

There are member functions which return the amount of total space used by the storage (`TotalSpace()`) and the amount of that total space which is free space (`FreeSpace()`).

Listing 9-5 Report amount of space in storage

```
printf ("%lu Total Space\n", anIStorage->TotalSpace());
printf ("%lu Free Space\n\n", anIStorage->FreeSpace());
```

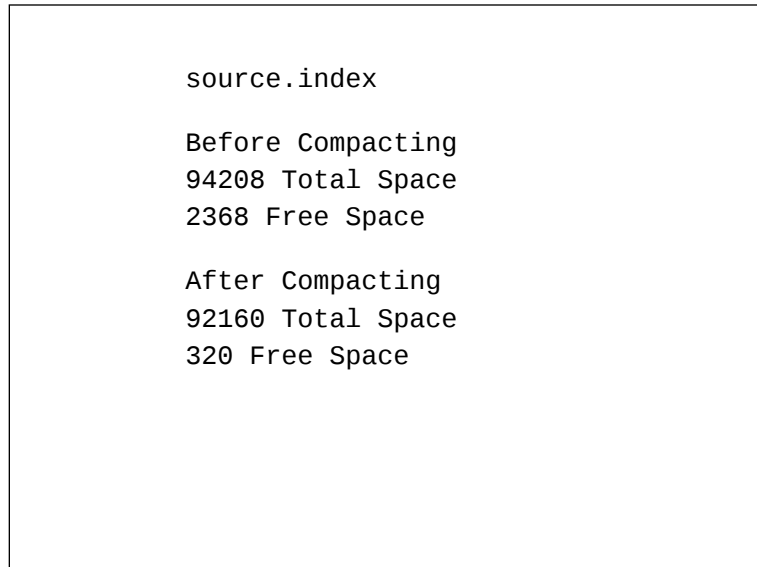
Compacting Storage

Storage that has been maintained extensively may develop fractured spots of free space within the allocated blocks. Compacting the storage will eliminate this free space and reduce the total size of storage. You must establish the storage and open it as writable before compacting. `Compact()` does the commit to storage; you do not need to commit

Storage Category

storage after it has been compacted. Figure 9-2 shows how an application might report the results of compacting storage.

Figure 9-2 A sample result of compacting storage



Using the Mutex Facility

A MUtual EXclusion semaphore, or mutex, allows you to control access to the storage when you are using multi-threaded applications. Although this presents no problem when reading storage, there are many times when writing to storage within the IAT functions that require access to storage be single-threaded to prevent lock-outs and accidental override of storage. IAT has the logic in place to create and use these semaphores to prevent this multiple access for its functions.

There is no implementation for the Mutex classes, however. If the application may be run in multiple threads, the developer must create an implementation for IAMutex and IALock. In addition, the application must ensure the mutex is invoked for any additional areas of the application where multi-thread access must be controlled.

A mutex is established using the extern IANewMutex().

```
anIAMutex = IANewMutex();
```

It is locked by creating an instance of IALock for the mutex:

```
IALock anIALock(anIAMutex);
```

Storage Category

Destructing the mutex or the lock releases the lock.

Cloning Store Streams

In order to provide additional support for multithreaded applications, IAT offers a way for each thread to get its own copy of the same store stream. This is called "cloning." With cloned streams, threads do not have to wait for each other to do disk access. Several of the storage classes, such as `IAOutputBlock`, have optional parameters for using a cloned stream.

All subclasses of `IASoreStream` must implement a `Clone()` method to support cloning. Cloning is not used by single-threaded applications.

Creating Storage Subclasses

You may need to create a storage subclass if your persistent storage needs to be based on something other than the Macintosh HFS file system.

The `IAStorage`, `IAInputBlock`, and `IAOutputBlock` classes will not require a specialized subclass. You will need to subclass `IASoreStream`, and you will need to create a new utility to construct your storage.

Creating a Storage Construction Utility

Storage is created by creating a store stream, then an object of `IAStorage`. There is a default construction utility, `IAMakeStorage(IASoreStream* anIASoreStream)` that must be invoked to construct storage. By supplying your file type's store stream, you effectively create your file types storage subclass. The following listing shows a storage construction utility built to create HFS storage.

Listing 9-6 A utility to construct storage

```
#include "Storage.h"
#include <Types.h>

IAStorage* MakeHFSStorage(short vRefNum, long dirID,
    const StringPtr fileName,
    OSType creator = 'VTWN',
    OSType fileType = 'STOR')
```

Storage Category

```
{
    return IAMakeStorage(new HFSStoreStream(vRefNum,
        dirID, fileName, creator, fileType));
}
```

Creating a Subclass of IASToreStream

IAStoreStream requires a subclass as it does the actual storage input and output. A specific subclass of this abstract base class is required to support the actual storage I/O for a specific platform.

See “IAStoreStream” on page 9-39 for detailed information. Listing 9-7 through Listing 9-16 show the HFS implementation of IASToreStream and its functions as an example.

Required Functions

- Clone
- IsOpen
- IsWritable
- Initialize
- Open
- GetEOF
- SetEOF
- Write
- Read

Listing 9-7 Sample header file of an IASToreStream subclass

```
#include "IAStoreStream.h"
#include <Files.h>

class HFSStoreStream : public IASToreStream {
public:
    HFSStoreStream(short vRefNum, long dirID, const StringPtr fileName,
        OSType creator = 'VTWN', OSType fileType = 'STOR');
    ~HFSStoreStream();

    void    Initialize();
    void    Open(bool writable);

    bool    IsOpen();
```

Storage Category

```

bool    IsWritable();

void    Flush();

uint32  GetEOF();
void    SetEOF(uint32 address);

virtual IASToreStream* Clone();

// Access methods for private member data
OSType   GetCreator() const {return creator;}
OSType   GetFileType() const {return fileType;}
const short GetVRefNum() const {return vRefNum;}
const long GetDirID() const {return dirID;}
StringPtr GetFileName() const {return fileName;}
short    GetFRefNum() const {return fRefNum;}
void     SetFRefNum(short fref) {fRefNum = fref;} // better be open!

protected:
    // constructor for use by Clone()
    HFSStoreStream(short vRef, long dirId, const StringPtr fileName,
                   OSType creator, OSType fileType, bool isOpen,
                   bool isWritable,
                   short fRefNum);

    void    Write(uint32 address, byte* data, uint32 length);
    uint32  Read(uint32 address, byte* data,  uint32 length);

private:
    bool     isOpen;
    bool     isWritable;

    const OSType creator;
    const OSType fileType;

    const short vRefNum;
    const long  dirID;
    StringPtr  fileName;

    // handle on the open file
    short      fRefNum;
};

```

Listing 9-8 Sample implementation of Clone()

```
IAStoreStream* HFSStoreStream::Clone() {
    return new HFSStoreStream(vRefNum, dirID, fileName,
                              creator, fileType, isOpen, isWritable,
                              fRefNum);
}
```

Listing 9-9 Sample implementation of IsWritable()

```
bool HFSStoreStream::IsWritable() {
    return isWritable;
}
```

Listing 9-10 Sample implementation of IsOpen()

```
bool HFSStoreStream::IsOpen() {
    return isOpen;
}
```

Listing 9-11 Sample implementation of Initialize()

```
void HFSStoreStream::Initialize() {
    IALock lock(mutex); // mutex created upon construction of IAStoreStream
    OSErr err = HCreate(vRefNum, dirID, fileName, creator, fileType);
    if (err == dupFNErr) { // already exists
        short fRef;
        err = HOpenDF(vRefNum, dirID, fileName, fsRdWrPerm, &fRef);
        IAAssertion(!err, "unable to open existing HFS file", StoreError);
        err = ::SetEOF(fRef, 0); // reset data fork
        IAAssertion(!err, "unable to reset data fork", StoreError);
        err = FSClose(fRef);
        IAAssertion(!err, "unable to close HFS file", StoreError);
    } else IAAssertion(!err, "unable to create HFS file", StoreError);
}
```

Listing 9-12 Sample implementation of Open()

```

void      HFSStoreStream::Open(bool forWrite) {
    IALock lock(GetMutex());
    IAAssertion(!isOpen, "store stream already open", StoreError);
    short fRef;
    OSerr err = HOpenDF(vRefNum, dirID, fileName,
                        forWrite ? fsRdWrPerm : fsRdPerm, &fRef);
    IAAssertion(!err, "can't open data fork for store stream", StoreError);
    fRefNum = fRef;
    isOpen = true;
    isWritable = forWrite;
}

```

Listing 9-13 Sample implementation of GetEof()

```

uint32    HFSStoreStream::GetEOF() {
    IALock lock(GetMutex());
    IAAssertion(isOpen, "store stream NOT Open", StoreError);
    long eof;
    OSerr err = ::GetEOF(fRefNum, &eof);
    IAAssertion(!err, "not able to get EOF", StoreError);
    return eof;
}

```

Listing 9-14 Sample implementation of SetEof()

```

void      HFSStoreStream::SetEOF(uint32 address) {
    IALock lock(GetMutex());
    IAAssertion((isOpen && isWritable),
                "store stream not open or writeable", StoreError);
    OSerr err = ::SetEOF(fRefNum, address);
    IAAssertion(!err, "unable to set EOF", StoreError);
}

```

Listing 9-15 Sample implementation of Write()

```

void HFSStoreStream::Write(uint32 address, byte* data, uint32 length) {
    IAAssertion((isOpen && isWritable),
                "store stream closed or read-only", StoreError);
    ParamBlockRec pb;
}

```

Storage Category

```

pb.ioParam.ioCompletion = NULL;
pb.ioParam.ioRefNum = fRefNum;
pb.ioParam.ioBuffer = (Ptr)data;
pb.ioParam.ioReqCount = length;
pb.ioParam.ioPosMode = fsFromStart;
pb.ioParam.ioPosOffset = address;
OSErr err = PBWriteSync(&pb);
IAAssertion(!err, "unable to write", StoreError);
IAAssertion(pb.ioParam.ioActCount == length,
            "actual write not equal length", IAAssertionFailure);
}

```

Listing 9-16 Sample implementation of Read()

```

uint32 HFSStoreStream::Read(uint32 address, byte* data, uint32 length) {
    IAAssertion(isOpen, "store stream not open", StoreError);
    ParamBlockRec pb;
    pb.ioParam.ioCompletion = NULL;
    pb.ioParam.ioRefNum = fRefNum;
    pb.ioParam.ioBuffer = (Ptr)data;
    pb.ioParam.ioReqCount = length;
    pb.ioParam.ioPosMode = fsFromStart;
    pb.ioParam.ioPosOffset = address;
    OSErr err = PBReadSync(&pb);
    if(err && err != eofErr) {
        IAAssertion(false, "unable to read", StoreError);
    }
    return pb.ioParam.ioActCount;
}

```

Creating a Subclass of IAMutex

If your application may be run in a multi-threaded environment, you will need to create your own operative subclass of IAMutex. This will allow the IAT code to prevent concurrent access when it would harm the integrity of storage. The resulting mutex may also be used by the application code.

Required Functions

- Lock
- Unlock

Implementing IANewMutex

IANewMutex is a variable used to by IAT construct a new mutex. The default implementation defines IANewMutex as follows:

```
IAMutexConstructor*IANewMutex = &IADefaultMutexConstructor;
```

where IADefaultMutexConstructor returns a pointer to a mutex with no-op implementations of Lock() and Unlock(). (These variables are declared as shown in Listing 9-17.) This default will work for single-threaded applications. Applications that are creating a working subclass of IAMutex must reset this variable to their own mutex class.

Listing 9-17 Current implementation of IAMutex

```
typedef IAMutex*IAMutexConstructor();
IAMutex*          IADefaultMutexConstructor();// no-op
extern IAMutexConstructor* IANewMutex;
```

Storage Class Category Reference

Header Files in the Storage Class Category

HFSStorage.h

MakeHFSStorage (utility)

HFSStoreStream.h

HFSStoreStream

IAMutex.h

IALock

IAMutex

IStorage.h

IAInputBlock

IAOutputBlock

IStorage

IStoreStream.h

IStoreStream

Class Specifications

Class HFSSoreStream

Header: HFSSoreStream.h

Hierarchy

Public subclass of IASoreStream. See “IASoreStream” on page 9-39.

Description

HFSSoreStream provides the I/O capabilities for HFSSStorage. When MakeHFSSStorage (see “MakeHFSSStorage” on page 9-45) creates storage it creates an HFSSoreStream.

Public Member Functions

constructor

Input

```
short vRefNum
    The volume reference number of the storage to be accessed.
long dirID
    Its directory ID.
const StringPtr filename
    The HFS filename of the storage.
OSType creator = 'VTWN'
    Who created the stream.
OSType fileType = 'STOR'
    The type of store stream.
```

destructor

Clone

See “IAStoreStream.Clone” on page 9-40.

Flush

See “IAStoreStream.GetMutex” on page 9-40.

GetCreator

Access method for HFSSStoreStream member data.

Output

OSType	creator
	The creator of the store stream.

GetDirID

Access method for HFSSStoreStream member data.

Output

const long	dirID
	The HFS directory ID of the storage to access.

GetEOF

See “IAStoreStream.GetEOF” on page 9-41. Returns HFS EOF position for file.

GetFileName

Access method for HFSSStoreStream member data.

Output

StringPtr fileName
The HFS filename of the storage to access.

GetFileType

Access method for HFSSStoreStream member data.

Output

OSType fileType
The file type of the storage.

GetFRefNum

Access method for HFSSStoreStream member data.

Output

short fRefNum
The HFS file reference number, a handle on the open file.

GetVRefNum

Access method for HFSSStoreStream member data.

Output

const short tvRefNum
The HFS volume reference number.

Initialize

See “IAStoreStream.Initialize” on page 9-42.

IsOpen

See “IAStoreStream.IsOpen” on page 9-40.

IsWritable

See “IAStoreStream.IsWritable” on page 9-41.

Open

See “IAStoreStream.Open” on page 9-42.

SetEOF

See “IAStoreStream.Protected Member Functions” on page 9-41.

SetFRefNum

Access method for HFSSStoreStream member data.

Input

short	fRefNum
	The HFS file reference number, a handle on the open file.

Protected Member Functions

constructor

Input

short	vRef
	HFS volume reference number.
long	dirId
	HFS directory ID.

Storage Category

```
const StringPtr filename
    HFS filename.
OSType creator
    The creator of the stream.
OSType fileType
    The type of the stream.
bool isOpen
    Whether the stream is open(true) or not (false).
bool isWritable
    Whether the stream is open for output (true) or read only(false).
short fRefNum
    The HFS file reference number.
```

Notes

Constructor for use by Clone().

Read

See “IAStoreStream.Read” on page 9-43.

Write

See “IAStoreStream.Protected Member Functions” on page 9-41.

Class **IAInputBlock**

Header: IASStorage.h

Hierarchy

Base Class.

Description

An input block is the logical container of storage. It serves as an interface between the storage and the store stream.

Relationships

IAInputBlock reads IASToreStream

One input block reads from one and only one store stream. This store stream may be a clone of the one created with the storage.

IAInputBlock addresses IASStorage

An IAInputBlock addresses part of one storage.

Client

See “IAStorable restores from IAInputBlock” on page 10-27.

Public Member Functions

constructor

Input

IASStorage* storage
The storage which has this block.

IABlockID id
The identification number of the block.

IAStoreStream* stream = NULL
A request for a cloned store stream.

Notes

Locks stream's mutex and positions stream at address for read.
A cloned IASToreStream can be supplied to improved threaded throughput.

Usage

```
IAInputBlock input(storage, id, stream);
```

destructor

Unlocks stream's mutex

GetPosition

Output

`uint32 position`
The current position in the input store stream.

Usage

```
uint32 start = input.GetPosition();
```

ReadBuffer

Input

`void* buffer`
Pointer to the buffer to be filled.
`uint32 length`
Number of bytes to place in the buffer.

Usage

```
input->ReadBuffer(newText, length);
```

ReadByte

Output

`byte`

Usage

```
byte length = input->ReadByte();
```

ReadUInt32

Output

uint32

The uint32 read.

Usage

```
long  newCreationDate = input->ReadUInt32();
```

IAOutputBlock

Class

Header: IASTorage.h

Hierarchy

Base Class.

Description

IAOutputBlock connects a logical block with a store stream and position within storage. It is used to write storage to disk.

Relationships

IAOutputBlock writes to IASToreStream

One block writes to one and only one store stream.

Clients

See “IAStorage creates IAOutputBlock by ID” on page 9-31.

See “IAStorable stores in IAOutputBlock” on page 10-27.

Public Member Functions

constructor

Input

IStorage* storage
The storage in which the block will lie.

IABlockID id
The identification number of the block.

IABlockSize storeSize
The size of the block.

IStoreStream* stream = NULL
A cloned store stream; used only to improve throughput. If NULL, the block will write to the storeStream contained in the storage.

Notes

Allocates block on the stack.

Locks stream's mutex and positions stream at address for write.

A cloned IStoreStream can be supplied to improved threaded throughput.

Usage

IAOutputBlock output(storage, id, storeSize, stream)

destructor

Flushes changes and unlocks stream's mutex

GetPosition

Output

uint32 position
The current position in the stream.

Storage Category

Usage

```
IABlockAddress position = output.GetPosition();
```

WriteBuffer

Input

```
void* buffer
    Pointer to the buffer to be written.
uint32 length
    Number of bytes to write.
```

Usage

```
output.WriteBuffer(&buffer, sizeof(buffer));
```

WriteByte

Input

```
byte b
    Byte to be written
```

Usage

```
output->WriteByte(fileName[0]);
```

WriteUInt32

Input

```
uint32 i
    The uint32 to write.
```

Usage

```
output->WriteUInt32(Count());
```

Class IALock

Header: IAMutex.h

Locks a mutex for the duration of its stack-allocated life.

Description

IALock is a semaphore that, when constructed, prevents access to a store stream by threads other than that of its creator.

See “IAMutex” on page 9-28 for more information.

Public Member Functions**constructor****Input**

IAMutex* mutex

Notes

Locks the mutex. Run before code requiring a lock.

Usage

```
IALock lock(mutex);
```

destructor**Notes**

Unlocks the mutex.

Class IAMutex

Header: IAMutex.h

Hierarchy

Base Class

Description

Interface to mutexes (MUtual EXclusion semaphores) used by IA library.

There is no explicit constructor for IAMutex. The body establishes a no-op default mutex by the automatic creation of IANewMutex.

Applications must subclass IAMutex and set IANewMutex to a real semaphore to make IA code thread-safe for the application's threads.

Relationships

IAMutex is locked by IALock

One mutex may be locked by one lock.

Public Member Functions

constructor

Notes

No explicit constructor. Defining causes a function to run as part of a typedef.

Usage

```
IAMutex *mutex;

mutex(IANewMutex()) // part of constructor initialization
```

destructor

Virtual.

No-op.

IADefaultMutexConstructor

Output

IAMutex*

Notes

Default, no-op mutex constructor. IAMutexConstructor is a typedef for IAMutex*. (This is not an actual IAMutex member function.)

Usage

```
IAMutexConstructor* IANewMutex = &IADefaultMutexConstructor;
```

Lock

Pure virtual.
Returns when we have control of the mutex.

Unlock

Pure virtual.
Releases control of the mutex. Not invoked directly; invoke through the destruction of IALock.

Class IASStorage

Header: IASStorage.h

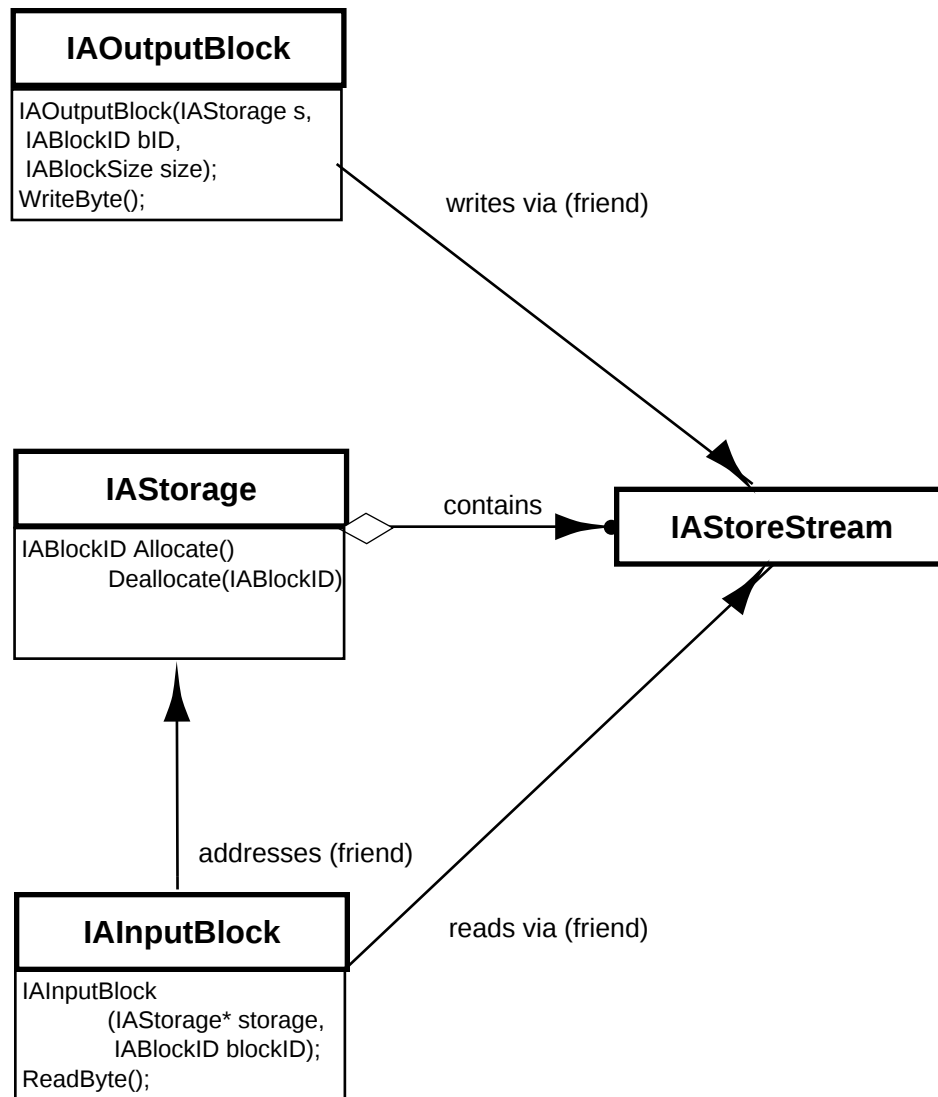
Hierarchy

Abstract Base Class.

Description

This abstract class provides for storage in persistent memory. Storage is done in logical blocks without knowledge of client data structures.

Relationships

Figure 9-3 IASStorage relationships**IASStorage creates IASOutputBlock by ID**

Blocks are allocated by ID. They are then constructed as input or output. One storage may create many blocks.

IStorage creates IStoreStream

One storage has one contained stream, but may have many clones.

Client

See “IAInputBlock addresses IStorage” on page 9-23.

Public Member Functions

constructor

Input

IStoreStream* s

uint32 t

Notes

Notes storeStream and type, creates mutex. Called through a utility. See “IAMakeStorage” on page 9-45.

destructor

Deletes storeStream and mutex

Allocate

Pure virtual.

Output

IABlockID id

The identification number of the new block.

Storage Category

Notes

Allocates a new block ID. Block is actually created with `IAOutputBlock` is constructed. See “`IAOutputBlock`” on page 9-25.

Usage

```
IABlockID id = storage->Allocate();
```

AllocateNamedBlock

Input

`char* name`
The name to be assigned to the block.

Output

`IABlockID id`
The identification number of the new block.

Usage

```
indexRoot = storage->AllocateNamedBlock(IADefaultIndexName);
```

Commit

Pure virtual.
Makes permanent any changes since open.

Usage

```
storage->Commit();
```

Compact

Pure virtual.
Attempts to compact the storage.

Storage Category

Usage

```
storage->Compact();
```

Deallocate

Pure virtual.

Input

IABlockID id

The identification number of block to delete.

Notes

Frees a previously allocated block. Does not remove the TOC entry in the case of named blocks.

▲ **WARNING**

You should use `RemoveNamedBlocks` if you have a named block to deallocate. If you use `Deallocate` on a named block without simultaneously calling `TOC_Remove` on the name, you will render that name unusable for the remaining life of the storage.

Usage

```
storage->Deallocate(id);
```

FreeSpace

Pure virtual.

Output

IABlockSize

The number of bytes of free space.

Usage

```
IABlockSize free = storage->FreeSpace();
```

GetMutex

Access method for IASStorage member data.

Output

`IAMutex*` `mutex`
The mutex established to enable locking. See “IAMutex” on page 9-28.

GetNamedBlock

Input

`const char*` `name`
the string used as a label for the block

Output

`IABlockID`
the ID of the block. Will allocate a new block if name not found in the TOC.

GetStorageType

Access method for IASStorage member data.

Output

`const uint32storageType`
The type of storage.

GetStoreStream

Access method for IASStorage member data.

Output

`IASStoreStream*` `storeStream`
The store stream created to access the storage.

Initialize

Pure virtual.

Initializes a new storage, or empties an existing one. The storage is left open afterwards.

Usage

```
storage->Initialize();
```

IsOpen

Output

bool

True: the storage is open. False: the storage is not open.

IsWritable

Output

bool

True: the storage is open with permission to write.
False: the storage is not open or open as read-only.

Open

Pure virtual.

Input

```
bool writable = false
```

Defaults to read only (false). True is write-permitted.

Notes

Opens the storage (and its storeStream), enabling subsequent operations.
If "writable" is true, destructive operations are supported.

Storage Category

Usage

```
storage->Open(true);
```

RemoveNamedBlock

Input

```
const char* label
```

The name of the named block to remove.

Output

```
bool
```

True if the named block is removed; false if no block by that name exists.

Notes

Frees a previously allocated named block, and deletes the TOC entry for it.

Usage

```
storage->RemoveNamedBlock("my block name");
```

TOC_Get

Pure virtual.

Input

```
char* label
```

The name assigned to the block.

Output

```
IABlockID
```

The identification number of the block.

Usage

```
indexRoot = storage->TOC_Get(IADefaultIndexName);
```

TOC_Remove

Pure virtual.

Input

`const char* label`

The name of the block to be removed from the TOC.

Output

`bool`

True if the block name was successfully removed; false otherwise.

Notes

Removes the entry in the storage TOC that maps the given name to a blockID. Does not deallocate the block itself from storage.

Normally, you will want to deallocate the block at the same time you remove the TOC entry. In that case, you should use `RemoveNamedBlock`, which does both.

Usage

```
indexRoot = storage->TOC_Remove("My Block Name");
```

TOC_Set

Input

`const char* label`

The name to be assigned to the block.

`IABlockID id`

The identification number of the block.

Notes

See also “`AllocateNamedBlock`” on page 9-33. This will replace the entry if found; that is, this function may be used to change the ID for a named block. If the entry is not found, the name and blockID are added to the TOC.

Usage

```
storage->TOC_Set("my block name", block);
```

TotalSpace

Pure virtual.

Output

`IABlockSize`

The number of bytes occuited by storage.

Usage

```
IABlockSize total = storage->TotalSpace();
```

Protected Member Functions

IAStoreStream

Class

Header: `IAStoreStream.h`

Hierarchy

Abstract base class.

Description

For implementing `IStorage` on different file systems.
Implementations need only implement pure virtual members.
Clients should not use `IAStoreStream` member functions directly, but rather use through an `IStorage`.

Clients

See “`IAInputBlock` reads `IAStoreStream`” on page 9-23.

See “`IAOutputBlock` writes to `IAStoreStream`” on page 9-25.

See “`IStorage` creates `IAStoreStream`” on page 9-32.

Public Member Functions

constructor

destructor

Virtual.

Clone

Pure virtual.
Returns a new storeStream read/writing the same store.

Usage

```
storage->storeStream->Clone()
```

GetMutex

Access method for IASToreStream member data.

Output

IAMutex* mutex
The mutex for the store stream.

IsOpen

Pure virtual. An implementation of this function should lock the mutex while executing.

Output

bool
True if the store stream is open, false if not. Returns the value of isOpen.

Storage Category

Usage

```
IAAssertion(IsOpen(), "Storage not open!", StorageNotOpen)
```

IsWritable

Pure virtual.

Output

bool

Returns the value of isWritable; true if the storage is open and writable, false if not open or open for read only.

Notes

An implementation of this function should lock the mutex while executing.

Protected Member Functions

Flush

Pure virtual.

Flushes buffered output to disk. An implementation of this function should lock the mutex while executing.

Usage

```
storeStream->Flush()
```

GetEOF

Pure virtual.

Output

uint32

The current EOF; one greater than last position currently occupied.

Storage Category

Notes

Returns one greater than the last position currently occupied. An implementation of this function should lock the mutex while executing.

Usage

```
storeStream->GetEOF()
```

Initialize

Pure virtual.

Creates a new store on disk and sets up the initial block tables. An implementation of this function should lock the mutex while executing. Does not open the store stream.

Usage

```
storeStream->Initialize();  
storeStream->Open(true);
```

MaybeFlushBuffer

Write buffer if it's dirty & mark it clean.

Open

Pure virtual.

Input

```
bool writable
```

True if open for output, false if open for read only.

Notes

Opens an existing store, enabling changes when “writable” is true. An implementation of this function should lock the mutex while executing.

Storage Category

Usage

```
storeStream->Open(true)
```

Read

Pure virtual.

Input

`uint32 fromPos`

The position in the storage to read.

`byte* buffer`

A pointer to the buffer; read data returned here.

`uint32 bytesWanted`

The number of bytes to read.

Output

`uint32 bytesActual`

The number of bytes actually read.

Notes

Mutex should be already locked.

Usage

```
bytesActual = storeStream->Read(fromPos, buffer, bytesWanted);
```

SetEOF

Pure virtual.

Input

`uint32 address`

The new position to become the end of file.

Notes

Truncates or extends the storage to the requested length. An implementation of this function should lock the mutex while executing.

Storage Category

Usage

```
if (newEOF < oldEOF) {
    storeStream->SetEOF(newEOF);
}
```

Write

Pure virtual.

Input

```
uint32 toPos
    The position in the stream to begin to write to.
byte* buffer
    The pointer to the buffer containing the data
uint32 bytes
    The number of bytes to write.
```

Notes

Mutex should be already locked.

Usage

```
storeStream->Write(toPos, buffer, bytes);
```

Storage Class Utilities

IAMakeStorage

Header: IASStorage.h

Input

IAStoreStream* storeStream
the store stream for the file type to be stored

Output

IAStorage*
a pointer to the new logical storage object for the storage

Notes

This is the prototype of a basic utility to construct storage. It should be used instead of a constructor for IASStorage.

MakeHFSSStorage

Header: HFSSStorage.h

Input

short vRef
The HFS volume reference number of the volume where the file is or is to be located.

long dirId
The directory ID of the directory where the file is located.

const StringPtr name
The name of the file.

OSType creator = 'VTWN'
The creator of the storage.

OSType fileType = 'STOR'
The type of file.

Storage Category

Output

IAStorage*
A pointer to the storage.

Notes

This is the “constructor” for HFSSStorage. It constructs an IAStorage as a Macintosh HFS file. All other operations on HFSSStorage will be done as a function of IAStorage. There is no true subclass named HFSSStorage.

Usage

```
IAStorage * exStorage =
    MakeHFSSStorage(vRefNum, dirID, exStorName)
```

VInt32Read

Header: VInt32.h

Input

IAInputBlock* input
The input block positioned for the read.

Output

int
The next VInt32.

Notes

A variable length decoding of a uint32.

Usage

```
vRefID = VInt32Read(input);
```

VInt32Size

Header: VInt32.h

Storage Category

Input

uint32 i
The item to be sized.

Output

byte
The actual size of the item when enclosed in VInt32 format.

Usage

```
IABlockSize HFSDoc::StoreSize() {
    return VInt32Size(vRefID) + VInt32Size(dirID)
        + 1 + fileName[0];
}
```

VInt32Write(uint32 i, IAOutputBlock* output)

Header: VInt32.h

Input

uint32 i
The item to be written.
IAOutputBlock* output
The block to write it to, positioned for the write.

Notes

A variable length encoding of a uint32.

Usage

```
VInt32Write(vRefID, output);
```

Typedefs

IABlockAddress

The storage address of the first byte of a logical block.

Type

uint32

Header

IStorage.h

IABlockID

A unique logical identifier for a block of storage.

Type

uint32

Header

IStorage.h

IABlockSize

The number of bytes allocated to a block.

Type

uint32

Header

IStorage.h

IAMutexConstructor();

Type

IAMutex*

Header

IAMutex.h

Storage Exceptions and Error Handling

Errors are currently handled by throwing exceptions.

VSAO

StorageAlreadyOpen.

You have tried to reopen storage that already is open. You may have tried an initialize.

Class

IStorage

VSBI

StorageBlockIDInvalid.

The block ID is not found in the table of contents of this storage.

Class

IStorage

VSDF

StorageFull.

The disk is full.

Class

IStorage

VSEr

StoreError.

Class

IStoreStream

Storage Category

VSEo

StorePastEOF.

Class

IAStoreStream

VSIV

StorageInvalid

Cannot make or open this type of storage.

Class

IAStorage

VSNI

StorageNotInitialized

You have tried to access storage that has been created, but not initialized using the Initialize() command.

Class

IAStorage

VSNO

StorageNotOpen

You have tried to access storage that has been established, but not opened using the Open() command.

Class

IAStorage

Storage Category

VSNW

StorageNotWritable

You are trying to change storage that was opened as read-only. This may be because of a deallocate, allocate, write, commit, or compact command.

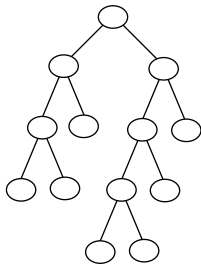
Class

IAStorage

VSPB

StorePastBlockEnd**Class**

IAStoreStream



Storable Category

Understanding Storables and Ordered Storables	3
Creating Subclasses	4
Creating a Subclass of IASortable	4
Creating a Subclass of IAOrderedSortable	6
Creating a subclass of IAOrderedSortableSet	7
Common Operations	7
Creating an Ordered Storable Set	7
Open an Existing Ordered Storable Set	8
Updating an Existing Ordered Storable Set	8
Sample Code for Updating an Ordered Storable Set	10
Searching and Iterating through an Ordered Storable Set	11
Storable Class Category Reference	13
Header File	13
Class Specifications	14
IAOrderedSortable	14
IAOrderedSortableIterator	17
IAOrderedSortableSet	18
IASortable	27
Class Utilities	32
Externs	33
Exceptions and Error Handling	33

Storable Category

Storable Category

IStorable classes have been created to allow easy organization and storage of objects within the IAT. You won't have to be aware of the storable classes to use the IAT for information access, but you may wish to use the classes for other object-oriented storage. These classes differ from other existing implementations of structures, as they support very large storable sets of variable-length objects. The sets are paged from disk.

This chapter describes the storable logic and requirements for re-use.

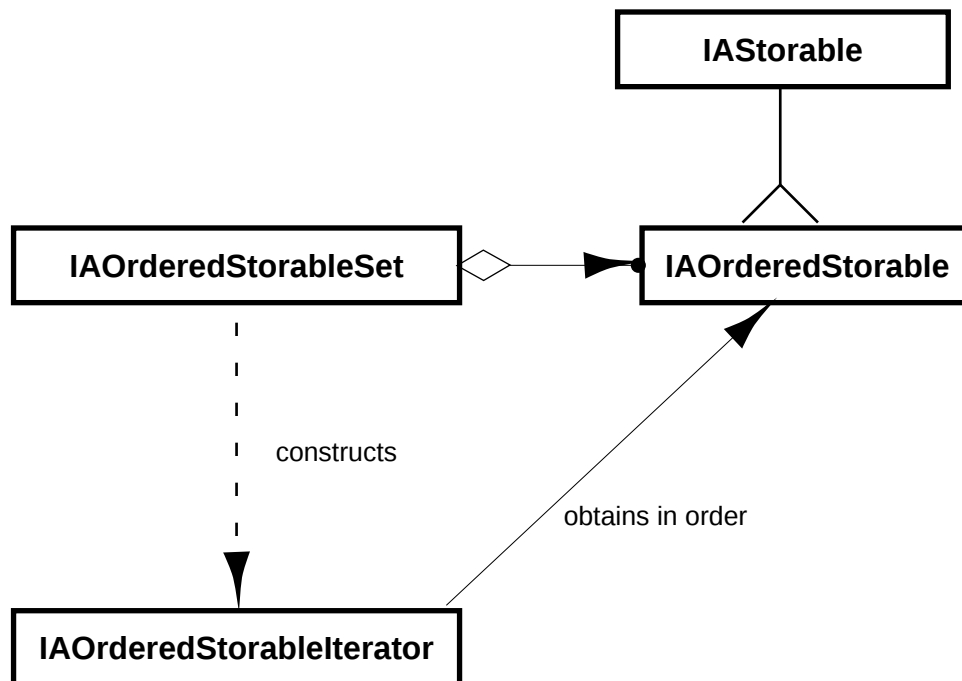
Understanding Storables and Ordered Storables

❄️
❄️

A **storable** (IStorable) is any object with member data that should persist beyond program execution. An **ordered storable** (IAOrderedStorable) is a storable object with a unique identifier, or key. This identifier is a piece of member data whose value is unique for any one occurrence of an object. This uniqueness allows sorts, equal, and less than operations. These permit the use of a set (IAOrderedStorableSet) and an iterator (IAOrderedStorableIterator) that allows access in sequential order.

Figure 10-1 shows the relationships between the storable classes.

Figure 10-1 Object storage structures



If it is possible to have a unique identifier, you should implement objects to be stored as subclasses of IAOrderedStorable.

Creating Subclasses

Creating a Subclass of IASTorable

An IASTorable is an object which may be stored in the logical input and output blocks of IAT Storage.

See “IAStorable” on page 10-27 for detailed information.

Required Functions

- DeepCopy
- Store
- Restore
- StoreSize

Listing 10-1 Sample header file for an IASTorable subclass

```
class HFSVolumeInfo : public IASTorable {
public:
    HFSVolumeInfo() : name(NULL) {}
    HFSVolumeInfo(short vRefNum);
    ~HFSVolumeInfo();

    // methods to store a HFSVolumeInfo
    IABlockSize StoreSize() const;
    void Store(IAOutputBlock* output) const;
    IASTorable* Restore(IAInputBlock* input) const;
    IASTorable* DeepCopy() const;

    short GetVolumeRefNum() const {return vRefNum;}
    StringPtr GetVolumeName() const {return name;}
    long GetCreationDate() const {return creationDate;}

    void SetVolumeRefNum(short refNum) {vRefNum = refNum;}
    void SetVolumeName(StringPtr vname) {name = vname;}
    void SetCreationDate(long cDate) {creationDate = cDate;}

private:
    HFSVolumeInfo(short v, StringPtr n, long c) :
```

Storable Category

```

        vRefNum(v), name(n), creationDate(c) {}
short    FindVRefNum(const StringPtr name, long creationDate) const;

        HFSVolumeInfo(HFSVolumeInfo&);// don't define a copy constructor

short    vRefNum;        // volume reference number (ephemeral)
StringPtr name;          // volume name (persistent)
long     creationDate;   // volume creation date (persistent)
};

```

Listing 10-2 Sample Constructor

```

HFSVolumeInfo::HFSVolumeInfo(short vrn) {
    Str255 nameBuffer;
    ParamBlockRec pb;
    pb.volumeParam.ioNamePtr = nameBuffer;// set up pb
    pb.volumeParam.ioVRefNum = vrn;
    pb.volumeParam.ioVolIndex = 0;
    OSErr err = PBGetVInfo(&pb, false);// get info
    IAAssertion(!err, "cannot get volume info!", HFSVolumeNotFound);
    vRefNum = pb.volumeParam.ioVRefNum;
    name = IAMallocArray(byte, pb.volumeParam.ioNamePtr[0] + 1);
    pstrcpy(name, pb.volumeParam.ioNamePtr);
    creationDate = pb.volumeParam.ioVCrDate;
}

```

Listing 10-3 Sample Implementation of DeepCopy

```

IAStorable*HFSVolumeInfo::DeepCopy() const {
    byte* newName = IAMallocArray(byte, name[0] + 1);
    pstrcpy(newName, name);
    return new HFSVolumeInfo(vRefNum, newName, creationDate);
}

```

Listing 10-4 Sample Implementation of Restore

```

IAStorable* HFSVolumeInfo::Restore(IAInputBlock* input) const {
    // read name
    byte length = input->ReadByte();
    byte* newName = IAMallocArray(byte, length + 1);

```

Storable Category

```

newName[0] = length;
input->ReadBuffer(newName + 1, length);

long newCreationDate = input->ReadUInt32();// read creationDate
short newVRefNum = FindVRefNum(newName, newCreationDate);// find vRefNum

if (newVRefNum == 0)
    return NULL;
else
    return new HFSVolumeInfo(newVRefNum, newName, newCreationDate);
}

```

Listing 10-5 Sample Implementation of StoreSize

```

IABlockSizeHFSVolumeInfo::StoreSize() const {
    return 1 + name[0] + sizeof(uint32);
}

```

Listing 10-6 Sample Implementation of Store

```

void HFSVolumeInfo::Store(IAOutputBlock* output) const {
    output->WriteByte(name[0]);
    output->WriteBuffer(name + 1, name[0]);
    output->WriteUInt32(creationDate);
}

```

Creating a Subclass of IAOrderedStorable

An IAOrderedStorableSubclass is the same as a storable subclass (see “Creating a Subclass of IASTorable” on page 10-4) with the addition of functions for Equal and Less Than.

See “IAOrderedStorable” on page 10-14 for more information.

Required Functions

- DeepCopy
- Store
- Restore
- StoreSize
- Equal

Storable Category

■ Less Than

Listing 10-7 Sample Implementation of Equal

```
bool OrderedStorableSubClass::Equal(IAOrderedStorable* neighbor) {
    int comparison = strcmp(name, neighbor->name);
    return (comparison == 0);
}
```

Listing 10-8 Sample Implementation of Less Than

```
bool OrderedStorableSubClass::LessThan(IAOrderedStorable* neighbor) {
    int comparison = strcmp(name, neighbor->name);
    return (comparison < 0);
}
```

Creating a subclass of IAOrderedStorableSet

You don't have to create a subclass of the IAOrderedStorableSet or the IAOrderedStorableIterator. The subclasses provided will work on any subclass of IAOrderedStorable. The application can create instances of these classes, then cast as required for the specific storable subclasses used.

Common Operations

Creating an Ordered Storable Set

The ordered storable set is the data structure that points to the members of the set and provides the iterator to allow access to them. Ordered storable sets are used to store large collections of persistent data.

You must have storage open for write access and an output block in the storage to establish an IAOrderedStorableSet. See Chapter 9, "Storage Category" for more information on establishing storage and allocating blocks. Generally you will want to allocate a named block for storable set so it may be easily reestablished from storage.

Sets are constructed using the utility IAMakedOrderedStorableSet, which takes a prototype of the OrderedStorable as input.

Listing 10-9 Creating an IAOrderedStorableSet

```
// construct an ordered storable set
IAOrderedStorableSet* anOSSet = IAMakeOrderedStorableSet
                                (new OrderedStorableSubclass());
// allocate a block for storing the set
IABlockID setBlockID = anIAStorage->AllocateNamedBlock(aBlockName);
// initialize the set
anOSSet->Initialize(anIAStorage, setBlockID);
```

Open an Existing Ordered Storable Set

An ordered storable set is restored from disk by restoring the storage, locating the blockID, and creating and opening the set.

The example assumes storage was created with a named output block.

Listing 10-10 Open an existing Ordered Storable Set

```
// open storage (See "Opening Existing Storage" on page 9-6)
// open ordered storable set
IAOrderedStorableSet* anOSSet = IAMakeOrderedStorableSet
                                (new OrderedStorableSubclass());
IABlockID setBlockID = anIAStorage->TOC_Get(aBlockName);
anOSSet->Open(anIAStorage, setBlockID, writable);
```

Updating an Existing Ordered Storable Set

IAOrderedStorableSet contains member functions to allow the set to be updated.

See "IAOrderedStorableSet" on page 10-18 for detailed information on each of these functions.

The Put(anIAOrderedStorable) function adds or replaces a member of the set. If a storable exists that is equal to the supplied input (that is, it has the same key data), the storable will be replaced with the new storable.

If the storable supplied with Put does not exist in the set, it will be added to the set.

▲ **WARNING**

Applications should validate supplied input to be certain no unwanted addition occurs because of an erroneous key.

The Get(anIAOrderedStorable) will retrieve any storable in the set with a matching key. The supplied input storable must have the key data (that used for the equal member function) in place. The retrieved storable will replace the input storable.

The Remove(anIAOrderedStorable) will locate and delete any storable with matching key data from the set.

Storable Category

If you wish to change the key data of a storable in the set, the storable with the existing key must first be removed. Then the storable with the new key may be added with the `Put()` function.

Old data is not overwritten in storage during the update. This allows the data to remain consistent if there is a failure. To replace the persistent ordered storable set following any updates, `Flush()` the set to place changes in storage, then `Commit()` the storage to make the changes persistent.

You can use an iterator during the updates; the results of the update are reflected in the iterator behavior.

Sample Code for Updating an Ordered Storable Set

Perhaps Chef Irina requires a list of her customers by name, with additional data such as number of recipes submitted. This could be kept as an ordered storable set. The following examples assume a data member “name” which is the key data for the OrderedStorableSubclass.

Listing 10-11 Adding a storable to an OrderedStorableSet

```
// add a storable
char* addName = "Liam";
OrderedStorableSubclass newOrderedStorable
    ((byte*)addName, strlen(addName));
bool exists = anOSSet->Get(&newOrderedStorable);
if (exists) {
    printf ("%s is already there; will not add\n",
            newOrderedStorable.name);
} else {
    anOSSet->Put(&newOrderedStorable);
    printf ("%s is added \n", newOrderedStorable.name);
}
```

Listing 10-12 Updating additional data for an existing storable

```
// change non-key data in a storable
char * existingName = "Liam";
char* newData = "updated";
OrderedStorableSubclass anOrderedStorable
    ((byte*)existingName, strlen(existingName));
bool exists=anOSSet->Get(anOrderedStorable);
if (!exists) {
    printf ("%s is not there; cannot change\n", anOrderedStorable.name);
} else {
    anOrderedStorable.data=newData;
    anOSSet->Put(&anOrderedStorable);
    printf ("%s is replaced \n", anOrderedStorable.name);
}
```

Listing 10-13 Removing a storable from an OrderedStorableSet

```
// remove storable
char * existingName = "Liam";
OrderedStorableSubclass anOrderedStorable
    ((byte*)existingName, strlen(existingName));
bool existed = anOSSet->Remove(anOrderedStorable);
if (!existed) {
    printf ("%s was not there; cannot remove\n", anOrderedStorable.name);
} else {
    printf ("%s has been removed \n", anOrderedStorable.name);
}
```

Searching and Iterating through an Ordered Storable Set

There are several means of reading the contents of an object stored in an Ordered Storable Set:

- getting the object by its key using the Get member function
- making an iterator and searching the set sequentially
- making an iterator positioned at the object

In the above set of customers, you could use the Get function to find a specific customer's data.

Use the sequential iterator to list all the customers.

Use a positioned iterator to locate a certain point in the list (such as the letter "L") and list from that point on.

If you have a large number of items to look up in an ordered storable set, it may be faster to iterate through the entire set than to do a series of lookups using Get.

Listing 10-14 Get an object by key

```
char * existingName = "Liam";
OrderedStorableSubclass anOrderedStorable((byte*)existingName,
                                           strlen(existingName));
bool exists = anOSSet->Get(anOrderedStorable);
if (!exists) {
    printf ("%s is not there; \n", anOrderedStorable.name);
}
```

Listing 10-15 Make a sequential iterator

```
// iterate through the entire set (list it)
uint32 numberStorables = anOSSet->Count();
printf ("%lu Number of storables\n", numberStorables);
IAOrderedStorableIterator*
    anOSIter = anOSSet->MakeIterator();
OrderedStorableSubclass* anOS;
while(anOS = (OrderedStorableSubclass*)anOSIter->Next());
    printf ("%s\n", (char*)anOS->name);
}
```

Listing 10-16 Make a positioned iterator

```
// Iterate from a given point
char* startingPoint = "L";
OrderedStorableSubclass pointOS((byte*)startingPoint,
                                strlen(startingPoint));
IAOrderedStorableIterator* anOSIter = anOSSet->MakeIterator(&pointOS);
OrderedStorableSubclass* anOS =
    (OrderedStorableSubclass*)anOSIter->Next();
if (!(anOS->Equal(&pointOS))) {
    printf("%s isn't in the set\n", startingPoint);
}
while(anOS = (OrderedStorableSubclass*)anOSIter->Next());
    printf ("%s\n", (char*)anOS->name);
}
```

Storable Class Category Reference

Header File

IAStorable.h

IAMakeOrderedStorableSet (utility)

IAOrderedStorable

IAOrderedStorableIterator

IAOrderedStorableSet

IAStorable

Class Specifications

Class `IAOrderedStorable`

Header: `IAStorable.h`

Hierarchy

Abstract Base Class.

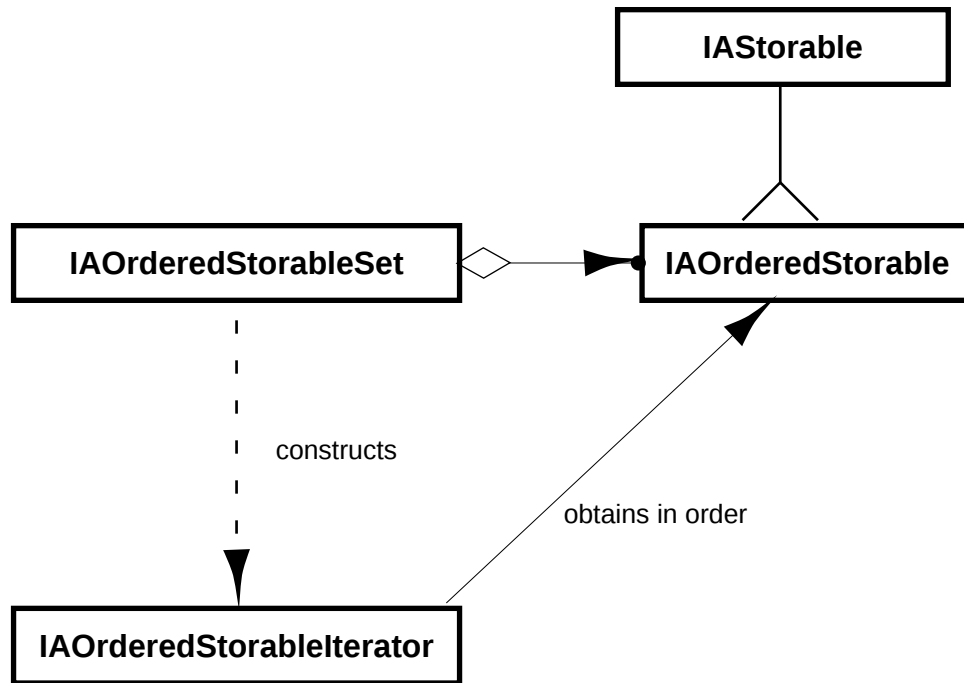
Superclass: `IAStorable`. See “`IAStorable`” beginning on page 10-27.

Description

An `IAOrderedStorable` object is something which is meant to be stored as part of an ordered set of persistent objects. Ordered storables are the same as storables except they have a unique key.

Relationships

Figure 10-2 IAOrderedStorable relationships



Clients

See “IAOrderedStorableIterator obtains (in order) IAOrderedStorable” on page 10-17.
 See “IAOrderedStorableSet contains IAOrderedStorable” on page 10-19.

Public Member Functions

Equal

Pure Virtual.

Storable Category

Input

IAOrderedStorable* neighbor

The item to be tested for equality to this object.

Output

bool

The result of the test (true if the keys of the items are equal, false if they are not.)

Notes

Equal returns true if this object is equal to the input object. The operation is performed on the member data which make up the key to the ordered storable. Put and Get use this function to allow access to an ordered storable by key. See “IAOrderedStorableSet” on page 10-18 for more information on retrieval and update by key.

Listing 10-17 Sample Implementation of Equal

```
bool OrderedStorableSubClass::Equal(IAOrderedStorable*
                                   neighbor) {
    int comparison = strcmp(name, neighbor->name);
    return (comparison == 0);
}
```

LessThan

Pure Virtual.

Input

IAOrderedStorable* neighbor

The item to be tested to see this object's key is less than the input object's key.

Output

bool

The result of the test (true if the key of this object is less than the Input object, false if it is not.)

Storable Category

Notes

LessThan returns true if this object is less than the input object, neighbor. The operation is performed on the member data which make up the key to the ordered storable.

Listing 10-18 Sample Implementation of LessThan

```
bool  OrderedStorableSubClass::LessThan(IAOrderedStorable*
                                         neighbor) {
    int comparison = strcmp(name, neighbor->name);
    return (comparison < 0);
}
```

IAOrderedStorableIterator**Class**

Header: IASTorable.h
Hierarchy

Base Class.
Description

The iterator returns members of an IAOrderedStorableSet in sequence of their keys.
Relationships**IAOrderedStorableIterator obtains (in order) IAOrderedStorable**

One iterator may obtain many storables.
Client

See “IAOrderedStorableSet constructs IAOrderedStorableIterator” on page 10-19.

Public Member Functions

constructor

There is no constructor for this class. Iterators should always be constructed with the `MakeIterator` functions of `IAOrderedStorableSet`. See “`MakeIterator()`” beginning on page 10-22.

Next

Pure Virtual.

Output

`IAOrderedStorable* key`

A pointer to a copy of the next object (sequentially) or NULL if at the end of the set.

Notes

This returns a deep copy of the next sequential (in terms of the key value) `IAOrderedStorable`. If invoked after the end of the set, it will return NULL.

Deep copies must be explicitly deleted by the client.

Class `IAOrderedStorableSet`

Header: `IStorable.h`

Hierarchy

Superclass: none.

This is an abstract base class; however, there is an internally implemented subclass that is used in all cases.

Description

An `IAOrderedStorableSet` is a collection of `IAOrderedStorable` objects kept in sequential order. Currently this set is implemented as a variant of a B-tree. `IAOrderedStorableSets` are kept in storage objects.

Relationships

IAOrderedStorableSet contains IAOrderedStorable

One set stores many storables.

IAOrderedStorableSet constructs IAOrderedStorableIterator

One set may construct many iterators.

Client

IAMakeOrderedStorableSet constructs IAOrderedStorableSet

IAMakeOrderedStorableSet is a class utility used to construct a set. There is no persistent relationship.

Public Member Functions

constructor

Do not use the constructor directly. Rather, use the IAMakeOrderedStorableSet utility found in this header. See “IAMakeOrderedStorableSet” on page 10-32 for more information.

Count

Pure Virtual.

Output

uint32

The number of objects in the set.

Storable Category

Notes

Count returns the number of objects in the set. The set must be initialized or open.

Usage

```
uint32 numKeys=KeyNameSet->Count();
```

Destroy

Pure Virtual.

Notes

Frees all storage blocks associated with the set.

Flush

Pure Virtual.

Output

void

Changes the storage to reflect changes in the set, but returns nothing.

Notes

Changes made to a set are cached. Flush empties the cache and writes the changes to disk. The set must be initialized or open, and it must be writable.

Usage

```
// All changes are complete
KeyNameSet->Flush(); //writes to disk
storage->Commit;
```

Get

Pure Virtual.

Storable Category

Input

IAOrderedStorable* key

A “dummy” storable object with the key data of the object to be found.

Output

IAOrderedStorable*

A deep copy of the storable object if it exists, or NULL if it does not.

Notes

Get provides a pointer to a deep copy of an IAOrderedStorable that exists within the set. If the object does not exist, the output pointer will be NULL.

The set must be open for Get to function.

The caller must explicitly delete the object returned by the Get function.

Usage

```
char * existingName = "Liam";
OrderedStorableSubclass anOrderedStorable
    ((byte*)existingName, strlen(existingName));
bool exists = anOSSet->Get(&anOrderedStorable);
if (!exists) {
    printf ("%s is not there; \n", anOrderedStorable.name);
} else (printf ("%s is the data\n", exists.data);
}
```

GetMutex

Pure Virtual.

Output

IAMutex *mutex

The mutex used to lock the storage.

Description

Get the lock or mutex for committing the entire storage in one transaction.

Initialize

Pure Virtual.

Input

IAStorage* storage
The storage allocated to the set. Storage must be open (or initialized).

IABlockID block
The allocated block ID for the root of the set.

bool cloneStoreStream
A command to use a duplicate, or clone, of the store stream to increase throughput in multi-thread applications. True will create the clone. Default is false.

Notes

Initialize establishes the set in storage with its root at the allocated block. The set is opened for output and left open.

If you want to improve throughput when working with multiple threads, you may ask for a cloned store stream.

Usage

```
IABlockID treeRoot = storage->AllocateNamedBlock(treeName);
KeyNameSet->Initialize(storage, treeRoot, );
```

MakeIterator()

Pure Virtual.

Output

IAOrderedStorableIterator*
An iterator set to the beginning of the set.

Notes

MakeIterator creates an IAOrderedStorableIterator positioned before the first IAOrderedStorable in the set. The first call of Next() will return the first object.

This function, or MakeIterator(IAOrderedStorable* key), should be used to construct the iterator.

Storable Category

You can have multiple concurrent iterators on the same set. Iterators will function correctly during concurrent updates to the set.

Usage

```
IAOrderedStorableIterator* iter =KeyNameSet->MakeIterator();
//get the smallest one (first one) in the collection
KeyName* baby = (KeyName*)iter->Next();
```

MakeIterator(IAOrderedStorable* key)

Pure Virtual.

Input

IAOrderedStorable* key

An object in the set. Only the key data is required to be present. This object is used to locate the item with the key and position the iterator at that item.

Output

IAOrderedStorableIterator*

An iterator set to the item whose key matches the input, or, if that item is not in the set, set to the next highest item.

Notes

MakeIterator(key) constructs an iterator. If the input IAOrderedStorable exists in the set, the iterator is positioned such that it will return that object when Next() is called. If the storable does not exist in the set, the iterator will return the next greater object.

Usage

```
// List all names after a given point in the list

IAOrderedStorableIterator*
    nameIter = KeyNameSet->MakeIterator(&initialLetter);
KeyName* newName = (KeyName*)nameIter->Next();
while (newName!= NULL) {
    printf ("%s\n", (char*)newName->name);
    newName = (KeyName*)nameIter->Next();
}
```

Open

Pure Virtual.

Input

IStorage* storage
The storage allocated to the set. Storage must be open (or initialized).

IABlockID block
The allocated block for the root of the set.

bool writable
Whether the set may be altered, or written. True if the set is writable, false if it is read-only. This must be true to Flush, Put, or Remove.

bool cloneStoreStream
A command to use a duplicate, or clone, of the store stream to increase throughput in multi-thread applications. True will create the clone. Default is false.

Notes

Open opens an existing ordered set. It is assumed that this set is rooted at the allocated block.

Setting writable to true allows the set to be updated; otherwise the set is read-only.

A cloned store stream may improve throughput for multithreaded applications.

Usage

```
KeyNameSet = IMakeOrderedStorableSet(new KeyName());
IABlockID treeRoot = storage->TOC_Get(treeName);
KeyNameSet->Open(storage, treeRoot, writable, true);
```

PositionEstimate

Pure virtual.

Input

IAOrderedStorable* key
An object whose position in the set is to be estimated.

Storable Category

Output

float

The fraction of the set that lies before the named key.

Notes

In conjunction with TotalSize(), this function can be useful in estimating the cost of range iteration.

Purge

Pure virtual.

Notes

Purges any cached data from memory.

Put

Pure Virtual.

Input

IAOrderedStorable* obj

The object to be placed in the set.

Output

bool

The results of the put. True if the object was replaced, false if it was added.

Notes

Put places the input IAOrderedStorable object into the ordered set. If the object is already in the set, it is replaced. An object is considered to be in the set if it Equals another object in the set. See “Equal” on page 10-15.

The IAT assumes responsibility for deleting the object passed to Put.

The ordered storable set must be opened and writable (or initialized) before a put will work.

Put caches the changes. You must Flush the set to write the changes to disk (and commit the storage). Changes made to the set by Put will be reflected in iterators and Gets before the set is flushed, however.

Storable Category

Usage

```
bool isReplaced = KeyNameSet->Put(&outputKey);
if (isReplaced){
    printf("%s has been replaced\n", outputKey.name);
} else{
    printf("%s has been added \n");outputKey.name);
}
```

Remove

Pure Virtual.

Input

IAOrderedStorable* key
A storable object with the key data of the object to be removed.

Output

bool
The results of the remove. True if the object was removed, false if it was not found.

Notes

Remove deletes the **IAOrderedStorable** matching the key object from the collection and thus from persistent storage, but does not delete the argument object from memory. Remove returns “true” if the object was found and removed from the set, “false” if the object did not exist in the set.

Remove changes the cache. You must Flush the set to write the changes to disk (and commit the storage). Changes made to the set by Remove will be reflected in iterators and Gets before the set is flushed.

Usage

```
bool isRemoved = KeyNameSet->Remove(&key);
if (isRemoved) {printf ("%s has been removed\n", key.name);}
else {printf ("%s was already gone\n", key.name);}
```

Storable Category

TotalSize

Pure virtual.

Output

uint32

The total number of bytes of storage allocated by the set.

Notes

In conjunction with PositionEstimate(), this function can be useful in estimating the cost of range iteration.

IStorable

Class

Header: IStorable.h

Hierarchy

Superclass: None

Abstract Base Class.

Description

An IStorable is an object that may be stored on disk or within a data structure.

Relationships

IStorable stores in IOutputBlock

One storable stores in one output block.

IStorable restores from IInputBlock

One storable restores from one input block.

Public Member Functions

DeepCopy

Pure Virtual.

Output

IAStorable*
A copy of this object.

Notes

Deep Copy returns a copy of the object itself as an IAStorable. There is no copy constructor defined for an IAStorable to avoid hidden type errors.

Listing 10-19 Sample Implementation of DeepCopy

```
IAStorable*HFSVolumeInfo::DeepCopy() const {
    byte* newName = IAMallocArray(byte, name[0] + 1);
    pstrcpy(newName, name);
    return new HFSVolumeInfo(vRefNum, newName, creationDate);
}
```

Restore

Pure Virtual.

Input

IAInputBlock* input
The input block containing the object and positioned at that object.

Output

IAStorable*
The object existing at the set position of the input block.

Storable Category

Notes

Restore reads a previously stored object from storage. It reads StoreSize() bytes from input.

IAInputBlock is a block allocated to the existing storage. This block must be established and contain a store stream pointing at the beginning of this stored object.

Similar to Deep Copy, implementations of restore should use the protected member function Restoring to copy the data.

Listing 10-20 Sample Implementation of Restore

```
IAStorable*HFSVolumeInfo::Restore(IAInputBlock* input) const {
    // read name
    byte length = input->ReadByte();
    byte* newName = IAMallocArray(byte, length + 1);
    newName[0] = length;
    input->ReadBuffer(newName + 1, length);

    long newCreationDate = input->ReadUInt32();// read creationDate
    short newVRefNum = FindVRefNum(newName, newCreationDate);//
    if (newVRefNum == 0)
        return NULL;
    else
        return new HFSVolumeInfo(newVRefNum, newName,
                                   newCreationDate);
}
```

StoreSize

Pure Virtual.

Output

IABlockSize*

An integer representing the storage size in bytes of a single storable object.

Notes

This function returns the amount of storage that will be used when this storable object is stored.

IABlockSize is a typedef of uint32. It represents the number of bytes the object will occupy after serialization for output.

Storable Category

Listing 10-21 Sample Implementation of StoreSize

```

IABlockSizeHFSVolumeInfo::StoreSize() const {
    return 1 + name[0] + sizeof(uint32);
}

```

Store

Pure Virtual.

Input

IAOutputBlock* output
 The output block positioned at the next available slot.

Notes

Store outputs the storable object to storage. It will write StoreSize() bytes to output. IAOutputBlock is an output block allocated to the storage that is to be used. It must be established and contain a store stream pointing to the position in which to write the object.

Listing 10-22 Sample Implementation of Store

```

void HFSVolumeInfo::Store(IAOutputBlock* output) const {
    output->WriteByte(name[0]);
    output->WriteBuffer(name + 1, name[0]);
    output->WriteUInt32(creationDate);
}

```

Protected Member Functions**DeepCopying**

Pure Virtual.

Storable Category

Input

IAStorable* source
 “this” object.

Output

void
 The data members of the object are updated with the input data; nothing is returned.

Notes

When the creation of a copy requires several steps, it is clearer to implement this internal routine to simplify the copy. If you do DeepCopying on a new object it will move the data items of the input object into place.

A map, for example, is a storable that contains two other storables. This example is a directory of names and numbers. Name and number are each contained storable objects.

Listing 10-23 Sample Implementation of Deep Copy and Deep Copying

```
IAStorable* StorableSubClass::DeepCopy() {
    StorableSubClass* copy = new StorableSubClass;
    copy->DeepCopying(this);
    return copy;
}
void StorableSubClass::DeepCopying(IAStorable* source) {
    StorableSubClass* other = (StorableSubClass*) source;
    name = (ContainedStorable)other->name->DeepCopy();
    number = (ContainedStorable)other->number->DeepCopy();}
}
```

The source is “this,” the object itself which is to be duplicated in this routine.

Restoring

Pure Virtual.

Input

IAInputBlock* input
 The input block containing the item with the store stream positioned at its beginning.

Storable Category

IAStorable* proto

An empty new object to be used as a prototype for the restore.

Notes

Restoring is an internal routine used when the creation of the storable object requires several steps.

If the storable object were a map, for example, of two other storables, this function will simplify the copy. This example shows a restore of a directory of names and numbers.

Listing 10-24 Sample Implementation of Restore and Restoring

```
IAStorable*StorableSubClass::Restore(IAInputBlock* input) {
    StorableSubClass* restoredObject = new StorableSubClass;
    restoredObject->Restoring(input, this);
    return restoredObject;
}
void StorableSubClass::Restoring(IAInputBlock* input,
                                IAStorable* source) {
    StorableSubClass* other = (StorableSubClass*) source;
    name = (ContainedStorable*)other->name->Restore(input);
    number =(ContainedStorable*)other->number->Restore(input);
}
```

Class Utilities

IAMakeOrderedStorableSet

Header: IAStorable.h

Input

IAOrderedStorable* proto

Used as a prototype for the Restore() functions. An empty example of the type of item stored in the set.

Output

IAOrderedStorableSet*

An empty storable set for the input object.

Storable Category

Notes

This constructs an `IAOrderedStorableSet`. This must be used rather than an explicit constructor. The input object provided is used as a prototype to establish **new** objects of the type.

Usage

```
IAOrderedStorableSet* KeyNameSet=
    IAMakeOrderedStorableSet(new KeyName());
```

Externs

```
extern bool IACloneOSSetStoreStreams;
    When true, OrderedStorableSets will use cloned StoreStreams. False by
    default.
```

Exceptions and Error Handling

Errors are currently handled by throwing exceptions.

VSBE

OrderedStorableSetEntryTooBig
 The store size is greater than the `IABlockSize / 2`. Currently, store sizes should be less than 2K.

Class

`IAOrderedStorableSet`

Storable Category

Alphabetical List of Functions

AddDoc 5-51	DeepCopying 10-30
AdvanceTo 7-38	DeepCopying 8-25
Allocate 9-32	DeepCopying 8-41
AllocateNamedBlock 9-33	DeleteDoc 5-35
AllocData 7-30	DeleteDoc 5-51
Clients 10-15	Destroy 10-20
Clients 6-28	DocID 5-74
Clients 6-30	DocLength 5-74
Clone 9-19	Equal 10-15
Clone 9-40	Equal 5-26
CollectDirInfo 8-35	Equal 5-44
Commit 9-33	Equal 5-61
Compact 5-35	Equal 7-28
Compact 5-48	Equal 8-22
Compact 9-33	Equal 8-40
ComponentsRead 5-64	EqualNonVirtual 7-29
ComponentsSize 5-64	Flush 10-20
ComponentsWrite 5-65	Flush 5-36
Count 10-19	Flush 5-51
CurrentPos 7-38	Flush 9-19
Deallocate 9-34	Flush 9-41
DeepCopy 10-28	FlushProgressFn 5-74
DeepCopy 5-26	FreeSpace 9-34
DeepCopy 5-61	Get 10-20
DeepCopy 5-67	GetAccessorType 6-24
DeepCopy 6-44	GetAnalysis 5-36
DeepCopy 7-28	GetBuffer 7-39
DeepCopy 7-46	GetBufferPos 7-39
DeepCopy 8-22	GetBytesForUpdate 5-59

Alphabetical List of Functions

GetCharStream 7-20	GetFileName 8-23
GetComponents 5-65	GetFileName 9-20
GetComponents 6-49	GetFileType 9-20
GetCorpus 5-36	GetFlushProgressData 5-53
GetCorpusType 8-43	GetFlushProgressFn 5-53
GetCreationDate 8-30	GetFlushProgressFreq 5-53
GetCreator 9-19	GetFRefNum 9-20
GetData 7-29	GetFreq 5-29
GetDataLength 7-29	GetFreqPostings 5-49
GetDeletedDocCount 5-49	GetHighFreqTerms 5-72
GetDir 8-33	GetIDDoc 5-54
GetDirCount 8-33	GetIDTerm 5-54
GetDirID 8-23	GetIndex 6-28
GetDirID 9-19	GetIndex 6-31
GetDirIndex 8-33	GetIndexCount 6-24
GetDirInfos 8-34	GetIndexRoot 5-38
GetDocCount 5-36	GetIndexType 5-38
GetDocCount 5-51	GetIndexTypes 5-38
GetDocID 5-26	GetIndices 6-24
GetDocID 5-28	GetInvertedRankedQueryMaxTerms 6-35
GetDocInfo 5-52	GetInvertedRankedQueryMinTerms 6-35
GetDocInfoIterator 5-52	GetMatchingTerms 6-44
GetDocIterator 5-37	GetMatchingTermsLen 6-44
GetDocIterator 5-53	GetMaxDocID 5-55
GetDocIterator 8-37	GetMaxDocumentSize 5-39
GetDocIterator 8-43	GetMaxTermID 5-55
GetDocIterator 5-37	GetModDate 8-40
GetDocIterator 5-53	GetMutex 10-21
GetDocText 8-18	GetMutex 9-35
GetDocText 8-44	GetMutex 9-40
GetDocTopic 6-38	GetName 8-23
GetDocTopic 6-52	GetName 8-48
GetDocument 5-26	GetNamedBlock 9-35
GetDocument 6-28	GetNextBuffer 7-22
GetDocument 6-31	GetNextBuffer 7-42
GetDocumentCount 5-61	GetNextBuffer 8-26
GetDocumentLength 5-26	GetNextBuffer 8-50
GetDocumentLength 5-65	GetNextChar 7-39
GetDocumentLength 6-49	GetNextCharInBuffer 7-40
GetEndChar 7-39	GetNextDoc 8-49
GetEndPosition 7-32	GetNextToken 7-19
GetEOF 9-19	GetNextToken 7-24
GetEOF 9-41	GetNextToken 7-34

Alphabetical List of Functions

GetNextToken 7-36	GetVolumeRefNum 8-30
GetNextToken 7-43	GetVolumeRefNum 8-37
GetPBRec 8-33	GetVRefID 8-19
GetPercent 6-31	GetVRefNum 8-19
GetPosition 9-24	GetVRefNum 9-20
GetPosition 9-26	HFSCorpus 8-16
GetPreferredAnalysis 5-39	HFSIterator 8-16
GetProtoDoc 8-18	HFSTextFolderCorpus 8-16
GetProtoDoc 8-37	HitEqual 6-39
GetProtoDoc 8-44	HitLessThan 6-40
GetProtoTerm 7-26	IAAccessor 6-20
GetProtoTerm 7-45	IAAssertion 4-4
GetQueryAnalysis 5-39	IABlockAddress 9-48
GetRefNum 8-27	IABlockID 9-48
GetRootDirID 8-37	IABlockSize 9-48
GetScore 6-45	IACorpus 8-16
GetStartPosition 7-33	IADefaultMutexConstructor 9-30
GetStorage 5-40	IAFree 4-7
GetStorageType 9-35	IAFreeArray 4-8
GetStoreStream 9-35	IAFreeArraySized 4-9
GetStreamBuffer 7-20	IAFreeSized 4-7
GetTerm 5-61	IAFreeStruct 4-9
GetTerm 6-46	IAMakeOrderedStorableSet 10-32
GetTerm 7-33	IAMakeStorage 9-45
GetTermCount 5-56	IAMalloc 4-6
GetTermID 5-61	IAMallocArray 4-7
GetTermInfo 5-56	IAMallocArraySized 4-8
GetTermInfoIterator 5-56	IAMallocSized 4-7
GetTermInfoIterator 5-57	IAMallocStruct 4-9
GetTextSpan 7-20	IAMutexConstructor 9-49
GetTextSpan 7-35	IAReadIndexTypes 5-72
GetTextSpan 7-37	IAThrowException 4-4
GetTextSpan 7-40	Increment 8-34
GetTFVector 5-70	Initialize 10-22
GetTheDirID 8-27	Initialize 5-40
GetTheFileName 8-27	Initialize 5-49
GetTheVolumeRefNum 8-27	Initialize 5-57
GetTWVector 6-52	Initialize 6-24
GetVectorBlockID 5-66	Initialize 7-25
GetVolumeCount 8-19	Initialize 8-45
GetVolumeInfos 8-20	Initialize 9-20
GetVolumeName 8-30	Initialize 9-36
GetVolumeRefID 8-23	Initialize 9-42

Alphabetical List of Functions

Initializing 8-20	Open 8-45
Initializing 8-38	Open 9-21
Initializing 8-46	Open 9-36
InitialSize 8-20	Open 9-42
InitialSize 8-38	Opening 8-20
InitialSize 8-46	Opening 8-38
InVecAccessor 6-20	Opening 8-46
InvertedAccessor 6-20	operator delete 4-11
IsDocIndexed 5-40	operator delete 4-12
IsDocIndexed 5-57	operator new 4-11
IsHit 6-40	operator new 4-12
IsInitializationValid 6-25	PositionEstimate 10-24
IsOpen 9-21	Purge 10-25
IsOpen 9-36	Put 10-25
IsOpen 9-40	RankedAccessor 6-20
IsWritable 9-21	RankedProgressFn 6-53
IsWritable 9-36	RankedSearch 6-33
IsWritable 9-41	RankedSearch 6-34
LessThan 10-16	RankedSearch 6-41
LessThan 5-27	RankedSearch 6-42
LessThan 5-62	RankedSearch 6-53
LessThan 7-29	RankedSearchBoolean 6-34
LessThan 8-23	Read 9-22
LessThan 8-40	Read 9-43
LessThanNonVirtual 7-30	ReadBuffer 9-24
Lock 9-30	ReadByte 9-24
MakeHFSStorage 9-45	ReadUInt32 9-25
MakeIterator 10-22	Remove 10-26
MakeIterator 10-23	RemoveNamedBlock 9-37
MakeTokenStream 7-26	RenameDoc 5-41
MakeTokenStream 7-45	RenameDoc 5-58
MaybeFlushBuffer 9-42	Restore 10-28
Merge 5-41	Restore 5-27
Merge 5-57	Restore 5-62
MergeHits 6-41	Restore 5-67
Next 10-18	Restore 7-30
Next 5-30	Restore 8-24
Normalize 6-49	Restore 8-30
Open 10-24	Restore 8-40
Open 5-41	Restoring 10-31
Open 5-49	Restoring 8-25
Open 5-57	Restoring 8-41
Open 7-27	SetAccessorType 6-25

Alphabetical List of Functions

SetBuffer 7-41	SetVectorBlockID 5-66
SetBufferPos 7-41	SetVolumeCount 8-20
SetBytesForUpdate 5-59	SetVolumeInfos 8-21
SetCharStream 7-20	SetVolumeName 8-31
SetComponents 5-65	SetVolumeRefID 8-24
SetComponents 6-50	SetVolumeRefNum 8-31
SetCreationDate 8-31	SetVolumeRefNum 8-38
SetDirID 8-24	ShortWordFilter.h 7-16
SetDirIndex 8-35	Similarity 6-50
SetDocument 5-27	Store 10-30
SetDocument 6-29	Store 5-27
SetDocument 6-31	Store 5-62
SetDocumentCount 5-62	Store 5-67
SetDocumentLength 5-65	Store 7-30
SetDocumentLength 6-50	Store 8-24
SetEndChar 7-41	Store 8-31
SetEOF 9-21	Store 8-41
SetEOF 9-43	StoreInitialization 6-26
SetFileName 8-24	StoreSize 10-29
SetFlushProgressData 5-58	StoreSize 5-27
SetFlushProgressFn 5-58	StoreSize 5-62
SetFlushProgressFreq 5-58	StoreSize 5-67
SetFRefNum 9-21	StoreSize 7-30
SetIndex 6-29	StoreSize 8-25
SetIndex 6-32	StoreSize 8-31
SetIndexCount 6-26	StoreSize 8-41
SetIndices 6-26	StoreSize 5-29
SetInvertedRankedQueryMaxTerms 6-36	StoreSize 5-29
SetInvertedRankedQueryMinTerms 6-36	Sum 6-51
SetMaxDocumentSize 5-42	TermFreq 5-75
SetModDate 8-40	TermID 5-75
SetNextCharInBuffer 7-41	Text 7-46
SetPercent 6-32	TextLen 7-46
SetPreferredAnalysis 5-42	TOC_Get 9-37
SetRefNum 8-28	TOC_Remove 9-38
SetRootDirID 8-38	TOC_Set 9-38
SetScore 6-45	TotalSize 10-27
SetStreamBuffer 7-21	TotalSpace 9-39
SetTerm 5-62	TWVector 6-20
SetTerm 6-46	Unlock 9-30
SetTheDirID 8-28	Update 5-43
SetTheFileName 8-28	Update 8-45
SetTheVolumeRefNum 8-28	UpdateSize 8-21

Alphabetical List of Functions

UpdateSize 8-39
 UpdateSize 8-47
 Updating 8-21
 Updating 8-39
 Updating 8-47
 VAAI 6-54
 VAIIV 6-54
 VANI 6-54
 VASU 7-48
 VCHE 8-51
 VCHV 8-51
 VCID 8-51
 VectorAccessor 6-21
 VIAI 5-79
 VIAO 5-79
 VIDN 5-79
 VIIV 5-79
 VINO 5-80
 VInt32Read 9-46
 VInt32Size 9-46
 VInt32Write 9-47
 VSAO 9-50
 VSBE 10-33
 VSBI 9-50
 VSDF 9-50
 VSEo 9-51
 VSEr 9-50
 VSIV 9-51
 VSNI 9-51
 VSNO 9-51
 VSNW 9-52
 VSPB 9-52
 VTSU 7-48
 Write 9-22
 Write 9-44
 WriteBuffer 9-27
 WriteByte 9-27
 WriteUInt32 9-27

A P P E N D I X A

Alphabetical List of Functions

This Apple manual was written, edited,
and composed on a desktop publishing
system using Apple Macintosh
computers and FrameMaker software.
Line art was created
using Adobe Illustrator[™]. PostScript was
developed by Adobe Systems
Incorporated.

Text type is Palatino[®] and display type is
Helvetica[®]. Bullets are ITC Zapf
Dingbats[®]. Some elements, such as
program listings, are set in Apple Courier.