

WAIS Objective-C Application Program Interface

by Paul Burchard <burchard@math.utah.edu>

Introduction.

The Objective-C interface to WAIS, which was used to construct the NeXT WAISStation client, consists of an abstract `Wais` superclass and its three subclasses `WaisQuestion`, `WaisSource`, and `WaisDocument`. The abstract superclass handles management tasks like accessing WAIS objects by name and loading them from disk, managing an object's information fields, and Mach thread support functions. The basic relationship between the subclasses is:

- Documents are retrieved from Sources.
- Questions produce lists of documents and relevance scores, when given:

- ⌘ a string of keywords to match,
- ⌘ a list of Sources to search,
- ⌘ a list of relevant Documents to emulate.

Here is a sample of how the object-oriented interface makes it easy to search a source `mydatabase.src` for information about the topic ``widgets'', retrieve the resulting documents, and do something with them:

```
id results, myquestion = [[WaisQuestion alloc] initWithKey:NULL];

[myquestion setKeywords:"widgets"];
[myquestion addSource:[WaisSource objectForKey:"mydatabase.src"]];

results = [myquestion search];
```

```
[results makeObjectsPerform:@selector(retrieve)];  
  
// Now do something with the retrieved documents in the results list...  
// Scores are accessed like this: [myquestion scoreForDocument:doc]
```

We begin by presenting the declaration of the abstract `Wais` class, follow with detailed explanations of its functionality, and finally, the give declarations of the subclasses. [Note: these classes may undergo some revision when NeXT's DBKit becomes available.]

The Abstract Wais Class.

```
@interface Wais : Object  
{  
    NXAtom key;
```

```
    id infoFields;  
}
```

Access by object keys; search paths for those keys:

```
+ objectForKey:(const char *)aKey;  
+ objectForCompleteKey:(const char *)aKey;  
- (const char *)key;  
- setKey:(const char *)aKey;  
+ waisObjectList;  
+ folderList;  
+ setFolderList:aList;  
+ (const char *)defaultHomeFolder;
```

Creating and destroying WAIS objects:

```
+ initialize;  
- initKey:(const char *)aKey;  
- free;
```

Editing info fields:

```
- (const char *)valueForKey:(const char *)aKey;  
- (const char *)insertStringKey:(const char *)aKey value:(const char *)aValue;
```

Reading/writing structured WAIS files:

```
- readWaisFile;
```

```
- writeWaisFile;
+ loadFolder:(const char *)folderName;
- (short)readWaisStruct:(const char *)structName
    forElement:(const char *)elementName
    fromFile:(FILE *)file
    withDecoder:(WaisDecoder)theDecoder;
- (short)writeWaisStruct:(const char *)structName
    forElement:(const char *)elementName
    toFile:(FILE *)file
    withDecoder:(WaisDecoder)theDecoder;
+ (const char *)fileStructName;
+ (WaisDecoder)fileStructDecoder;
+ (BOOL)checkFileName:(const char *)fileName;
```

Routing error messages:

```
+ setQuiet:(BOOL) yn;  
+ (BOOL)isQuiet;  
+ setStringTable:aTable;  
+ (const char *)errorTitle;
```

Support for multiple threads:

```
+ lockTransaction;  
+ unlockTransaction;  
+ lockFileIO;  
+ unlockFileIO;  
+ callback:anObject perform:(SEL)aSelector with:anArgument;  
+ (port_t)callbackPort;  
+ waisNewLocks;
```

@end

Name Lookup Service.

To facilitate user manipulation of Sources and Documents (which can be shared among multiple Questions), WAIS objects can be accessed by string keys via the `+objectForKey:` class method. The string key is the full path of the file in which the object is stored or would be stored upon retrieval. WAIS objects will be automatically loaded from their WAIS structured files as necessary when a key lookup is performed. Note that keys are actually `NXAtoms` (uniqued strings).

To support lookup with incomplete keys and creation of objects from incomplete keys, each class also has a `+folderList` (a `Storage` object of `NXAtoms`) which is used as a

search path. For creation, the first element of the folder list is used. Each subclass should set its own folder list, initially containing an appropriate subfolder of `~/Library/WAIS/`.

Information Fields.

All WAIS objects contain lists of information fields, which are stored via associative tables connecting field names to values. These info fields encode most of the parameters of the WAIS objects. These info fields, along with any other necessary data, are read from and written to WAIS structured files (see below) using the `-readWaisFile` and `-writeWaisFile` methods (which use the key as the file name). Note that these info field keys and values are copied by these methods and don't need to be held fixed by the application program.

Decoding structured WAIS files.

WAIS structured files contain a single structure, of the form

(*:struct-name elements*)

where *elements* is a sequence composed of:

- Fields, of the form

:element-name data

- Substructures, of the form

:element-name (:struct-name elements)

- Lists of structures of a single type, of the form

*:element-name ((:struct-name elements)
(:struct-name elements) ...)*

For each structure there is a `WaisDecoder` which tells the software what types and names

of elements to expect in the structure. The decoders give the type, `W_FIELD`, `W_STRUCT`, or `W_LIST`, for each element name. The declaration of the decoder table is:

```
typedef struct waisDecoder
{
    const char *name;
    int elementType;
    const char *structName;
    struct waisDecoder *subDecoder;
    long (*reader)();
    int readArgs;
    long (*writer)();
    int writeArgs;
    unsigned int maxBufSize;
}
```

```
_WaisDecoder, *WaisDecoder;
```

W_FIELDS are read/written as formatted strings by the `reader()` and `writer()`. These are stored in the WAIS object as a table associating field name to field value. Args of field readers/writers are (annoyingly) arranged thus:

<u># Args</u>	<u>Reader/writer Template</u>
---------------	-------------------------------

1	long <i>function</i> (FILE *file)
---	-----------------------------------

2	long <i>function</i> (char *buffer, FILE *file)
---	---

3	long <i>function</i> (char *buffer, FILE *file, int bufsize)
---	--

(Actually the arg patterns are even less consistent in the WAIS IR library, so we have replaced `ReadLong()`, `WriteLong()`, `ReadDouble()`, and `WriteDouble()`.) The return values are `TRUE`, `FALSE`, or `END_OF_STRUCT_OR_LIST`.

`W_STRUCTS` are read/written with a recursive call to the `-readWaisStruct:::` / `-writeWaisStruct:::` methods. This allows subclasses of `Wais` to intercept these calls or make preparations before calling super to do the actual I/O (e.g. creating objects for info to be read into/written from).

`W_LISTS` are handled similarly, except that the recursive call is repeated until a `FALSE` or `END_OF_STRUCT_OR_LIST` return. Note that the element name of the list is passed as a method parameter so that subclasses can find the right list of objects in which to store the structures of the list (however be aware that this arg may be `NULL`).

The `-readWaisStruct:::` / `-writeWaisStruct:::` methods should quietly pass over unknown fields and structures if possible, and return `TRUE`, `FALSE`, or

`END_OF_STRUCT_OR_LIST` as appropriate. The `+fileStructName` method should return the `structName` expected at the beginning of the file for that class, and `+fileStructDecoder` gives the corresponding `WaisDecoder`.

Mach and NeXTstep support.

If so enabled during compilation (by `#define-ing` `WAIS_THREAD_SUPPORT`), these classes provide support for running WAIS procedures (typically, retrievals and searches) in separate Mach threads. The support has two parts: the first is a set of mutex locking methods, which are also used here internally. The second is a callback method that allows you to send thread-unsafe object messages (such as those involving the NeXTstep AppKit) back to the main thread for execution.

Aside from setting up callbacks (if so enabled), these classes use the NeXTstep AppKit only

to put up Alert Panels; and even this is only enabled if an `NXStringTable` of error messages has been established via the class method `+setStringTable:.` The `NXStringTable` of error messages is shared among the current classes, so a call to `[Wais setStringTable:¼]` is enough. (The list of string keys this table must contain can be found in the file `WAIS.strings` of the `WAIStation.app` program.)

The mutex locking methods can be used to prevent conflicts between multiple threads of execution. These methods are no-ops if thread support is not enabled. If your program uses threads and contains potential conflicts with the operations below, you'll need to frame those operations with the corresponding pair of Wais class locking methods.

`+lockTransaction/+unlockTransaction`

Frames all uses of the network for WAIS search and retrieval

transactions. (Typically on port 210.)

```
+lockFileIO/+unlockFileIO
```

Frames all reading and writing of files, `stderr` messages, and so on.

The callback method lets you send object messages back to the main thread for execution. This is useful for asynchronous program events like user actions and signals which cannot be mutex locked. The callback is implemented with Mach messaging. If thread support is not enabled, the callback just directly performs the object message.

```
+callback:anObject perform:(SEL)aSelector with:anArgument
```

Just like calling `[anObject aSelector:anArgument]` except

that it runs (asynchronously) in the main thread.

If a thread is aborted in the middle of a locked procedure, you may need to create new mutexes to prevent blockage:

```
+waisNewLocks
```

Other General Notes.

The `+loadFolder:` method returns a `List` of `Wais` objects of a given type, free it after use.

The WaisDocument Subclass.

There are two local files associated with a `WaisDocument`. The *contents* file is that named by the key, which can be *any* type of file. This is the file created by the `-retrieve` method. The *WAIS specifications* file (a WAIS structured file) has, as its name, the key with `.wais` appended (the string `W_D_EXT` from `Wais.h`). This is the file read/written by the `-readWaisFile/-writeWaisFile` methods. (Their associated `.wais` files allow documents to be reused in later `WAIStation` sessions.) The `-initKey:`, `-setKey:`, and `+objectFor[Complete]Key:` methods ignore any terminal `.wais`. The `+checkFileName:` method requires the `.wais`, but that is correct since this method is used to determine which files the `-readWaisFile/-writeWaisFile` methods can be applied to.

Note: we reproduce here *only* the methods which are new to this subclass (not listing those

which are merely subclassed).

```
@interface WaisDocument : Wais
{
    id fromSource;
    DocID *waisDocID;
    BOOL isRetrieved;
}
```

Retrieving Documents from a Source:

```
- retrieve;
- (BOOL)isRetrieved;
- setUnretrieved;
- fromSource;
```

```
- setFromSource:aSource;  
- (DocID *)waisDocID;  
- setWaisDocID:(DocID *)theDocID;  
- setWaisDocIDFromAny:(any *)docAny;  
- setKeyFromInfo;
```

@end

The `-setKeyFromInfo` method uses the `:headline`, `:type`, and `:filename` info fields. The `theDocID` argument to `-setWaisDocID:` should be `s_free()`-able.

The WaisQuestion Subclass.

The main method here is `-search:`, which returns a `List` of `WaisDocuments` (the `-resultList`) upon successful search. These results are sorted in decreasing order of scores, which are normalized to the range 0.0 to 1.0. The scores can be retrieved with the `-scoreForDocument:` method.

Note: we reproduce here *only* the methods which are new to this subclass (not listing those which are merely subclassed).

```
@interface WaisQuestion : Wais
{
    id sourceList;
    id relevantList;
    id resultList;
    id scoreList;
```

```
    unsigned int searchLimit;  
    int listCounter;  
}
```

Setting up for search:

- setKeywords:(const char *)theText;
- (const char *)keywords;
- addSource:waisSource;
- removeSource:waisSource;
- clearSources;
- sourceList;
- addRelevantDocument:waisDocument;
- removeRelevantDocument:waisDocument;
- clearRelevantDocuments;

```
- relevantList;  
- setSearchLimit:(int)maxDocs;  
- (int)searchLimit;  
+ setSearchLimit:(int)maxDocs;  
+ (int)searchLimit;
```

Getting search results:

```
- search;  
- resultList;  
- (float)scoreForDocument:waisDocument;
```

@end

The actual maximum number of search results is the maximum of the class-wide and

instance-specific `searchLimits`. Instance-specific search limits are initialized to one.

The WaisSource Subclass.

In some of the WAIS software there is a distinction between sources and source-id's. This distinction is eliminated here, with sources having a `:filename` info field so they can be act as their own source-id's. This info field is kept consistent with the source's object-lookup key.

The user registration method sets the string that remote servers will be sent for logging purposes. The string should include the user's name, the host name, and the application from which these methods are being called.

Note: we reproduce here *only* the methods which are new to this subclass (not listing those which are merely subclassed).

```
@interface WaisSource : Wais
{
    FILE *connection;
    BOOL isConnected;
    long bufferLength;
    id dataFileList;
    int listCounter;
}
```

Managing connections to the database server:

- `setConnected: (BOOL) yn;`

```
- (BOOL) isConnected;  
- (FILE *) connection;  
- (long) bufferLength;  
+ registerUser:(const char *)userInfoString;
```

Managing the list of files and folders to be indexed, for a local source:

```
- dataFileList;  
- addDataFile:data;  
- removeDataFile:data;  
- clearDataFiles;
```

@end