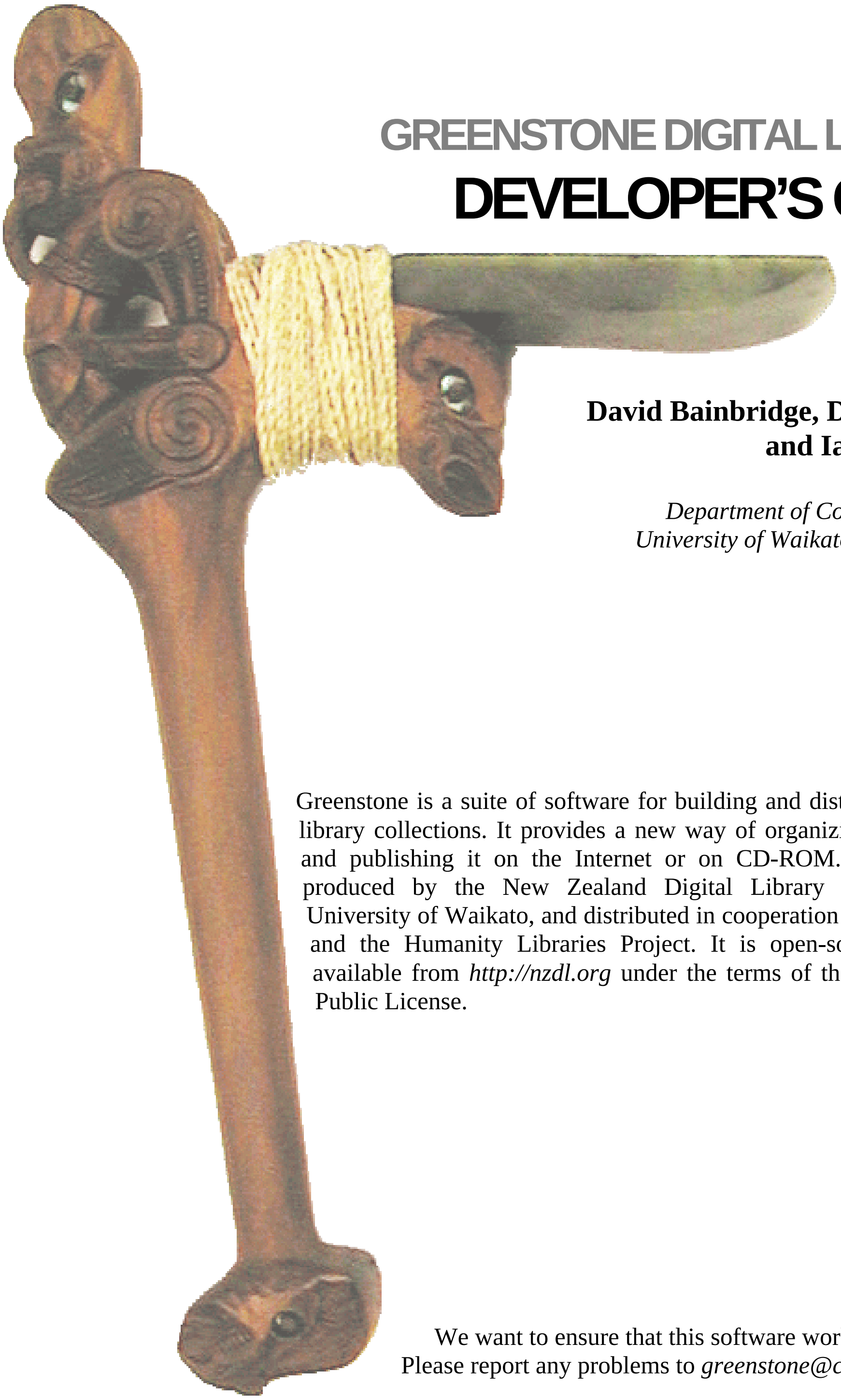


GREENSTONE DIGITAL LIBRARY

DEVELOPER'S GUIDE

David Bainbridge, Dana McKay
and Ian H. Witten

Department of Computer Science
University of Waikato, New Zealand

Greenstone is a suite of software for building and distributing digital library collections. It provides a new way of organizing information and publishing it on the Internet or on CD-ROM. Greenstone is produced by the New Zealand Digital Library Project at the University of Waikato, and distributed in cooperation with UNESCO and the Humanity Libraries Project. It is open-source software, available from *http://nzdl.org* under the terms of the GNU General Public License.

We want to ensure that this software works well for you.
Please report any problems to *greenstone@cs.waikato.ac.nz*

Greenstone gsdl-2.31 **February 2001**

About this manual

This manual explains how Greenstone works. It is aimed at those who wish to customise collections and to develop and maintain the software.

Section 1 gives an insider's account of the collection-building process, including the directory structure and internal document format, and the configuration file that governs the structure of each collection. Section 2 describes the parts of Greenstone that process source documents (and metadata) and dictate how the information is accessed through the user interface. It also describes "external" software components that are distributed with Greenstone but were not implemented by the NZDL group. Section 3 includes a comprehensive explanation of the structure of the Greenstone runtime system, and also gives details of the software that help to understand how it works and how to modify the system to suit different needs. Section 4 describes the Greenstone configuration files, and an Appendix introduces the C++ Standard Template Library.

When working with the Greenstone software, you may encounter references to features that are not described in this manual, for Greenstone is under constant development. To learn about software under development, join the Greenstone mailing list (instructions at nzdl.org).

Companion documents

The complete set of Greenstone documents includes three volumes:

- Greenstone Digital Library Installer's Guide
- Greenstone Digital Library User's Guide
- Greenstone Digital Library Developer's Guide *(this document)*

Acknowledgements

The Greenstone software is a collaborative effort between many people. Rodger McNab and Stefan Boddie are the principal architects and implementors. Contributions have been made by David Bainbridge, George Buchanan, Hong Chen, Elke Duncker, Carl Gutwin, Geoff Holmes, John McPherson, Craig Nevill-Manning, Gordon Paynter, Bernhard Pfahringer, Todd Reed, Bill Rogers, and Stuart Yeates. Other members of the New Zealand Digital Library project provided advice and inspiration in the design of the system: Mark Apperley, Sally Jo Cunningham, Steve Jones, Te Taka Keegan, Michel Loots, Malika Mahoui and Lloyd Smith. We would also like to acknowledge all those who have contributed to the GNU-licensed packages included in this distribution: MG, GDBM, WGET, WVWARE, PDFTOHTML.

Contents

1
Understanding the collection-building process

End users of Greenstone can build collections using the Collector, described in the *Greenstone Digital Library User's Guide* (Section 3). This makes it very easy to build collections modelled after existing ones but with new content. However, it is not really feasible to use the Collector to create collections with completely new structures. It does invite you to edit the collection configuration file, which governs the collection's structure, but you need to know quite a lot about Greenstone to make radical yet effective changes to the configuration file. This section tells you what you need to know to do this. It also describes the Greenstone directory structure and the format in which documents are stored internally.

We assume throughout this manual that you have installed Greenstone on your computer, be it Windows or Unix. If you have not yet done this you should consult the *Greenstone Digital Library Installer's Guide*. The variable *GSDLHOME* will be used throughout to refer to the Greenstone home directory, which is called *%GSDLHOME%* on Windows systems and *$GSDLHOME* on Unix ones. You set this directory during the installation procedure (on Windows, the default is *C:\Program Files\gsdl*).

1.1 Building collections from the command line

Let us begin by walking through the operations involved in building a collection from the command line, to help understand the process better. Of course, for more user-friendly collection building, you should use the Collector instead. The collection we take as an example is one that comes on the Greenstone software distribution CD-ROM, and contains the WWW home pages of many of the people who have worked on the New Zealand Digital Library Project and the Greenstone software.

Separate subsections follow for building under Windows and Unix. In fact, the two subsections are very nearly identical—you need only go through the one that pertains to your system. When following the walkthrough, you will probably find some operations mysterious and arcane, but follow them closely—their meaning will be explained later on. After the walkthroughs is a brief summary of the differences between building a collection under the two systems.

Collection building under Windows

The first challenge when building a Greenstone collection from the command line under Windows is to get at the "command prompt," the place where you type commands. Try looking in the *Start* menu, or under the *Programs* submenu, for an entry like *MS-DOS Prompt*, *DOS Prompt*, or *Command Prompt*. If you can't find it, invoke the *Run* entry and try typing *command* (or *cmd*) in the dialog box. If all else fails, seek help from one who knows, such as your system administrator.

Change into the *GSDLHOME* directory (that is, *%GSDLHOME%*). At the prompt, type *setup.bat*. This batch file (which you can read if you like) tells the system where to look for Greenstone programs. On Windows 95/98 systems it may fail with an *Out of environment space* error. If this happens, you should edit your system's *config.sys* file (normally found at *C:\config.sys*) and add the line *shell=C:\command.com /e:4096 /p* (where *C:* is your system drive letter). You'll need to reboot for this change to take effect.

Now run the Perl program *mkcol.pl*, whose name stands for "make a collection." Depending on how your system is set up, you might run this program by typing *perl –S mkcol.pl*, or simply *mkcol.pl*. If you're not accustomed to running Perl programs you'll have to experiment. Once you run it, *mkcol.pl* creates the directory structure that is required for a new collection. A description of usage and a list of arguments will appear on the screen. The only required argument is *creator*, which tells who built the collection. A good name for the collection might be *dlpeople*, so type the command *mkcol.pl –creator <your-email-address-here> dlpeople*.

List the contents of your collection's directory. There should be seven subdirectories: *archives*, *building*, *etc*, *images*, *import*, *index* and *perllib*.

Source material for the *dlpeople* collection can be found on the Greenstone distribution CD-ROM under the directory *collect/dlpeople*. The CD-ROM drive is usually *D:\* when using Windows—you can verify this using the *My Computer* icon on your desktop. If you can't find it,

Figure 1
Collection configuration file created by *mkcol.pl*

```
creator        dana@example.com
maintainer     dana@example.com
public         true
beta           true

indexes         document:text
defaultindex    document:text

plugin          ZIPPlug
plugin          GMLPlug
plugin          TEXTPlug
plugin          HTMLPlug
plugin          EMAILPlug
plugin          ArcPlug
plugin          RecPlug

classify        AZList -metadata “Title”

collectionmeta collectionname    "dlpeople"
collectionmeta iconcollection    ""
collectionmeta collectionextra   ""
collectionmeta .document:text    "documents"
```

Figure 2
Collection icon

contact your system administrator.

Next, copy the contents of the *D:\collect\dlpeople* directory into the *dlpeople* collection’s *import* directory. You can do this by selecting all the contents of the *dlpeople* directory and dragging them across using the visual file manager, or by typing the command

xcopy /s d:\collect\dlpeople\* GSDLHOME\collect\dlpeople\import

In the collection’s *etc* directory there is a file called *collect.cfg*. Open it using your favourite text editor. It should look something like Figure 1, which shows the collection configuration file that was created by using the command *mkcol.pl –creator dana@example.com dlpeople*.

Now you are ready to “import” the collection. This is the process of bringing the documents into the Greenstone system, standardising the document format, the way that metadata is specified, and the file structure in which the documents are stored. Type *import.pl* at the prompt to get a list of all the options for the import program. To keep things simple, stick to the basic command *import.pl dlpeople*. Don’t worry about all the text that scrolls past—it’s just reporting the progress of the import. Be aware that importing this collection takes about five minutes on a 1000 MHz computer, and correspondingly longer on slower machines.

Now let's make some changes to the collection configuration file to customize its appearance. First, give the collection a name. This will be treated by web browsers as the page title for the front page of the collection, and used as the collection icon in the absence of a picture. Change the line that reads *collectionmeta collectionname "dlpeople"* to something like *collectionmeta collectionname "The People of the NZDL"*.

Add a description of your collection between the quotes of the line that reads *collectionmeta collectionextra ""*. This will be used as the *About this collection* text on the collection's home page. I added something along the lines of “This collection is made up of the web pages of some of the people who have worked on the Greenstone software.”

You can use any picture you can view in a web browser for a collection icon—the image I created is shown in Figure 2. Put the location of the image between the quotes of the line *collectionmeta iconcollection ""* in the configuration file. To specify the image, use a URL starting with *_httpprefix_*, where *_httpprefix_* is a pointer to your GSDLHOME directory, e.g. *_httpprefix_/collect/dlpeople/images/icon.gif* if you have put the image in the collection's *images* directory.

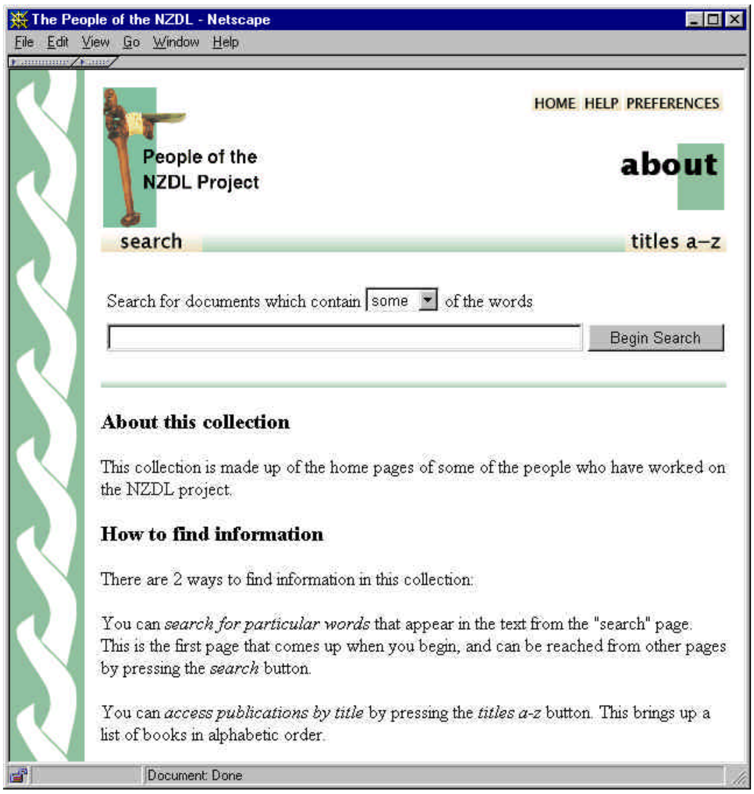
Save the collection configuration file, and close it—you won't need to look at it again during this tutorial.

The next phase is to “build” the collection, which creates all the indexes and files that make the collection work. Type *buildcol.pl* at the command prompt for a list of collection-building options. These options are explained more fully in Section 1.3. For now, stick to the defaults by typing *buildcol.pl dlpeople* at the prompt. Again, don't worry about the “progress report” text that scrolls past.

Make the collection “live” by using the visual file manager to copy the contents of the collection's *building* directory into the *index* directory or from the command line enter *rd /s index,* then *ren building index*.

You should be able to access the collection from your Greenstone homepage. You will have to reload the page if you already had it open in your browser, or perhaps even close the browser and restart it (to prevent caching problems). Alternatively, if you are using the “local library” version of Greenstone you will have to restart the library program. To view the new collection, click on the image. The result should look something like Figure 3.

Figure 3
"About" page for the new collection


The People of the NZDL - Netscape
File Edit View Go Window Help
HOME HELP PREFERENCES
People of the NZDL Project
about
search
titles a–z
Begin Search
Document: Done

Search for documents which contain some of the words

About this collection

This collection is made up of the home pages of some of the people who have worked on the NZDL project.

How to find information

There are 2 ways to find information in this collection:

You can *search for particular words* that appear in the text from the "search" page. This is the first page that comes up when you begin, and can be reached from other pages by pressing the *search* button.

You can *access publications by title* by pressing the *titles a-z* button. This brings up a list of books in alphabetic order.

Collection building under Unix

First change into the *GSDLHOME* directory (that is, *$GSDLHOME*). At the prompt, type *source setup.bash* if you're running the Bash shell, or *source setup.csh* if you're using the C shell. These scripts (which you can read if you like) tell the system where to look for Greenstone programs.

Now type *mkcol.pl*. This is a Perl program whose name stands for "make a collection." It creates the directory structure that is required for a new collection. A description of usage and a list of arguments will appear on the screen. The only required argument is *creator*, which tells who built the collection. A good name for the collection might be *dlpeople*, so type the command *mkcol.pl –creator <your-email-address-here> dlpeople*.

List the contents of your collection's directory. There should be seven subdirectories: *archives*, *building*, *etc*, *images*, *import*, *index* and *perllib*.

Source material for the *dlpeople* collection can be found on the

Table 1 Differences between Windows and Linux, when building collections

Windows	**Linux**
Run *setup.bat* to make Greenstone programs available	Source *setup.bash* or *setup.csh* to make programs available
Copy files from CD-ROM using *D:\* and visual manager or Windows commands	Copy files from CD-ROM using *mount* and Unix commands
New collection index created by typing *xcopy /s building\* index* or using visual file manager	New collection index created by typing *mv building/* index*
Old collection index replaced by typing *rd /s index*, then *ren building index* or by using visual file manager.	Old collection index replaced by typing *rm –R index/** then *mv building/* index*

Greenstone distribution CD-ROM under the directory *collect/dlpeople*. To get information from a CD-ROM under Linux type *mount /cdrom* at the prompt: if this fails, contact your system administrator. Once mounted, the CD-ROM can be used like any other directory, so type *ls /cdrom*. This should reveal a directory called *collect/dlpeople* on the CD-ROM.

Next, copy the contents of the */cdrom/collect/dlpeople* directory into the *GSDLHOME/collect/dlpeople/import* directory. To do this, type the command

```
cp –R /cdrom/collect/dlpeople/* GSDLHOME/collect/dlpeople/import/
```

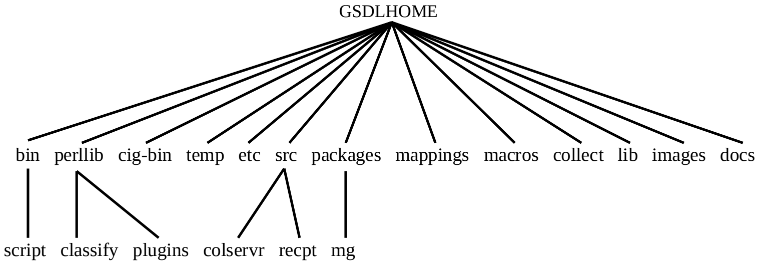
Then type *umount /cdrom/* to close the CD-ROM drive.

In the collection's *etc* directory there is a file called *collect.cfg*. Open this using your favourite text editor. It should look something like Figure 1, which shows the collection configuration file that was created by using the command *mkcol.pl –creator dana@example.com dlpeople*.

Now you are ready to "import" the collection. This is the process of bringing the documents into the Greenstone system, standardising the document format, the way that metadata is specified, and the file structure in which the documents are stored. Type *import.pl* at the prompt to get a list of all the options for the import program. To keep things simple, stick to the basic command *import.pl dlpeople*. Don't worry about all the text that scrolls past—it's just reporting the progress of the import. Be aware that importing this collection takes about five minutes on a 1000 MHz computer, and correspondingly longer on slower machines.

Now let's make some changes to the collection configuration file to

Figure 4
Structure of the *GSDLHOME* directory



customize its appearance. First, give the collection a name. This will be treated by web browsers as the page title for the front page of the collection, and used as the collection icon in the absence of a picture. Change the line that reads *collectionmeta collectionname "dlpeople"* to something like *collectionmeta collectionname "The People of the NZDL"*.

Add a description of your collection between the quotes of the line that reads *collectionmeta collectionextra ""*. This will be used as the *About this collection* text on the collection’s home page. I added something along the lines of “This collection is made up of the web pages of some of the people who have worked on the Greenstone software.”

You can use any picture you can view in a web browser for a collection icon—the image I created is shown in Figure 2. Put the location of the image between the quotes of the line *collectionmeta iconcollection ""* in the configuration file. To specify the image, use a URL starting with *_httpprefix_*, where *_httpprefix_* is a pointer to your GSDLHOME directory, e.g. *_httpprefix_/collect/dlpeople/images/icon.gif* if you have put the image in the collection’s *images* directory.

Save the collection configuration file, and close it—you won’t need to look at it again during this tutorial.

The next phase is to “build” the collection, which creates all the indexes and files that make the collection work. Type *buildcol.pl* at the command prompt for a list of collection-building options. These options are explained more fully in Section 1.3. For now, stick to the defaults by typing *buildcol.pl dlpeople* at the prompt. Again, don’t worry about the “progress report” text that scrolls past.

Make the collection “live” by putting all the material that has just been put in the collection’s *building* directory into the *index* directory. Type *mv building/* index/* (assuming you are in the *dlpeople* directory) at the prompt. If you have built this collection before, first remove the old index using *rm –R index/**.

Table 2 Where in this document to find information about directories

Directory	Contents	Section
bin	Executable code, including binaries in the directory with your O/S name.	–
bin/script	Perl scripts used for creating and building collections (for example *import.pl* and *buildcol.pl*). To get a description of any of these programs, type their name at the command prompt.	1.3
perllib	Perl modules used at import and build time (plugins, for example).	2.1
perllib/plugins	Perl code for document processing plugins.	2.1
perllib/classify	Perl code for classifiers (for example the AZList code that makes a document list based on the alphabetical order of some attribute).	2.2
cgi-bin	All Greenstone CGI scripts, which are moved to the system cgi-bin directory.	–
tmp	Directory used by Greenstone for storing temporary files.	–
etc	Configuration files, initialisation and error logs, user authorisation databases.	–
src	C++ code used for serving collections via a web server.	3
src/colservr	C++ code for serving collections—answering queries and the like.	3.7
src/recpt	C++ code for getting queries from the user interface and formatting query responses for the interface.	3.9
packages	Source code for non-Greenstone software packages that are used by Greenstone.	2.5
packages/mg	The source code for MG, the compression and indexing software used by Greenstone.	2.5
mappings	Unicode translation tables (for example for the GB Chinese character set).	–
macros	The macro files used for the user interface.	2.4
collect	Collections being served from this copy of Greenstone	1.1
lib	C++ source code used by both the collection server and the receptionist.	3.1
images	Images used in the user interface.	–
docs	Documentation.	–

You should be able to access the collection from your Greenstone homepage. You will have to reload the page if you already had it open in your browser, or perhaps even close the browser and restart it (to prevent caching problems). To view the new collection, click on the image. The result should look something like Figure 3.

Table 3 Options for the *import* and *build* processes

Option	Argument	Function
–verbosity	Number 0–3	Control how much information about the process will be printed to standard error; 0 gives a little, 3 gives lots.
–archivedir	Directory name	Specify where the archive files (GML files and an information file) are stored—that is, where *import.pl* puts them and where *buildcol.pl* finds them. Defaults to *GSDLHOME/collect/col_name/archives*
–maxdocs	Number >0	Indicates the maximum number of documents to be imported or built. Useful when testing a new collection configuration file, or new plugins.
–collectdir	Directory name	Specify where the collection can be found. Defaults to *GSDLHOME/collect/col_name*
–out	Filename	Specify a file to which to write all output messages, which defaults to standard error (the screen). Useful when working with debugging statements.
–keepold	None	Do not remove the result of the previous import or build operation. In the case of import, do not remove the contents of the *archives* directory; when building, do not remove the content of the *building* directory.
–debug	None	Print plugin output to standard output.

Differences between Windows and Unix

The collection building process under Unix is very similar to that under Windows, but there are some small differences which are summarised in Table 1.

1.2 Greenstone directories

Figure 4 shows the structure of the *GSDLHOME* directory. Table 2 gives a brief description of the contents of each of the directories shown in the diagram. Some directories are better described in a later section of the manual—use the section guide in Table 2 to see where to find more information.

1.3 The import and build processes

In the command-line collection-building process that we walked through in Section 1.1, one command, *import.pl*, is used for importing documents and another, *buildcol.pl,* for building the collection. This section gives more information about what these programs do and the various options they support. We use the variable *col_name* to refer to the collection

Table 4 Additional options for the *import* process

Option	Argument	Function
–importdir	Directory name	Where material to be imported is found. Defaults to *GSDLHOME/collect/col_name/import*.
–removeold	None	Remove the contents of the *archives* directory before importing.
–gzip	None	Use the GNU utility to "zip up" the GML documents produced by import. (This requires ZIPPlug to be included in the plugin list, and *gzip* to be installed on your machine).
–groupsize	Number >0	The number of documents to group together into one GML file. Defaults to 1 (that is, one document per file).
–sortmeta	Metadata tag name	Sort the documents alphabetically by the named metadata tag. However, if the collection has more than one group in the collection (i.e. *groupsize* >1), this functionality is disabled.

being built or imported.

Options accepted by both *import.pl* and *buildcol.pl*

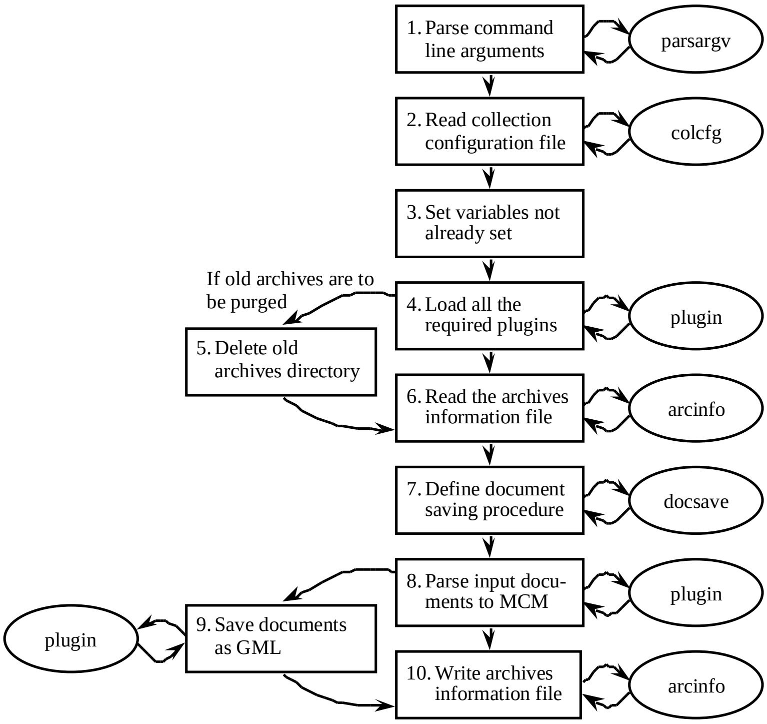
The import and build processes have many similarities, and as a result take many of the same options, described in Table 3. (Remember that to see the options for any Greenstone Perl script you just type its name with no options at the command prompt).

The *import* process

The import process's primary responsibility is to convert documents from their native format into the GML (Greenstone Markup Language) format used within Greenstone, and write a summary file (called *archives.inf*) which will be used when the collection is built. *Import.pl* needs to know what plugins should be used, and where to find the original document files. Table 3 shows the options common to both import and build processes; Table 4 shows additional options applicable to the import process only.

Figure 5 represents the import process implemented by the *import.pl* program. Each oval represents a module used to perform tasks that relate to a specific part of the Greenstone system. All these modules can be found in the *GSDLHOME/perllib* directory.

Figure 5
Steps in the *import* process


1. Parse command line arguments
parsargv
2. Read collection configuration file
colcfg
3. Set variables not already set
If old archives are to be purged
4. Load all the required plugins
plugin
5. Delete old archives directory
6. Read the archives information file
arcinfo
7. Define document saving procedure
docsave
8. Parse input documents to MCM
plugin
plugin
9. Save documents as GML
10. Write archives information file
arcinfo

For step 3, note that import variables like *importdir* and *archivedir* can be set from the collection configuration file or from the command line. If set in the command line, any configuration file setting is ignored.

In step 6, the archives information file (*archives.inf*) is created.

Step 7 creates an object that knows where documents are to be saved, and obeys any special saving instructions (such as *sortmeta*, which sorts the documents according to a specified metadata tag).

Most of the work done in the import process is actually accomplished by plugins, which are called by the *plugin* module. This module creates a pipeline of the plugins specified in the collection configuration file. It also handles the writing of GML documents (using a *document* object).

The *build* process

During the building process the text is compressed, and the full-text indexes that are specified in the collection configuration file are created.

Table 5 Additional options for the *build* process

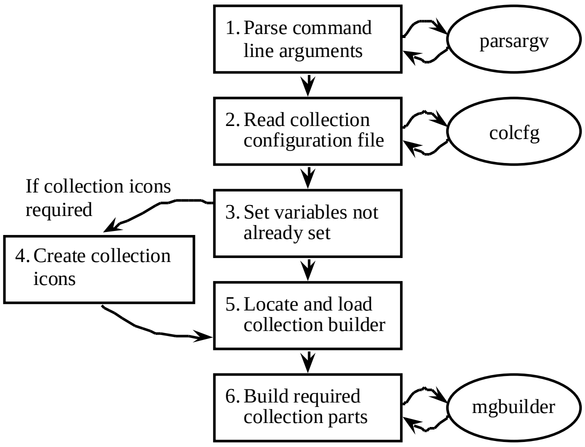
Option	**Argument**	**Function**
–*builddir*	Directory name	Specify where the result of building is to be stored (defaults to *GSDLHOME/collect/col_name/building*).
–*index*	Index name (e.g. *section:Title*)	Specify which indexes to build. This defaults to all the indexes indicated in the collection configuration file.
–*allclassifications*	None	Prevent the build process from removing classifications that include no documents (for example, the "X" classification in titles if there are no documents whose titles start with the letter *X*).
–*create_images*	None	Create collection icons automatically (to use this, GIMP, and the Gimp Perl module, must be installed).
–*mode*	*all*, *compress_text*, *infodb*, or *build_index*	Determine what the build process is to do (defaults to *all*). *All* does a full build, *compress_text* only compresses the document text, *infodb* creates a database of information pertaining to the collection—name, files, associated files, classification information and the like—and *build_index* builds the indexes specified in the collection configuration file or on the command line.

Moreover, all necessary information about how the collection is to appear on the web is precalculated and incorporated into the collection—for example information about icons and titles, and information produced by classifiers. *Buildcol.pl* has many options that it shares with *import.pl*, shown in Table 3, and some that are specific to its purpose, in Table 5.

The diagram in Figure 6 represents the execution of *buildcol.pl*. Many of the steps are common to the import process. The first one that is not is step 4 (to the left). This is performed only if the *create_images* option has been set. Then, the images are created and registered in the collection configuration file by a function in the *buildcol.pl* script. For this to work properly, GIMP (GNU Image Manipulation Program), and the Gimp Perl module, must be installed and properly configured. Also, there must be write (as well as read) access to the collection configuration file.

Step 5 first checks to see whether there is a collection-specific build procedure. A few collections require special build-time processing, in which case a collection-specific builder must be written and placed in the collection's *perllib* directory, named by the collection name with "builder" suffixed. Collection-specific builders are derived from *mgbuilder*. In step 5 the builder (be it the default or a collection-specific one) is initialised with information such as how many documents are to be included, whether or not the old version of the collection is to be

Figure 6
Steps in the *build* process


1. Parse command line arguments
parsargv
2. Read collection configuration file
colcfg
If collection icons required
3. Set variables not already set
4. Create collection icons
5. Locate and load collection builder
6. Build required collection parts
mgbuilder

retained, and where the *building* and *archive* directories are located.

Step 6 is the actual building step, in which the document text is compressed and indexed, collection titles and icons are stored in a collection information database, and data structures are built to support the required classifiers. All these steps are handled by *mgbuilder* (or the collection-specific builder), which in turn uses the MG ("Managing Gigabytes," see Witten *et al.*, 1999) software for compressing and indexing. The parts of the collection that are built can be specified by the *mode* option, but the default is to build everything—compressed text, indexes, and collection information database.

Recall that to make a collection available over the web once it is built, you must move it from the collection's *building* directory to the *index* directory. Collections are not built directly into *index* because large collections may take hours or days to build. It is important that the building process does not affect an existing copy of the collection until the build is complete.

1.4 GML documents

All source documents are brought into the Greenstone system by converting them to a format known as "Greenstone Markup Language" or GML. This format marks documents into sections, and can also be used to store metadata at the document or section level.

The HTML convention of tags enclosed in angle brackets is adopted for markup. Any <, >, or " characters within the text are stored as their HTML equivalent *&lt;*, *&gt;* and *&quot;*.

Table 6 Subtags of <*gsdlsection*>

gsdlsourcefilename	Original file from which the GML was generated
gsdldoctype	Type of document (currently the only recognised type is *indexed_doc*)
gsdlassocfile	File associated with the document (e.g. an image file)
gsdlnum	Subsection number (this does not exist at the document's top level)

Figure 7
Three examples of documents in GML format

(a)

```
<gsdlsection
  gsdlsourcefilename = "uu02fe.txt"
  gsdldoctype = "indexed_doc"
  Identifier="HASHa723e7e164df07c833bfc4"
  Title = "Freshwater Resources in Arid Lands"
  gsdlassocfile="cover.jpg:image/jpeg"
  gsdlassocfile="p21.jpg:image/jpeg"
  gsdlassocfile="p22.jpg:image/jpeg">
   This is the text of the document
</gsdlsection>
```

(b)

```
<gsdlsection
  gsdlsourcefilename = "uu02fe.txt"
  gsdldoctype = "indexed_doc"
  Identifier="HASHa723e7e164df07c833bfc4"
  Title = "Freshwater Resources in Arid Lands">
  <gsdlsection gsdlnum="1" Title="Preface">
  This is the text of the preface
  </gsdlsection>
  <gsdlsection gsdlnum="2" Title="Conclusions">
    <gsdlsection gsdlnum="1" Title="Part 1">
   This is the first part of the conclusions
      </gsdlsection>
    <gsdlsection gsdlnum="2" Title="Part 2">
   This is the second part of the conclusions
      </gsdlsection>
</gsdlsection>
```

(c)

```
<gsdlsection
  gsdlsourcefilename = "uu02fe.txt"
  gsdldoctype = "indexed_doc"
  Identifier="HASHa723e7e164df07c833bfc4"
  Title = "Freshwater Resources in Arid Lands">
  <gsdlsection gsdlnum="1" Title="Chapter 1">
    <gsdlsection gsdlnum="1">
   This is the text of the first page of chapter 1
    </gsdlsection>
    <gsdlsection gsdlnum="2">
   This is the text of the second page of chapter 1
    </gsdlsection>
  </gsdlsection>
  <gsdlsection gsdlnum="2" Title="Chapter 2">
    <gsdlsection gsdlnum="1">
   This is the text of the first and only page of chapter 2
    </gsdlsection>
  </gsdlsection>
</gsdlsection>
```

A <*gsdlsection*> tag denotes the start of each document section, and the corresponding </*gsdlsection*> closing tag marks the end of that section. Each <*gsdlsection*> tag can contain any number of the subtags shown in Table 6. Metadata is indicated within the *gsdlsection* tag (for example, <*gsdlsection Title="Food and Nutrition Bulletin"*>); thus different metadata can be associated with individual sections of a document. Any subtags other than those in Table 6 are considered to be metadata that is attached to that section.

Figure 7a is a GML file that contains a simple document comprising a single section with title, and three associated images. Figure 7b shows a more complex document: a book with two sections called *Preface* and *Conclusions*, the second of which has two subsections. Note that a chapter is simply treated as a top-level section. In some collections documents are split into individual pages. These are treated as sections, though they don't usually have titles. Figure 7c shows a book with two sections (corresponding to chapters), the first of which has two pages and the second one page.

Document metadata

Metadata is descriptive information such as author, title, date, keywords, and so on, that is associated with a document. It has already been mentioned that metadata is stored with documents. Looking at Figure 7, you can see that metadata is stored within <*gsdlsection*> tags in *attribute=value* pairs. One example is the line *Title="Freshwater Resources in Arid Lands"* in Table 2c—the title of a document is a piece of metadata associated with it. The Dublin Core metadata standard[1] is used for defining metadata types. Table 7 shows what types are available in the standard—starred entries are used in collections available from the New Zealand Digital Library web site today. If there is no type that aptly describes a particular kind of metadata, metadata types that are not in the Dublin Core may be used too. For example, the Demo collection contains *how to* and *Magazine* metadata.

Inside GML documents

Within a single document, the GML format imposes a limited amount of structure. Documents are divided into paragraphs. They can be split hierarchically into sections and subsections; these may be nested to any depth. Each document has an associated Object Identifier or OID—these are extended to identify sections and subsections by appending section

[1] For more information on Dublin Core see the web site "The Dublin Core Metadata Initiative" at http://purl.org/dc/, accessed 16 January 2001.

Table 7 The Dublin Core metadata standard

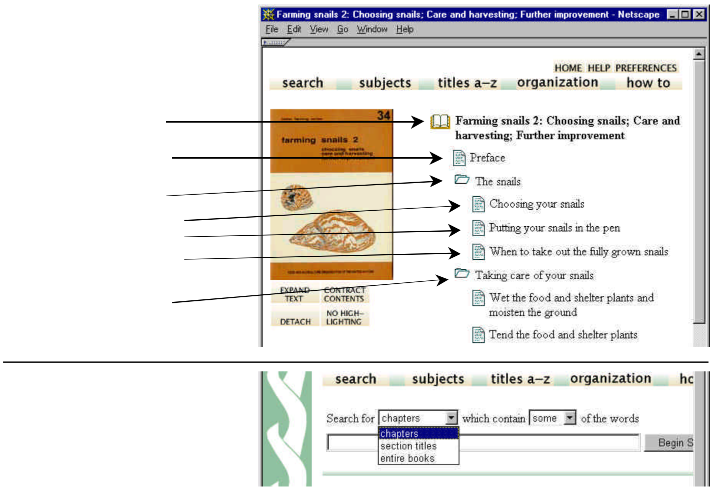
Name	**Metadata subtag**	**Definition**
*Title	*Title*	A name given to the resource
*Creator	*Creator*	An entity primarily responsible for making the content of the resource
*Subject and keywords	*Subject*	The topic of the content of the resource
*Description	*Description*	An account of the content of the resource
*Publisher	*Publisher*	An entity responsible for making the resource available
Contributor	*Contributor*	An entity responsible for making contributions to the content of the resource
*Date	*Date*	The date that the resource was published or some other important date associated with the resource.
Resource type	*Type*	The nature or genre of the content of the resource
Format	*Format*	The physical or digital manifestation of the resource
*Resource identifier	*Identifier*	An unambiguous reference to the resource within a given context: this is the object identifier or OID
*Source	*Source*	A reference to a resource from which the present resource is derived
*Language	*Language*	A language of the intellectual content of the resource
Relation	*Relation*	A reference to a related resource
*Coverage	*Coverage*	The extent or scope of the content of the resource
Rights management	*Rights*	Information about rights held in and over the resource

and subsection numbers, separated by periods, to the document's OID. For example, subsection 3 of section 2 of document HASHa7 is referred to as HASHa7.2.3.

When you read a book in a Greenstone collection, the section hierarchy is manifested in the table of contents of the book. For example, books in the Demo collection have a hierarchical table of contents showing chapters, sections, and subsections, as illustrated in Figure 8a. Documents in the Computer Science Technical Reports collection do not have a hierarchical subsection structure, but each document is split into pages and you can browse around the pages of a retrieved document. Chapters, sections, subsections, and pages are all implemented simply as "sections" within the document.

The document structure is also used for searchable indexes. There are three possible levels of index: *document*, *section*, and *paragraph*, though most collections will not use all three levels. A *document* index contains the full document—you use it to find all documents that contain a particular set of words (the words may be scattered far and wide

Figure 8 Hierarchical structure in the Demo collection



Farming snails 2: Choosing snails; Care and harvesting; Further improvement - Netscape
File Edit View Go Window Help
HOME HELP PREFERENCES
search subjects titles a–z organization how to
34
farming snails 2
EXPAND TEXT
CONTRACT CONTENTS
DETACH
NO HIGH-LIGHTING

Farming snails 2: Choosing snails; Care and harvesting; Further improvement
Preface
The snails
Choosing your snails
Putting your snails in the pen
When to take out the fully grown snails
Taking care of your snails
Wet the food and shelter plants and moisten the ground
Tend the food and shelter plants

search subjects titles a–z organization hc
Begin S

Search for chapters which contain some of the words
chapters
section titles
entire books

throughout the document). When a *section* index is created, each portion of text that is indexed stretches from a *gsdlsection* tag to the next-occurring *gsdlsection* tag—thus a chapter that immediately begins with a new section will produce an empty document in the index. Sections and subsections are treated alike: the hierarchical document structure is flattened for the purposes of creating searchable indexes.

The pulldown menu in Figure 8b shows the searchable indexes for the Demo collection. “Chapters” and “section titles” are section-level indexes, while “entire books” is a document-level index. As well as indexes of text, indexes of any kind of metadata can also be created. For example, some collections offer searchable indexes of section titles, and Figure 8b illustrates this.

1.5 The collection configuration file

The collection configuration file governs the structure of a collection as seen by the user, allowing you to customise the “look and feel” of your collection and the way in which its documents are processed and presented. A simple collection configuration file is created when you run *mkcol.pl*, which lists your name and E-mail address as the creator and maintainer. Remember from the walkthrough that the *creator* argument is required—this is where the E-mail address comes from.

Each line of the collection configuration file is essentially an “attribute, value” pair. Each attribute gives a piece of information about the collection that affects how it is supposed to look or how documents are to be processed. Table 8 shows the items that can be included in a collection

Table 8 Items in the collection configuration file

creator	E-mail address of the collection’s creator
maintainer	E-mail address of the collection’s maintainer
public	Whether collection is to be made public or not
beta	Whether collection is beta version or not
indexes	List of indexes to build
defaultindex	The default index
subcollection	Define a subcollection based on metadata
indexsubcollections	Specify which subcollections to index
defaultsubcollection	The default indexsubcollection
languages	List of languages to build indexes in
defaultlanguage	Default index language
collectionmeta	Defines collection-level metadata
plugin	Specify a plugin to use at build time
format	A format string (explained below)
classify	Specify a classifier to use at build time

configuration file, and what each is used for.

The collection configuration file created by the *mkcol.pl* script, shown in Table 9, is a very simple one and contains a bare minimum of information. Lines 1 and 2 stem from the *creator* option supplied to the *mkcol.pl* program, and contain the E-mail addresses of the person who created the collection, and that of the person responsible for maintaining it (not necessarily the same person).

Line 3 indicates whether the collection will be available to the public when it is built, and is either *true* (the default, meaning the collection will be publicly available), or *false* (meaning that it will not). This is useful when building collections to test software, or building collections of material for personal use. Line 4 indicates whether the collection is beta or not (this also defaults to *true*, meaning that the collection is a beta release).

Line 5 determines what collection indexes are created at build time: In this example only the document text is to be indexed. Indexes can be constructed at the *document*, *section*, and *paragraph* levels. They can contain the material in *text*, or in any metadata—most commonly *Title*. The form used to specify an index is *level:data*. For example, to include an index of section titles as well, you should change line 5 to *indexes document:text section:Title*. More than one type of data can be included in the same index by separating the data types with commas. For example, to create a section-level index of titles, text and dates, the line should read *indexes section:text,Title,Date.* The default index defined in line 6 is the default to be used on the collection’s search page.

Table 9 Collection configuration file created by *mkcd.pl*

	Attribute	Value
1	creator	username@email.com
2	maintainer	username@email.com
3	public	True
4	beta	True
5	indexes	document:text
6	defaultindex	document:text
7	plugin	ZIPPlug
8	plugin	GMLPlug
9	plugin	TextPlug
10	plugin	HTMLPlug
11	plugin	EMAILPlug
12	plugin	ArcPlug
13	plugin	RecPlug
14	classify	AZList metadata Title
15	collectionmeta	collectionname "collect"
16	collectionmeta	iconcollection ""
17	collectionmeta	collectionextra ""
18	collectionmeta	.document:text "documents"

Lines 7–13 specify which plugins to use when converting documents to GML format and when building collections from GML files. Section 2.1 gives information about what plugins are available. The order in which plugins are listed is the order in which they are tried on each document, and once a plugin that is able to process a document is found, no more are tried.

Line 14 specifies that an alphabetic list of titles is to be created for browsing purposes. Browsing structures are constructed by “classifiers”. Section 2.2 gives information about classifiers and what they can do.

Lines 15–18 are used to specify collection-level metadata. The long form of the name is used as the collection’s “title” for the web browser. The *collectionicon* entry gives the URL of the collection’s icon. If an index is specified (as in line 18), the string following is displayed as the name of that index on the collection’s search page. A particularly important piece of collection-level metadata is *collectionextra*, which gives a stretch of text, surrounded by double quotes, describing the collection. This will be shown as the “About this collection” text.

This simple collection configuration file does not include any examples of format strings, nor of the subcollection and language facilities provided by the configuration file. Format strings are covered more thoroughly in Section 2.3, but we will look at subcollections and languages here.

Subcollections

Greenstone allows you to define subcollections and build separate indexes for each one. For example, in one collection there is a large subset of documents called *Food and Nutrition Bulletin*. We use this collection as an example.

This collection has three indexes, all at the section level: one for the whole collection, one for the *Food and Nutrition Bulletin*, and the third for the remaining documents. The relevant lines from the collection configuration file can be seen below.

```
indexes            section:text
subcollection      fn “Title/^Food and Nutrition Bulletin/i”
subcollection      other “!Title/^Food and Nutrition Bulletin/i”
indexsubcollections fn  other  fn,other
```

The second and third lines define subcollections called *fn*, which contains the *Food and Nutrition Bulletin* documents, and *other*, which contains the remaining documents. The third field of these definitions is a Perl regular expression that identifies these subsets using the *Title* metadata: we seek titles that begin with *Food and Nutrition Bulletin* in the first case and ones that do not in the second case (note the “!”). The final *i* makes the pattern-matching case-insensitive. The metadata field, in this case *Title*, can be any valid field, or *Filename* to match against the document’s original filename. The fourth line, *indexsubcollections*, specifies three indexes: one for the *fn* subcollection, one for the *other* subcollection, and the third for both subcollections (i.e. all the documents). Note that if two entries had been specified on the *indexes* line, the total number of indexes generated would have been six rather than three.

If a collection contains documents in different languages, separate indexes can be built for each language. Language is a metadata statement; values are specified using the ISO 639 standard two-letter codes for representing the names of languages—for example, *en* is English, *zh* is Chinese, and *mi* is Maori. Since metadata values can be specified at the section level, parts of a document can be in different languages.

For example, if the configuration file contained

```
indexes section:text section:Title document:text paragraph:text
languages en zh mi
```

section text, section title, document text, and paragraph text indexes would be created for English, Chinese, and Maori—twelve indexes altogether. Adding a couple of subcollections multiplies the number of indexes again. Care is necessary to guard against index bloat.

(This index specification could be defined using the *subcollection* facility rather than the *languages* facility. However, since the current syntax precludes creating subcollections of subcollections, it would then be impossible to index each language in the subcollections separately.)

2 Getting the most out of your documents

Collections can be individualised to make the information they contain accessible in different ways. This chapter describes how Greenstone extracts information from documents and presents it to the user: the document processing (section 2.1) and classification structures (section 2.2), and user interface tools (sections 2.3 and 2.4).

2.1 Plugins

Plugins parse the imported documents and extract metadata from them. For example, the HTML plugin converts HTML pages to GML and extracts metadata which is explicit in the document format—such as titles, enclosed by *<title></title>* tags.

All plugins are written in the Perl language. All derive from a basic plugin called *BasPlug*, which performs universally-required operations like creating a new GML document to work with, assigning an object identifier (OID), and handling the sections in a document. Plugins are kept in the *perllib/plugins* directory.

To find more about any plugin, just type *pluginfo.pl plugin-name* at the command prompt. (You will need to invoke the appropriate *setup* script first, if you haven't already). This displays information about the plugin on the screen—what plugin-specific options it takes, and what general options are allowed.

You can easily write new plugins that process document formats not handled by existing plugins, format documents in some special way, or extract a new kind of metadata.

Table 10 Options applicable to all plugins

input_encoding	Character encoding of the source documents. The default is to automatically work out the character encoding of each individual document. It is sometimes useful to set this value though, for example, if you know that all your documents are plain ASCII, setting the input encoding to 'ascii' will greatly increase the speed at which your collection is imported and built. There are many possible values. Use *pluginfo.pl BasPlug* to get a complete list.
default_encoding	The encoding that will be used if *input_encoding* is 'auto' and automatic encoding detection fails.
process_exp	A Perl regular expression to match against filenames (for example, to locate a certain kind of file extension). This dictates which files a plugin will process. Each plugin has a default (*HTMLPlug*'s default is *(?i).html?*—that is, anything with the extension *.htm* or *.html*).
block_exp	A regular expression to match against filenames that are not to be passed on to subsequent plugins. This can prevent annoying error messages about files you aren't interested in. Some plugins have default blocking expressions—for example, *HTMLPlug* blocks files with *.gif, .jpg, .jpeg, .png, .rtf* and *.css* extensions.
extract_acronyms	Extract acronyms from documents and add them as metadata to the corresponding GML documents.
markup_acronyms	Add acronym information into document text.
extract_language	Identify each document's language and associate it as metadata. Note that this will be done automatically if *input_encoding* is 'auto'.
default_language	If automatic language extraction fails, language metadata will be set to this value.
first	Extract a comma-separated list of the first stretch of text and add it as *FirstNNN* metadata (often used as a substitute for *Title*).
extract_email	Extract E-mail addresses and add them as document metadata.
extract_date	Extract dates relating to the content of historical documents and add them as *Coverage* metadata.

General Options

Table 10 shows options that are accepted by any plugin derived from *BasPlug*.

Document processing plugins

Document processing plugins are used by the collection-building software to parse each source document in a way that depends on its format. A collection's configuration file lists all plugins that are used when building it. During the import operation, each file or directory is passed to each

Table 11 Greenstone plugins

	Plugin	Purpose	File types	Ignores files
General	*ArcPlug*	Processes files named in the file *archives.inf*, which is used to communicate between the import and build processes. Must be included (unless *import.pl* will not be used).	—	—
	RecPlug	Recurses through a directory structure by checking to see whether a filename is a directory and if so, inserting all files in the directory into the plugin pipeline.	—	—
	IndexPlug	Assigns metadata from a (manually-created) *index.txt* spreadsheet file used mainly by UN collections.	—	—
	GMLPlug	Processes GML files generated by *import.pl.* Must be included (unless *import.pl* will not be used).	*.gml, .gm*	—
	TEXTPlug	Processes plain text by placing it between *<pre> </pre>* tags (treating it as preformatted).	*.txt, .text*	—
	HTMLPlug	Processes HTML, replacing hyperlinks appropriately. If the linked document is not in the collection, an intermediate page is inserted warning the user they are leaving the collection. Extracts readily available metadata such as *Title*.	*.htm, .html, .cgi, .php, .asp, .shm, .shtml*	*.gif, .jpeg, .jpg, .png, .css, .rtf*
	EMAILPlug	Processes E-mail messages, recognising author, subject, date, etc. This plugin does not yet handle MIME-encoded E-mails properly—although legible, they often look rather strange.	Must end in digits or digits followed by *.Email*	—
	WordPlug	Processes Microsoft Word documents, extracting author and title where available, and keeping diagrams and pictures in their proper places. The conversion utilities used by this plugin often produce HTML that is poorly formatted, and we recommend that you provide the original documents for viewing when building collections of WORD files. However, the text that is extracted from the documents is adequate for searching and indexing purposes.	*.doc*	*.gif, .jpeg, .jpg, .png, .css, .rtf*

Table 11 Greenstone plugins (continued)

	Plugin	Purpose	File types	Ignores files
	PDFPlug	Processes PDF documents, extracting the first line of text as a title. The *pdftohtml* program fails on some PDF files. What happens is that the conversion process takes an exceptionally long time, and often an error message relating to the conversion process appears on the screen. If this occurs, the only solution that we can offer is to remove the offending document from the collection and re-import.	*.pdf*	*.gif, .jpeg, .jpg, .png, .css, .rtf*
	PSPlug	Processes PostScript documents, optionally extracting date, title and page number metadata. This plugin does not work on Windows systems because it relies on *ps2ascii,* a Unix utility.	*.ps*	*.eps*
	HBPlug	Processes HTML marked up for UN collections	—	—
	FOXPlug	Processes FoxBASE dbt files	*.dbt, .dbf*	—
	ZIPPlug	Uncompresses *gzip*, *bzip*, *zip*, and *tar* files, provided the appropriate GNU tools are available.	*.gzip, .bzip, .zip, .tar, .gz, .bz, .tgz, .taz*	—
Collection Specific	*PrePlug*	Processes HTML output using PRESCRIPT, splitting documents into pages for the Computer Science Technical Reports collection.	*.html, .html.gz*	—
	GBPlug	Processes Project Gutenberg etext—which includes manually-entered title information.	*.txt.gz, .html, .htm*	—
	TCCPlug	Processes E-mail documents from Computists' Weekly	Must begin with *tcc or cw*	—

plugin in turn until one is found that can process it—thus earlier plugins take priority over later ones. If no plugin can process the file, a warning is printed (to standard error) and processing passes to the next file. (This is where the *block_exp* option can be useful—to prevent these error messages for files that might be present but don't need processing.) During building, the same procedure is used, but the *archives* directory is processed instead of the *import* directory.

The standard Greenstone plugins are listed in Table 11. Recursion is necessary to traverse directory hierarchies. Although the import (and build) program does not perform explicit recursion, some plugins cause indirect recursion. A plugin may itself pass files or directory names into the plugin pipeline. For example, the standard way of recursing through a hierarchical directory structure is to specify *RecPlug*, which does exactly this. If present, it should be the last element in the pipeline. Only the first three plugins in Table 11 are recursive.

Some plugins are written specifically for particular collections. For example, the collection may have a document format not found elsewhere, like the e-text used in the Gutenberg collection. These plugins are found in the collection's *perllib/plugins* directory. If a collection-specific plugin has the same name as a general one, the collection-specific version will be used.

Some document-processing plugins use external programs that parse specific proprietary formats—for example, Microsoft Word—into either plain text or HTML. A general plugin called *ConvertToPlug* invokes the appropriate conversion program and passes the result to either *TEXTPlug* or *HTMLPlug*. We describe this in more detail shortly.

Plugin-specific options

Some plugins have individual options, which control what they do in finer detail than the general options allow. Table 12 describes them.

Plugins to import proprietary formats

Proprietary formats pose difficult problems for any digital library system.

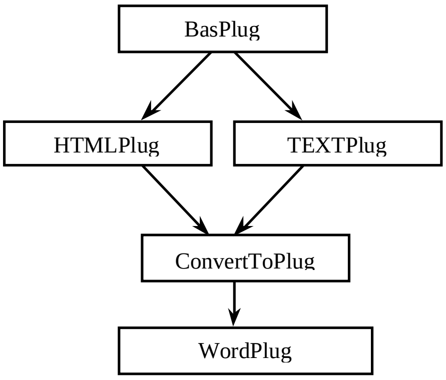
Although documentation may be available about how they work, they are subject to change without notice, and it is difficult to keep up with changes. Greenstone has adopted the policy of using GPL (GNU Public License) conversion utilities written by people dedicated to the task. Utilities to convert Word and PDF formats are included in the *packages* directory. These all convert documents to either text or HTML. Then *HTMLPlug* and *TEXTPlug* are used to further convert them to the GML format. *ConvertToPlug* is used to include the conversion utilities. Like *BasPlug* it is never called directly. Rather, plugins written for individual formats are derived from it as illustrated in Figure 9. *ConvertToPlug* uses Perl's dynamic inheritance scheme to inherit from either *TEXTPlug* or *HTMLPlug*, depending on the format to which a source document has been converted.

Table 12 Plugin-specific options

Plugin	Option	Purpose
HTMLPlug	*nolinks*	Do not attempt to trap links within the collection. This speeds up the import/build process, but any links in the collection will be broken.
	keep_head	Do not strip out HTML headers.
	no_metadata	Do not attempt to find any metadata (this may speed up the import/build process).
	metadata_fields	Takes a comma-separated list of metadata types (defaults to *Title*) to extract. To rename the metadata in the GML file, use *tag<newname>* where *tag* is the HTML tag sought and *newname* its new name.
	hunt_creator_metadata	Find as much metadata as possible about authorship and put it in the GML document as *Creator* metadata. You also need to include *Creator* using the *metadata_fields* option.
	w3mir	Use this option if the *w3mir* program has been used to create the structure of the documents to be imported.
	assoc_files	Gives a Perl regular expression that describes file types to be treated as associated files. The default types are *.jpg, .jpeg, .gif, .png, .css*
	rename_assoc_files	Rename files associated with documents. During this process the directory structure of any associated files will become much shallower (useful if a collection must be stored in limited space).
	title_sub	Perl substitution expression to modify titles.
PSPlug	*extract_date*	Extract the creation date from the PostScript header and store it as metadata.
	extract_title	Extract the document title from the PostScript header and store it as title metadata.
	extract_pages	Extract the page numbers from the PostScript document and add them to the appropriate sections as metadata with the tag *Pages*

When *ConvertToPlug* receives a document, it calls *gsConvert.pl* (found in *GSDLHOME/bin/script*) to invoke the required conversion utility. Once the document has been converted, it is returned to *ConvertToPlug,* which invokes the text or HTML plugin as appropriate. Any plugin derived from *ConvertToPlug* has an option *convert_to*, whose argument is either *text* or

Figure 9 Plugin inheritance hierarchy



html, to specify which intermediate format is preferred. Text is faster, but HTML is better for pictures and suchlike.

Sometimes there are several conversion utilities for a particular format, and *gsConvert* may invoke more than one for any given document. For example, the preferred Word conversion utility *wv* does not cope with anything less than Word 6, and a program called *AnyToHTML*, which essentially just extracts whatever text strings can be found, is called to convert Word 5 documents.

The steps involved in adding a new external document conversion utility are:

1. Install the new conversion utility so that it is accessible by Greenstone (we recommend you put it in the *packages* directory).

2. Alter *gsConvert.pl* to use the new conversion utility. This involves adding a new clause to the *if* statement in the *main* function, and adding a function that calls the conversion utility.

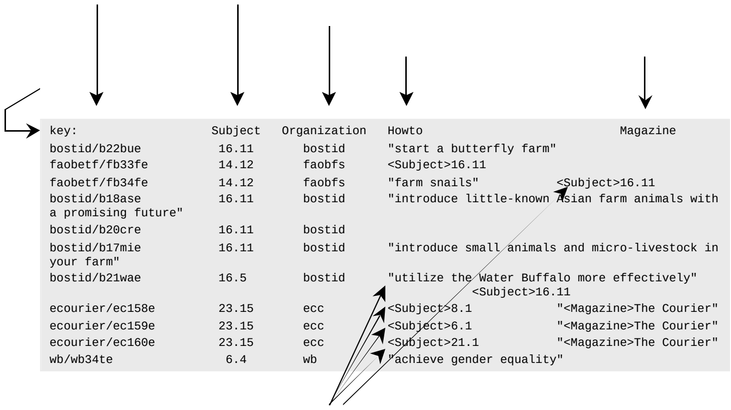
3. Write a top-level plugin that inherits from *ConvertToPlug* to catch the format and pass it on.

Assigning metadata from a file

Using *IndexPlug*, metadata can be assigned to documents from a manually created spreadsheet-like file. We describe this in some detail, so that you can create metadata “index” files in the appropriate format, or alter *IndexPlug* to use a file format of your choice.

IndexPlug checks to see if *index.txt* exists, and if so, reads the list of files,

Figure 10
The *index.text* metadata file, used by *IndexPlug*



```
key:                    Subject   Organization   Howto                          Magazine
bostid/b22bue            16.11       bostid      "start a butterfly farm"
faobetf/fb33fe           14.12       faobfs      <Subject>16.11
faobetf/fb34fe           14.12       faobfs      "farm snails"           <Subject>16.11
bostid/b18ase            16.11       bostid      "introduce little-known Asian farm animals with
a promising future"
bostid/b20cre            16.11       bostid
bostid/b17mie            16.11       bostid      "introduce small animals and micro-livestock in
your farm"
bostid/b21wae            16.5        bostid      "utilize the Water Buffalo more effectively"
                                                             <Subject>16.11
ecourier/ec158e          23.15       ecc         <Subject>8.1            "<Magazine>The Courier"
ecourier/ec159e          23.15       ecc         <Subject>6.1            "<Magazine>The Courier"
ecourier/ec160e          23.15       ecc         <Subject>21.1           "<Magazine>The Courier"
wb/wb34te                 6.4        wb          "achieve gender equality"
```

assigns metadata to them as specified, and passes each one in turn to the plugin pipeline. The *index.txt* file associated with UN collections contains extra metadata that is associated with each book. This includes the Subject classification, Organisation, “how to” classification, and Magazine title. The information was generated manually from spreadsheet files supplied with the collection, by far the most time-consuming part of making the collection.

An annotated excerpt of the *index.txt* file appears in Figure 10. The first line is a key to what metadata fields are included, in this case *Subject*, *Organisation*, *Howto*, and *Magazine* metadata. Subsequent lines give the filename of a document followed by its metadata values. For example, the second line of the example in Figure 10 assigns to the document that originated in file *bostid/b22bue* the Subject *16.11*, the Organisation *bostid*, the “how to” *start a butterfly farm*, and nothing for *Magazine*. A mechanism is provided to allow for multiple field values and missing fields, by placing field names in angle brackets. The next line not only assigns 14.12 to Subject and *faobfs* to Organisation but also assigns 16.11 to Subject as well; it makes no assignment to “how to”. All metadata specifications may be repeated in a document file. This combines a mechanism to utilise metadata provided in spreadsheet files with a way of allowing more flexible non-column-oriented specifications.

2.2 Classifiers

Classifiers are used to create a collection’s browsing indexes. Examples are the *dlpeople* collection’s *Titles A-Z* index, and the *Subject*, *How to*, *Organisation* and *Titles A-Z* indexes in the Humanity Development

Library—of which the Demo collection is a subset. The navigation bar near the top of the screenshots in Figures 3 and 8a include the *search* function, which is always provided, followed by buttons for any classifiers that have been defined. The information used to support browsing is stored in the collection information database, and is placed there by classifiers which are called during the final phase of *buildcol.pl*.

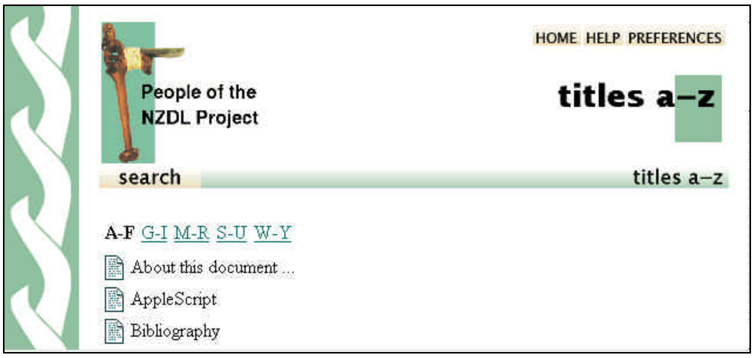
Classifiers, like plugins, are specified in a collection's configuration file. For each one there is a line starting with the keyword *classify* and followed by the name of the classifier and any options it takes. The basic collection configuration file discussed in Section 1.5 includes the line *classify AZList –metadata Title*, which makes an alphabetic list of titles by taking all those with a *Title* metadata field, sorting them and splitting them into alphabetic ranges. An example is shown in Figure 11.

A simpler classifier, called *List*, illustrated in Figure 12, creates a sorted list of a given metadata element and displays it without any alphabetic subsections. An example is the *how to* metadata in the Demo collection, which is produced by a line *classify List –metadata Howto* in the collection configuration file. Another general-purpose list classifier is *DateList*, illustrated in Figure 13, which generates a selection list of date ranges. (The *DateList* classifier is also used in the Greenstone Archives collection.)

Other classifiers generate browsing structures that are explicitly hierarchical. Hierarchical classifications are useful for subject classifications and subclassifications, and organisational hierarchies. The Demo collection's configuration file contains the line *classify Hierarchy –hfile sub.txt –metadata Subject –sort Title*, and Figure 14 shows the subject hierarchy browser that it produces. The bookshelf with a bold title is the one currently being perused; above it you can see the subject classification to which it belongs.

All classifiers generate a hierarchical structure that is used to display a browsing index. The leaves (i.e. lowest levels) of the hierarchy are usually documents, but in some classifiers they are sections. The internal nodes of the hierarchy are either *Vlist*, *Hlist*, or *Datelist*. A *Vlist* is a list of items displayed vertically down the page, like the "how to" index in the Demo collection (see Figure 12). An *Hlist* is displayed horizontally. For example, the *AZList* display in Figure 11 is a two-level hierarchy of internal nodes consisting of an *Hlist* (giving the A-Z selector) whose children are *Vlists*—and their children, in turn, are documents. A *Datelist* (Figure 13) is a special kind of *Vlist* that allows selection by year and month.

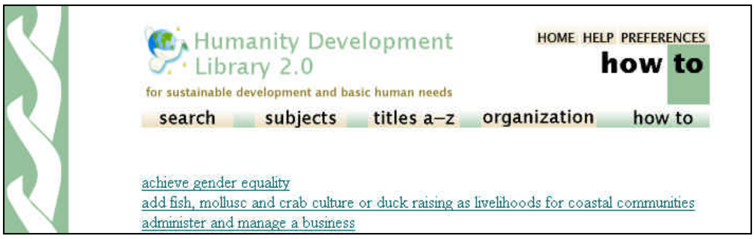
Figure 11
Using the *AZList* classifier



A-F G-I M-R S-U W-Y

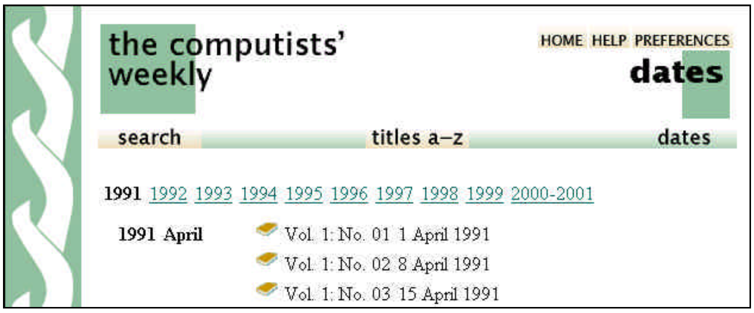
- About this document ...
- AppleScript
- Bibliography

Figure 12
Using the *List* classifier


Humanity Development Library 2.0
for sustainable development and basic human needs
HOME HELP PREFERENCES
how to
search
subjects
titles a–z
organization
how to

achieve gender equality
add fish, mollusc and crab culture or duck raising as livelihoods for coastal communities
administer and manage a business

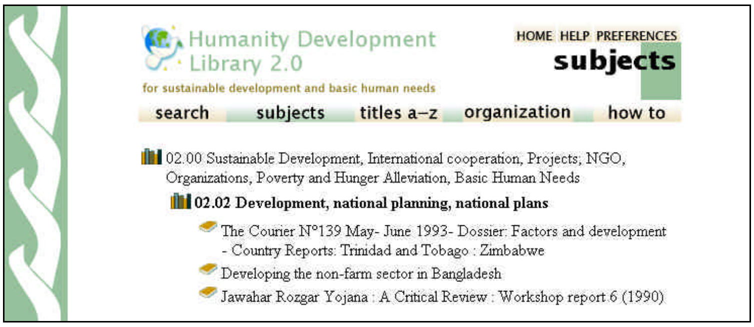
Figure 13
Using the *DateList* Classifier



1991 1992 1993 1994 1995 1996 1997 1998 1999 2000-2001

1991 April
- Vol. 1: No. 01 1 April 1991
- Vol. 1: No. 02 8 April 1991
- Vol. 1: No. 03 15 April 1991

Figure 14
Using the *Hierarchy* classifier


Humanity Development Library 2.0
for sustainable development and basic human needs
HOME HELP PREFERENCES
subjects
search
subjects
titles a–z
organization
how to

02.00 Sustainable Development, International cooperation, Projects; NGO, Organizations, Poverty and Hunger Alleviation, Basic Human Needs
- **02.02 Development, national planning, national plans**
  - The Courier N°139 May- June 1993- Dossier: Factors and development - Country Reports: Trinidad and Tobago : Zimbabwe
  - Developing the non-farm sector in Bangladesh
  - Jawahar Rozgar Yojana : A Critical Review : Workshop report 6 (1990)

Table 13 Greenstone classifiers

Classifier	Argument	Purpose
Hierarchy		Hierarchical classification
	hfile	Classification file
	metadata	Metadata element to test against *hfile* identifier
	sort	Metadata element to sort documents by (defaults to *Title*)
	buttonname	Name of the button used to access this classifier (defaults to value of metadata argument)
List		A list of documents
	metadata	Include documents containing this metadata element
	buttonname	Name of button used to access this classifier (defaults to value of metadata argument)
SectionList		List of sections in documents
AZList		List of documents split into alphabetical ranges
	metadata	Include all documents containing this metadata element
	buttonname	Name of button used to access this classifier (defaults to value of metadata argument)
AZSectionList		Like *AZList* but includes every section of the document
DateList		Similar to *AZList* but sorted by date

The lines used to specify classifiers in collection configuration files contain a *metadata* argument that identifies the metadata by which the documents are classified and sorted. Any document in the collection that does not have this metadata defined will be omitted from the classifier (but it will still be indexed and searchable). If no *metadata* argument is specified, all documents will be included in the classifier, in the order in which they are encountered during the building process. This is useful if you want a list of all documents in your collection. The current set of classifiers is listed in Table 13. Just as you can use the *pluginfo.pl* program to find out about any plugin, there is a *classinfo.pl* program that will give you information about any classifier, and the options it provides.

All classifiers accept the argument *buttonname*, which defines what is written on the Greenstone navigation button that invokes the classifier (it defaults to the name of the metadata argument). Buttons are provided for each Dublin Core metadata type, and for some other types of metadata.

Each classifier receives an implicit name from its position in the configuration file. For example, the third classifier specified in the file is called CL3. This is used to name the collection information database fields that define the classifier hierarchy.

Collection-specific classifiers can be written, and are stored in the

Figure 15
Part of *sub.tex*, the hierarchy specification file for the HDL collection

```
1      1      "General reference"
1.2    1.2    "Dictionaries, glossaries, language courses, terminology
2      2      "Sustainable Development, International cooperation, Pro
2.1    2.1    "Development policy and theory, international cooperatio
2.2    2.2    "Development, national planning, national plans"
2.3    2.3    "Project planning and evaluation (incl. project managem
2.4    2.4    "Regional development and planning incl. regional profil
2.5    2.5    "Nongovernmental organisations (NGOs) in general, self-
2.6    2.6    "Organisations, institutions, United Nations (general, d
2.6.1  2.6.1  "United Nations"
2.6.2  2.6.2  "International organisations"
2.6.3  2.6.3  "Regional organisations"
2.6.5  2.6.5  "European Community - European Union"
2.7    2.7    "Sustainable Development, Development models and example
2.8    2.8    "Basic Human Needs"
2.9    2.9    "Hunger and Poverty Alleviation"
```

collection's *perllib/classify* directory. The Humanity Development Library (and consequently the Demo collection, which is a subset of it) has a collection-specific classifier called *HDLList*, which is a minor variant of *AZList*.

List classifiers

The various flavours of list classifier are shown below.

- *SectionList*—like *List* but the leaves are sections rather than documents. All document sections are included except the top level. This is used to create lists of sections (articles, chapters or whatever) such as in the Computists' Weekly collection, where each issue is a single document and comprises several independent news items, each in its own section.
- *AZList*—generates a two-level hierarchy comprising an *HList* whose children are *VLists*, whose children are documents. The *HList* is an A-Z selector that divides the documents into alphabetic ranges. Documents are sorted alphabetically by metadata, and the resulting list is split into ranges.
- *AZSectionList*—like *AZList* but the leaves are sections rather than documents.
- *DateList*—like *AZList* except that the top-level *HList* allows selection by year and its children are *DateLists* rather than *VLists*. The metadata argument defaults to *Date*.

The *hierarchy* classifier

All classifiers are hierarchical. However, the list classifiers described above have a fixed number of levels, whereas the "hierarchy" classifiers described in this section have an arbitrary number of levels. Hierarchy classifiers are more complex to specify than list classifiers.

The *hfile* argument gives the name of a file, like that in Figure 15, which defines the metadata hierarchy. Each line describes one classification, and the descriptions have three parts:

- Identifier, which matches the value of the metadata (given by the *metadata* argument) to the classification.
- Position-in-hierarchy marker, in multi-part numeric form, e.g. 2, 2.12, 2.12.6.
- The name of the classification. If this contains spaces, it should be surrounded by quotation marks.

Figure 15 is part of the *sub.txt* file used to create the subject hierarchy in the Humanity Development Library (and the Demo collection). This example is a slightly confusing one because the number representing the hierarchy appears twice on each line. The metadata type *Hierarchy* is represented in documents with values in hierarchical numeric form, which accounts for the first occurrence. It is the second occurrence that is used to determine the hierarchy that the hierarchy browser implements.

The *hierarchy* classifier has an optional argument, *sort*, which determines how the documents at the leaves are ordered. Any metadata can be specified as the sort key. The default is to produce the list in the order in which the building process encounters the documents. Ordering at internal nodes is determined by the order in which things are specified in the *hfile* argument.

How classifiers work

Classifiers are Perl objects, derived from *BasClas.pm*, and are stored in the *perllib/classify* directory. They are used when the collection is built, and their execution involves four steps:

1. The *new* method creates the classifier object.
2. The *init* method initialises the object with parameters such as metadata type, button name and sort criterion.
3. The *classify* method is invoked once for each document, and stores information about the classification made within the classifier object.
4. The *get_classify_info* method returns the locally stored classification information to the build process, which writes it to the collection information database for use when the collection is displayed.

The *classify* method retrieves each document's OID, the metadata value on which the document is to be classified, and, where necessary, the metadata value on which the documents are to be sorted. The

Table 14 The format options

format DocumentImages true/false	If *true*, display a cover image at the top left of the document page (default *false*).
format DocumentHeading formatstring	If *DocumentImages* is *false*, the format string controls how the document header shown at the top left of the document page looks (default *[Title]*).
format DocumentContents true/false	Display table of contents (if document is hierarchical), or next/previous section arrows and "page k of n" text (if not).
format DocumentButtons string	Controls the buttons that are displayed on a document page (default *Detach*|*Highlight*).
format DocumentText formatstring	Format of the text to be displayed on a document page (default *[Text]*).
format DocumentArrowsBottom true/false	Display next/previous section arrows at bottom of document page (default *true*).

get_classify_info method performs all sorting and classifier-specific processing (for example, in the case of the *AZList* classifier, it splits the list into ranges).

The build process initialises the classifiers as soon as the *builder* object is created (see Section 1.4). Classifications are created during the build phase, when the information database is created, by *classify.pm*, which resides in Greenstone's *perllib* directory.

2.3 Formatting Greenstone output

The web pages you see when using Greenstone are not pre-stored but are generated "on the fly" as they are needed. The appearance of many aspects of the pages is controlled using "format strings." Format strings belong in the collection configuration file, introduced by the keyword *format* followed by the name of the element to which the format applies. There are two different kinds of page element that are controlled by format strings. The first comprises the items on the page that show documents or parts of documents. The second comprises the lists produced by classifiers and searches. All format strings are interpreted at the time that pages are displayed. Since they take effect as soon as any changes in *collect.cfg* are saved, experimenting with format strings is quick and easy.

Table 14 shows the format statements that affect the way documents look. The *DocumentButtons* option controls what buttons are displayed on a document page. Here, *string* is a list of buttons (separated by |), possible values being *Detach*, *Highlight*, *Expand Text*, and *Expand Contents*. Reordering the list reorders the buttons.

Table 15 Items that may appear in format strings

[Text]	The document's text
[link] … [/link]	The HTML to link to the document itself
[icon]	An appropriate icon (e.g. the little text icon in a *Search Results* string)
[num]	The document result number (useful for debugging).

Formatting Greenstone lists

Format strings that control how lists look can apply at different levels of the display structure. They can alter all lists of a certain type within a collection (for example *DateList*), or all parts of a list (for example all the entries in the *Search* list), or specific parts of a certain list (for example, the vertical list part of an *AZList* classifier on title).

Following the keyword *format* is a two-part keyword, only one part of which is mandatory. The first part identifies the list to which the format applies. The list generated by a search is called *Search*, while the lists generated by classifiers are called *CL1, CL2, CL3,…* for the first, second, third,… classifier specified in *collect.cfg*. The second part of the keyword is the part of the list to which the formatting is to apply—either *HList* (for horizontal list, like the A-Z selector in an *AZList*), *VList* (for vertical list, like the list of titles under an *AZList*), or *DateList*. For example:

format CL4VList ...	applies to all *VLists* in CL4
format CL2HList ...	applies to all *HLists* in CL2
format CL1DateList ...	applies to all *DateLists* in CL1
format SearchVList ...	applies to the Search Results list
format CL3 ...	applies to all nodes in CL3, unless otherwise specified
format VList ...	applies to all *VLists* in all classifiers, unless otherwise specified

The "..." in these examples stand for HTML format specifications that control the information, and its layout, that appear on web pages displaying the classifier. As well as HTML specifications, any metadata may appear within square brackets: its value is interpolated in the indicated place. Also, any of the items in Table 15 may appear in format strings. The syntax for the strings also includes a conditional statement, which is illustrated in an example below.

Recall that all classifiers produce hierarchies. Each level of the hierarchy is displayed in one of four possible ways. We have already encountered *HList*, *VList*, and *DateList*. There is also *Invisible*, which is how the very top levels of hierarchies are displayed—because the name of the classifier is already shown separately on the Greenstone navigation bar.

Figure 16
Excerpt from the Demo collection's *collect.cfg*

```
 1 classify Hierarchy -hfile sub.txt -metadata Subject -sort Title
 2 classify AZList    -metadata Title
 3 classify Hierarchy -hfile org.txt -metadata Organisation -sort Title
 4 classify List      -metadata Howto
 5 format SearchVList "<td valign=top [link][icon][/link]</td><td>{If}
                       {[parent(All':'):Title],[parent(All':'):Title]:}
 6                     [link][Title][/link]</td>"
 7 format CL4Vlist        "<br>[link][Howto][/link]"
 8 format DocumentImages  true
 9 format DocumentText    "<h3>[Title]</h3>\\n\\n<p>[Text]"
10 format DocumentButtons "Expand Text|Expand contents|Detach|Highlight"
```

Examples of classifiers and format strings

Figure 16 shows part of the collection configuration file for the Demo collection. We use this as an example because it has several classifiers which are quite richly formatted. Note that statements in collection configuration files must not contain newline characters—in the Table, longer lines are broken up for readability.

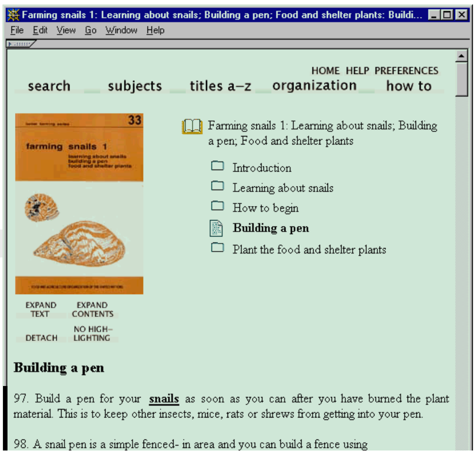
Line 4 specifies the Demo collection's *How To* classifier. This is the fourth in the collection configuration file, and is therefore referred to as CL4. The corresponding format statement is line 7 of Figure 16. The "how to" information is generated from the *List* classifier, and its structure is the plain list of titles shown in Figure 12. The titles are linked to the documents themselves: clicking a title brings up the relevant document. The children of the hierarchy's top level are displayed as a *VList* (vertical list), which lists the sections vertically. As the associated *format* statement indicates, each element of the list is on a newline ("*<br>*") and contains the *Howto* text, hyperlinked to the document itself.

Line 1 specifies the Demo collection's *Subject* classification, referred to as CL1 (the first in the configuration file), and Line 3 the *Organisation* classification CL3. Both are generated by the *Hierarchy* classifier and therefore comprise a hierarchical structure of *VLists*.

Line 2 shows the final classification for the Demo collection, *Titles A-Z* (CL2). Note that there are no corresponding format strings for these three classifiers. Greenstone has built-in defaults for each format string type and so it's not necessary to set a format string unless you want to override the default.

This accounts for the four *classify* lines in Figure 16. Coincidentally, there are also four *format* lines. We have already discussed one, the *CL4Vlist* one. The remaining three are the first type of format string, documented in Table 14. For example, line 8 places the cover image at the top left of each document page. Line 9 formats the actual document text, with the title of the relevant chapter or section preceding the text itself. These are illustrated in Figure 17.

Figure 17
Formatting the document


Farming snails 1: Learning about snails; Building a pen; Food and shelter plants: Buildi...
File Edit View Go Window Help
HOME HELP PREFERENCES
search subjects titles a–z organization how to
33
farming snails 1
EXPAND TEXT
EXPAND CONTENTS
DETACH
NO HIGH-LIGHTING

Farming snails 1: Learning about snails; Building a pen; Food and shelter plants

- Introduction
- Learning about snails
- How to begin
- **Building a pen**
- Plant the food and shelter plants

Building a pen

97. Build a pen for your **snails** as soon as you can after you have burned the plant material. This is to keep other insects, mice, rats or shrews from getting into your pen.

98. A snail pen is a simple fenced- in area and you can build a fence using

```
format DocumentImages true
```

```
format DocumentText
   "<h3>[Title]</h3>\\n\\n<p>[Text]"
```

Line 5 of Figure 16 is a rather complicated specification that formats the query result list returned by a search, whose parts are illustrated in Figure 18. A simplified version of the format string is

```
<td valign=top>[link][icon][/link]</td>
<td>[link][Title][/link]</td>
```

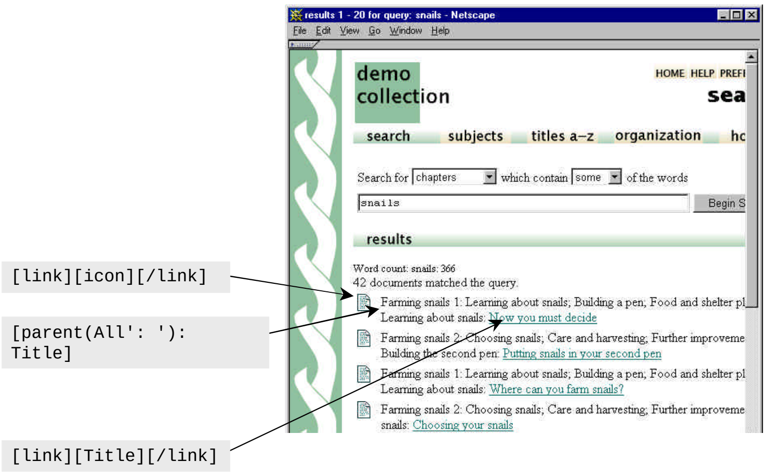
This is designed to appear as a table row, which is how the query results list is formatted. It gives a small icon linked to the text, as usual, and the document title, hyperlinked to the document itself.

In this collection, documents are hierarchical. In fact, the above hyperlink anchor will evaluate to the title of the section returned by the query. However, it would be better to augment it with the title of the enclosing section, the enclosing chapter, and the book in which it occurs. There is a special metadata item, *parent*, which is not stored in documents but is implicit in any hierarchical document, that produces such a list. It either returns the parent document, or, if used with the qualifier *All*, the list of hierarchically enclosing parents, separated by a character string that can be given after the *All* qualifier. Thus

```
<td valign=top>[link][icon][/link]</td>
 <td>{[parent(All': '):Title]: }[link][Title][/link]</td>
```

has the effect of producing a list containing the book title, chapter title, etc. that enclose the target section, separated by colons, with a further colon followed by a hyperlink to the target section's title.

Figure 18
Formatting the search results



Search for chapters which contain some of the words

snails

Word count: snails: 366
42 documents matched the query.
Farming snails 1: Learning about snails; Building a pen; Food and shelter pl
Learning about snails: Now you must decide
Farming snails 2: Choosing snails; Care and harvesting; Further improveme
Building the second pen: Putting snails in your second pen
Farming snails 1: Learning about snails; Building a pen; Food and shelter pl
Learning about snails: Where can you farm snails?
Farming snails 2: Choosing snails; Care and harvesting; Further improveme
snails: Choosing your snails

Unfortunately, if the target is itself a book, there is no parent and so an empty string will appear followed by a colon. To get around this problem, you can use *if* and *or … else* statements in a format string:

```
{If}{[metadata], action-if-non-null, action-if-null}
{Or}{action, else another-action, else another-action, etc}
```

In either case curly brackets are used to signal that the statements should be interpreted and not just printed out as text. The *If* tests whether the metadata is empty and takes the first clause if not, otherwise the second one (if it exists). Any metadata item can be used, including the special metadata *parent*. The *Or* statement evaluates each action in turn until one is found that is non-null. That one is sent to the output and the remaining actions are skipped.

Returning to line 5 of Figure 16, the full format string is

```
<td valign=top>[link][icon][/link]</td>
<td>{If}{[parent(All': '):Title],
         [parent(All': '):Title]:}
    [link][Title][/link]</td>
```

This precedes the *parent* specification with a conditional that checks whether the result will be empty and only outputs the parent string when it is present. Incidentally, *parent* can be qualified by *Top* instead of *All*, which gives the top-level document name that encloses a section—in this case, the book name. No separating string is necessary with *Top*.

Some final examples illustrate other features. The *DateList* in Figure 13 is used in the *Dates* classification of the Computists' Weekly collection (which happens to be the second classifier, CL2). The classifier and format specifications are shown below. The *DateList* classifier differs from *AZList* in that it always sorts by *Date* metadata, and the bottom branches of the browsing hierarchy use *DateList* instead of *VList*, which causes the year and month to be added at the left of the document listings.

```
classify AZSectionList metadata=Creator
format CL2Vlist   "<td>[link][icon][/link]</td>
                  <td>[Creator]</td>
                  <td>&nbsp;&nbsp;[Title]</td>
                  <td>[parent(Top):Date]</td>"
```

The format specification shows these *VLists* in the appropriate way.

The format-string mechanism is powerful, but tricky to learn. The best way is by studying existing collection configuration files.

2.4 Controlling the Greenstone user interface

The entire Greenstone user interface is controlled by macros which reside in the *GSDLHOME/macros* directory. They are written in a language designed especially for Greenstone, and are used by the *library* program at run time to generate web pages. Translating the macro language into HTML is the last step in displaying a page. Thus changes to a macro file affect the display immediately, making experimentation quick and easy. All macro files used by *library* are listed in *GSDLHOME/etc/main.cfg* and loaded every time it starts.

Web pages are generated on the fly for a number of reasons, and the macro system is how Greenstone implements the necessary flexibility. Pages can be presented in many languages, and a different macro file is used to store all the interface text in each language. When the *library* program displays a page the macro interpreter checks a language variable and loads the page in the appropriate language (this does not, unfortunately, extend to translating document content). Also, the values of certain display variables, like the number of documents found by a search, are not known ahead of time; these are interpolated into the page text in the form of macros.

The macro file format

Macro files have a *.dm* extension. Each file defines one or more *packages*, each containing a series of macros used for a single purpose. Like classifiers and plugins, there is a basis from which to build macros, called *base.dm*; this file defines the basic content of a page.

Macros have names that begin and end with an underscore, and their content is defined using curly brackets. Content can be plain text, HTML (including links to Java applets and JavaScript), macro names, or any combination of these. This macro from *base.dm* defines the content of a page in the absence of any overriding macro:

```
_content_ {<p><h2>Oops</h2>_textdefaultcontent_}
```

The page will read “Oops” at the top, and *_textdefaultcontent_*, which is defined, in English, to be *The requested page could not be found. Please use your browsers ‘back’ button or the above home button to return to the Greenstone Digital Library, and in other languages to be a suitable translation of this sentence.*

*_textdefaultcontent_* and *_content_* both reside in the *global* package because they are required by all parts of the user interface. Macros can use macros from other packages as content, but they must prefix their names with their package name. For example,

```
_collectionextra_ {This collection contains _about:numdocs_
documents. It was last built _about:builddate_ days ago.)
```

comes from *english.dm*, and is used as the default description of a collection. It is part of the *global* package, but *_numdocs_* and *_builddate_* are both in the *about* package—hence the *about:* preceding their names.

Macros often contain conditional statements. They resemble the format string conditional described above, though their appearance is slightly different. The basic format is *_If_(x,y,z)*, where *x* is a condition, *y* is the macro content to use if that condition is true, and *z* the content if it is false. Comparison operators are the same as the simple ones used in Perl (less than, greater than, equals, not equals). This example from *base.dm* is used to determine how to display the top of a collection’s *about* page:

```
_imagecollection_ {
    _If_("_iconcollection_" ne "",
         <a href = "_httppageabout_">
             <img src = "_iconcollection_" border = 0>
             </a>,
         _imagecollectionv_)
}
```

This looks rather obscure. *_iconcollection_* resolves to the empty string if the collection doesn’t have an icon, or the filename of an image. To paraphrase the above code: If there is a collection image, display the *About this Collection* page header (referred to by *_httppageabout_*) and then the image; otherwise use the alternative display *_imagecollectionv_*.

Macros can take arguments. Here is is a second definition for the

Table 16 The *packages* directory

Directory	Package	URL
mg	MG, short for "Managing Gigabytes." Compression, indexing and search software used to manage textual information in Greenstone collections.	*www.citri.edu.au/mg*
wget	Web mirroring software for use with Greenstone. Written in C++	*www.tuwien.ac.at/~prikryl/wget.html*
w3mir	A web mirroring program written in Perl. This is not Greenstone's preferred mirroring program because it relies on a specific outdated version of a certain Perl module (which is distributed in the *w3mir* directory).	*www.math.uio.no/~janl/w3mir*
windows	Packages used when running under Windows.	–
windows/gdbm	Version of the GNU Database Manager created for Windows. GDBM comes as a standard part of Linux.	–
windows/crypt	Encryption program used for passwords for Greenstone's administrative functions.	–
wv	Microsoft Word converter (for building collections from Word documents) slimmed down for Greenstone.	*sourceforge.net/projects/wvware*
pdftohtml	PDF converter used when building collections from PDF documents.	*www.ra.informatik.uni-stuttgart.de/~gosho/pdftohtml*
yaz	Z39.50 client program being used for research in making Greenstone Z39.50 compliant. Progress is reported in the *README.gsdl* file.	*www.indexdata.dk*

*_imagecollection_* macro which immediately follows the definition given above in the *base.dm* file:

```
_imagecollection_[v=1]{_imagecollectionv_}
```

The argument *[v=1]* specifies that the second definition is used when Greenstone is running in text-only mode. The language macros work similarly—apart from *english.dm*, because it is the default, all language macros specify their language as an argument. For example,

```
_textimagehome_ {Home Page}
```

appears in the English language macro file, whereas the German version is

Figure 19
Part of the *about.dm* macro file

```
package about

######################################################################
# about page content
######################################################################

_pagetitle_ {_collectionname_}

_content_ {
<center>
_navigationbar_
</center>
_query:queryform_
<p>_iconblankbar_
<p>_textabout_
_textsubcollections_
<h3>_help:textsimplehelpheading_</h3>
_help:simplehelp_
}

_textabout_ {
<h3>_textabcol_</h3>
_Global:collectionextra_
}
```

```
   _textimagehome_ [l=de] {Hauptaseite}
```

The English and German versions are in the same package, though they are in separate files (package definitions may span more than one file). Greenstone uses its *l* argument at run time to determine which language to display.

As a final example, Figure 19 shows an exerpt from the macro file *about.dm* that is used to generate the “About this collection” page for each collection. It shows three macros being defined, *_pagetitle_*, *_content_* and *_textabout_*.

Using macros

Macros are powerful, and can be a little obscure. However, with a good knowledge of HTML and a bit of practice, they become a quick and easy way to customise your Greenstone site.

For example, suppose you wanted to create a static page that looked like your current Greenstone site. You could create a new package, called *static*, for example, in a new file, and override the *_content_* macro. Add the new filename to the list of macros in *GSDLHOME/etc/main.cfg* which Greenstone loads every time it is invoked. Finally, access the new page at *http://servername/cgi-bin/library?a=p&p=static*, where *servername* is the name of your Greenstone server.

To change the “look and feel” of Greenstone you can edit the *base* and *style* packages. To change the Greenstone home page, edit the *home* package (this is described in the *Greenstone Digital Library Installer's Guide*). To change the query page, edit *query.dm*.

Experiment freely with macros. Changes appear instantly, because macros are interpreted as pages are displayed. The macro language is a powerful tool that can be used to make your Greenstone site your own.

2.5 The *packages* directory

The *packages* directory, whose contents are shown in Table 16, is where all the code used by Greenstone but written by other research teams resides. All software distributed with Greenstone has been released under the GNU Public license. The executables produced by these packages are placed in the Greenstone *bin* directory. Each package is stored in a directory of its own. Their functions vary widely, from indexing and compression to converting Microsoft Word documents to HTML. Each package has a README file which gives more information about it.

3
The Greenstone runtime system

This chapter describes the Greenstone runtime system so that you can augment and extend its capabilities. The software is written in C++ and makes extensive use of virtual inheritance. If you are unfamiliar with this language you should learn about it before proceeding. Deitel and Deitel (1994) provide a comprehensive tutorial, while Stroustroup (1997) is the definitive reference.

We begin by explaining the design philosophy behind the runtime system since this has a strong bearing on implementation. Then we provide the implementation details, which forms the main part of this chapter.

3.1 Process structure

Figure 20 shows several users, represented by computer terminals at the top of the diagram, accessing three Greenstone collections. Before going online, these collections undergo the importing and building processes described in earlier chapters. First, documents, shown at the bottom of the figure, are imported into the GML format. Then the GML files are built into various searchable indexes and a collection information database that includes the hierarchical structures that support browsing. This done, the collection is ready to go online and respond to requests for information.

Two components are central to the design of the runtime system: “receptionists” and “collection servers.” From a user’s point of view, a receptionist is the point of contact with the digital library. It accepts user input, typically in the form of keyboard entry and mouse clicks; analyzes it; and then dispatches a request to the appropriate collection server (or servers). This locates the requested piece of information and returns it to the receptionist for presentation to the user. Collection servers act as an abstract mechanism that handle the content of the collection, while

Figure 20
Overview of a general Greenstone system

receptionists are responsible for the user interface.

As Figure 20 shows, receptionists communicate with collection servers using a defined protocol. The implementation of this protocol depends on the computer configuration on which the digital library system is running. The most common case, and the simplest, is when there is one receptionist and one collection server, and both run on the same computer. This is what you get when you install Greenstone. In this case the two processes are combined to form a single executable (called *library*), and consequently using the protocol reduces to making function calls. We call this the *null protocol*. It forms the basis for the standard out-of-the-box Greenstone digital library system. This simplified configuration is illustrated in Figure 21, with the receptionist, protocol and collection server bound together as one entity, the *library* program. The aim of this chapter is to show how it works.

Usually, a “server” is a persistent process that, once started, runs indefinitely, responding to any requests that come in. Despite its name, however, the collection server in the null protocol configuration is not a server in this sense. In fact, every time any Greenstone web page is requested, the *library* program is started up (by the CGI mechanism), responds to the request, and then exits. We call it a “server” because it is

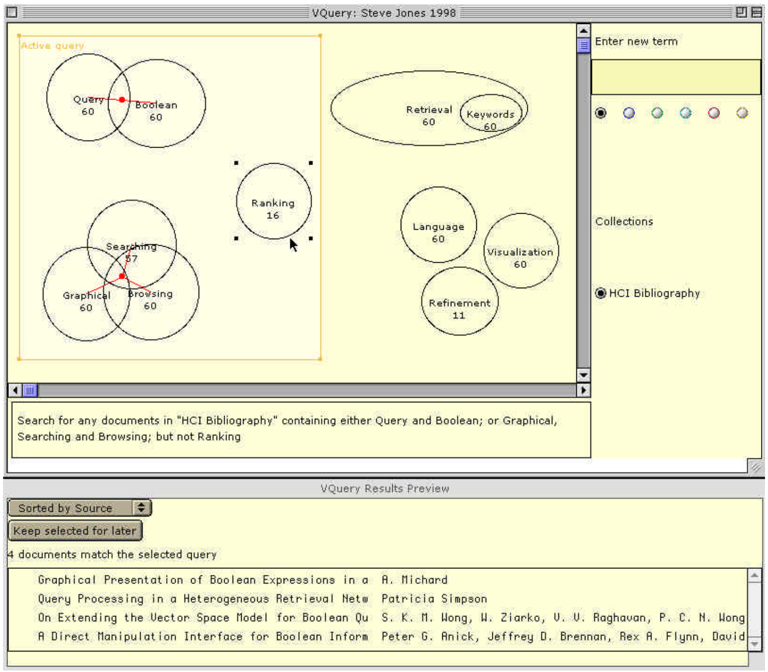
Figure 21
The Greenstone system using the "null protocol"

also designed to work in the more general configuration of Figure 20.

Surprisingly, this is not as slow as one might expect, and results in a perfectly usable service. However, it is clearly inefficient. There is a mechanism called Fast-CGI (*www.fastcgi.com*) which provides a middle ground. Using it, the *library* program can remain in memory at the end of the first execution, and have subsequent sets of CGI arguments fed to it, thus avoiding repeated initialisation overheads and accomplishing much the same behaviour as a server. Using Fast-CGI is an option in Greenstone, and is enabled by recompiling the source code with appropriate libraries.

As an alternative to the null protocol, the Greenstone protocol has also been implemented using the well-known CORBA scheme (Slama *et al.*, 1999). This uses a unified object oriented paradigm to enable different processes, running on different computer platforms and implemented in different programming languages, to access the same set of distributed objects over the Internet (or any other network). Then, scenarios like Figure 19 can be fully implemented, with all the receptionists and collection servers running on different computers.

Figure 22
Graphical query interface to Greenstone



Search for any documents in "HCI Bibliography" containing either Query and Boolean; or Graphical, Searching and Browsing; but not Ranking

4 documents match the selected query

```
Graphical Presentation of Boolean Expressions in a  A. Michard
Query Processing in a Heterogeneous Retrieval Netw  Patricia Simpson
On Extending the Vector Space Model for Boolean Qu  S. K. M. Wong, W. Ziarko, V. V. Raghavan, P. C. N. Wong
A Direct Manipulation Interface for Boolean Inform  Peter G. Anick, Jeffrey D. Brennan, Rex A. Flynn, David
```

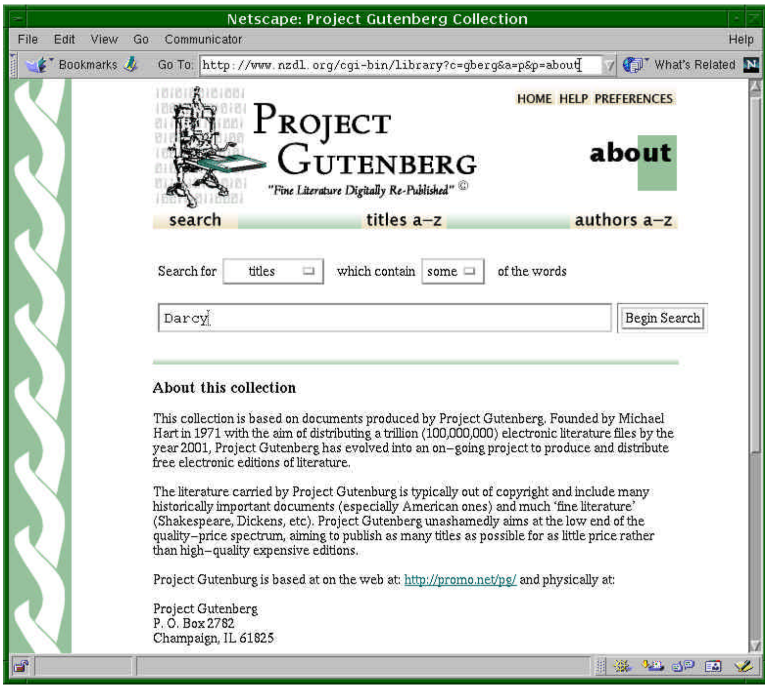
This allows far more sophisticated interfaces to be set up to exactly the same digital library collections. As just one example, Figure 22 shows a graphical query interface, based on Venn diagrams, that lets users manipulate Boolean queries directly. Written in Java, the interface runs locally on the user's own computer. Using CORBA, it accesses a remote Greenstone collection server, written in C++.

The distributed protocol is still being refined and readied for use, and so this manual does not discuss it further (see Bainbridge *et al*., submitted, for more information).

3.2 Conceptual framework

Figure 23 shows the "about this collection" page of a particular Greenstone collection (the Project Gutenberg collection). Look at the URL at the top. The page is generated as a result of running the CGI program called *library*, which is the above-mentioned executable comprising both receptionist and collection server connected by the null protocol. The arguments to *library* are *c=gberg*, *a=p*, and *p=about*. They can be interpreted as follows:

Figure 23
Generating the "about this collection" page


Netscape: Project Gutenberg Collection
File Edit View Go Communicator Help
Bookmarks
Go To:
What's Related
HOME HELP PREFERENCES
PROJECT GUTENBERG
"Fine Literature Digitally Re-Published" ©
about
search
titles a–z
authors a–z
Begin Search

http://www.nzdl.org/cgi-bin/library?c=gberg&a=p&p=about

Search for titles which contain some of the words

Darcy

About this collection

This collection is based on documents produced by Project Gutenberg. Founded by Michael Hart in 1971 with the aim of distributing a trillion (100,000,000) electronic literature files by the year 2001, Project Gutenberg has evolved into an on–going project to produce and distribute free electronic editions of literature.

The literature carried by Project Gutenburg is typically out of copyright and include many historically important documents (especially American ones) and much 'fine literature' (Shakespeare, Dickens, etc). Project Gutenberg unashamedly aims at the low end of the quality–price spectrum, aiming to publish as many titles as possible for as little price rather than high–quality expensive editions.

Project Gutenburg is based at on the web at: http://promo.net/pg/ and physically at:

Project Gutenberg
P. O. Box 2782
Champaign, IL 61825

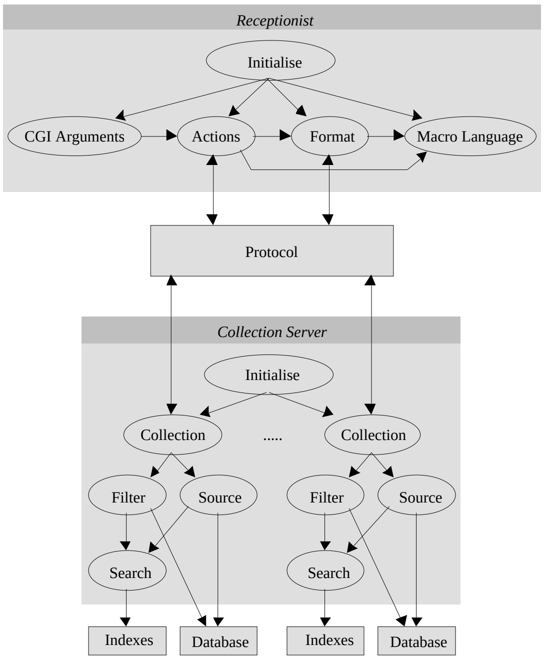
For the Project Gutenberg collection (c=gberg), the action is to generate a page (a=p), and the page to generate is called "about" (p=about).

Figure 24 illustrates the main parts of the Greenstone runtime system. At the top, the receptionist first initialises its components, then parses the CGI arguments to decide which action to call. In performing the action (which includes further processing of the CGI arguments), the software uses the protocol to access the content of the collection. The response is used to generate a web page, with assistance from the format component and the macro language.

The macro language, which we met in Section 2.4, is used to provide a Greenstone digital library system with a consistent style, and to create interfaces in different languages. Interacting with the library generates bare bones of web pages; the macros in *GSDLHOME/macros* put flesh on them.

The Macro Language object in Figure 24 is responsible for reading these files and storing the parsed result in memory. Any action can use this object to expand a macro. It can even create new macro definitions and

Figure 24
Greenstone runtime system



override existing ones, adding a dynamic dimension to macro use.

The layout of the "about this collection" page (Figure 23) is known before runtime, and encoded in the macro file *about.dm*. Headers, footers, and the background image are not even mentioned because they are located in the *Global* macro package. However, the specific "about" text for a particular collection is not known in advance, but is stored in the collection information database during the building process. This information is retrieved using the protocol, and stored as *_collectionextra_* in the *Global* macro package. To generate the content of the page, the *_content_* macro in the *about* package (shown in Figure 19) is expanded. This in turn expands *_textabout_*, which itself accesses *_collectionextra_*, which had just been dynamically placed there.

One further important ingredient is the Format object. Format statements in the collection configuration file affect the presentation of particular pieces of information, as described in Section 2.3. They are handled by the Format object in Figure 24. This object's main task is to parse and evaluate statements such as the format strings in Figure 16. As we learned in Section 2.3, these can include references to metadata in square brackets (e.g. *[Title]*), which need to be retrieved from the collection server. Interaction occurs between the Format object and the Macro Language object, because format statements can include macros that, when expanded, include metadata, which when expanded include macros, and so on.

At the bottom of Figure 24, the collection server also goes through an initialisation process, setting up Filter and Source objects to respond to incoming protocol requests, and a Search object to assist in this task. Ultimately these access the indexes and the collection information database, both formed during collection building.

Ignoring blank lines, the receptionist contains 15,000 lines of code. The collection server contains only 5,000 lines (75% of which are taken up by header files). The collection server is more compact because content retrieval is accomplished through two pre-compiled programs. MG, a full-text retrieval system, is used for searching, and GDBM, a database management system, is used to hold the collection information database.

To encourage extensibility and flexibility, Greenstone uses inheritance widely—in particular, within Action, Filter, Source, and Search. For a simple digital library dedicated to text-based collections, this means that you need to learn slightly more to program the system. However, it also means that MG and GDBM could easily be replaced should the need arise. Furthermore, the software architecture is rich enough to support full multimedia capabilities, such as controlling the interface through speech input, or submitting queries as graphically drawn pictures.

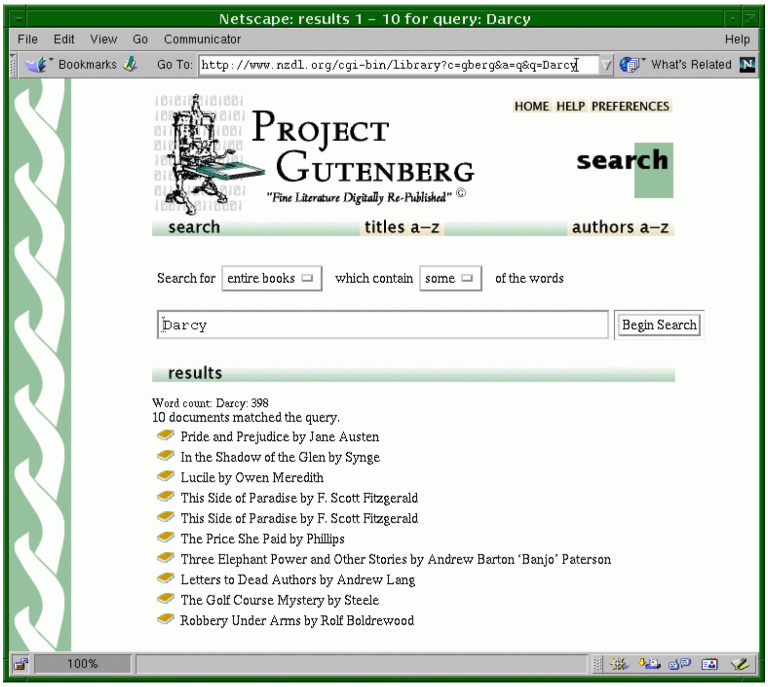
3.3 How the conceptual framework fits together

Sections 3.7 and 3.9 explain the operation of the collection server and receptionist in more detail, expanding on each module in Figure 23 and describing how it is implemented. It is helpful to first work through examples of a user interacting with Greenstone, and describe what goes on behind the scenes. For the moment, we assume that all objects are correctly initialised. Initialisation is a rather intricate procedure that we revisit in Section 3.10.

Performing a search

When a user enters a query by pressing *Begin search* on the search page, a new Greenstone action is invoked, which ends up by generating a new HTML page using the macro language. Figure 25 shows the result of searching the Project Gutenberg collection for the name *Darcy*. Hidden within the HTML of the original search page is the statement *a=q*. When the search button is pressed this statement is activated, and sets the new action to be *queryaction*. Executing *queryaction* sets up a call to the

Figure 25
Searching the Gutenberg collection for *Darcy*



http://www.nzdl.org/cgi-bin/library?c=gberg&a=q&q=Darcy

Search for entire books which contain some of the words

Darcy

Word count: Darcy: 398
10 documents matched the query.

- Pride and Prejudice by Jane Austen
- In the Shadow of the Glen by Synge
- Lucile by Owen Meredith
- This Side of Paradise by F. Scott Fitzgerald
- This Side of Paradise by F. Scott Fitzgerald
- The Price She Paid by Phillips
- Three Elephant Power and Other Stories by Andrew Barton 'Banjo' Paterson
- Letters to Dead Authors by Andrew Lang
- The Golf Course Mystery by Steele
- Robbery Under Arms by Rolf Boldrewood

designated collection's Filter object (*c=gberg*) through the protocol.

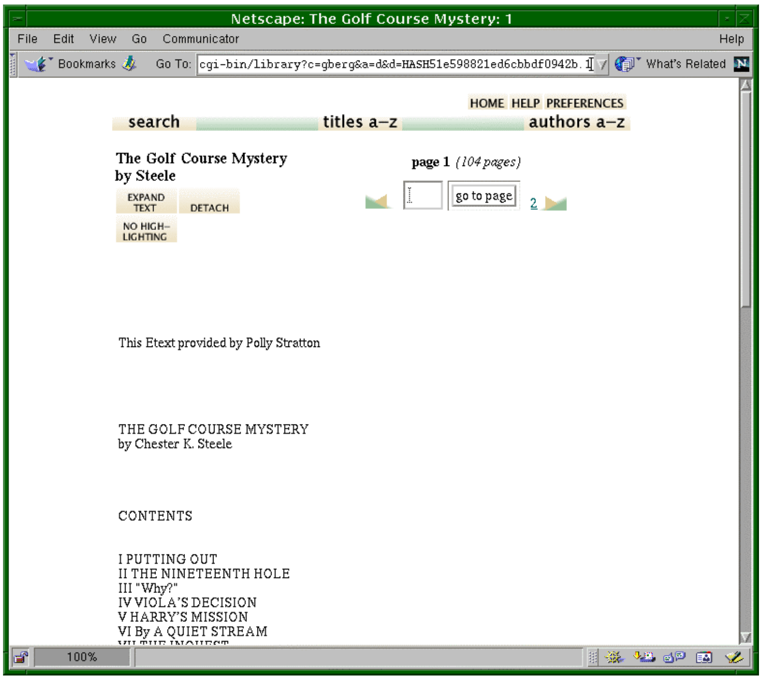
Filters are an important basic function of collection servers. Tailored for both searching and browsing activities, they provide a way of selecting a subset of information from a collection. In this case, the *queryaction* sets up a filter request by:

- setting the filter request type to be *QueryFilter* (Section 3.7 describes the different filter types);
- storing the user's search preferences—case-folding, stemming and so on—in the filter request;
- calling the *filter()* function using the null protocol.

Calls to the protocol are synchronous. The receptionist is effectively blocked until the filter request has been processed by the collection server and any data generated has been returned.

When a protocol call of type *QueryFilter* is made, the Filter object (in Figure 24) decodes the options and makes a call to the Search object, which uses MG to do the actual search. The role of the Search object is to provide an abstract program interface that supports searching, regardless of the underlying search tool being used. The format used for returning

Figure 26
The Golf Course Mystery



Netscape: The Golf Course Mystery: 1
File Edit View Go Communicator Help
Bookmarks
Go To: cgi-bin/library?c=gberg&a=d&d=HASH51e598821ed6cbbdf0942b.1
What's Related
HOME HELP PREFERENCES
search
titles a–z
authors a–z
EXPAND TEXT
DETACH
NO HIGH–LIGHTING
go to page
2
100%

The Golf Course Mystery
by Steele

page 1 *(104 pages)*

This Etext provided by Polly Stratton

THE GOLF COURSE MYSTERY
by Chester K. Steele

CONTENTS

I PUTTING OUT
II THE NINETEENTH HOLE
III "Why?"
IV VIOLA'S DECISION
V HARRY'S MISSION
VI By A QUIET STREAM
VII THE INQUEST

results also enforces abstraction, requiring the Search object to translate the data generated by the search tool into a standard form.

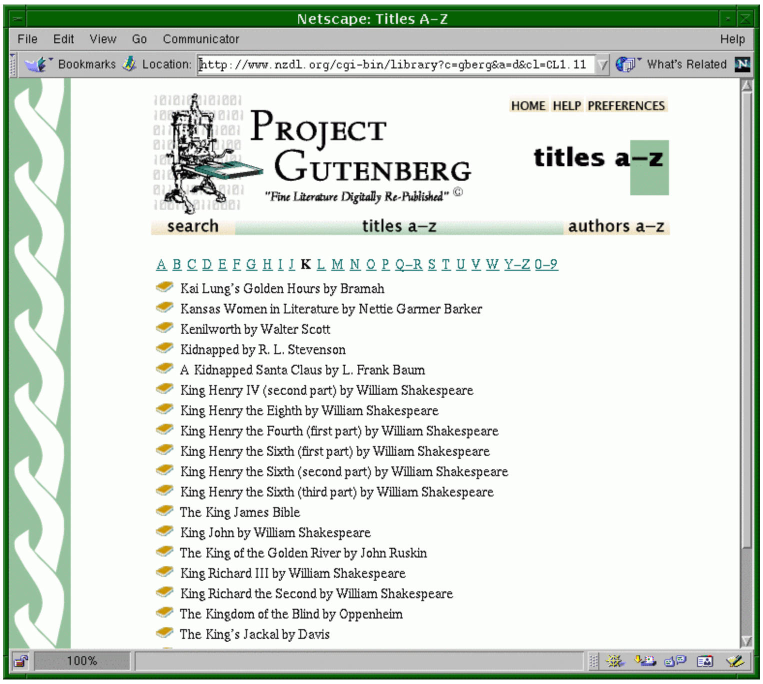
Once the search results have been returned to the receptionist, the action proceeds by formatting the results for display, using the Format object and the Macro Language. As Figure 25 shows, this involves generating the standard Greenstone header, footer, navigation bar and background; repeating the main part of the query page just beneath the navigation bar; and displaying a book icon, title and author for each matching entry. The format of this last part is governed by the *format SearchVList* statement in the collection configuration file. Before title and author metadata can be displayed, they must be retrieved from the collection server. This requires further calls to the protocol, this time using *BrowseFilter*.

Retrieving a document

Following the above query for *Darcy*, consider what happens when a document is displayed. Figure 26 shows the result of clicking on the icon beside *The Golf Course Mystery* in Figure 25.

The source text for the Gutenberg collection comprises one long file per book. At build time, these files are split into separate pages every 200

Figure 27
Browsing titles in the Gutenberg collection

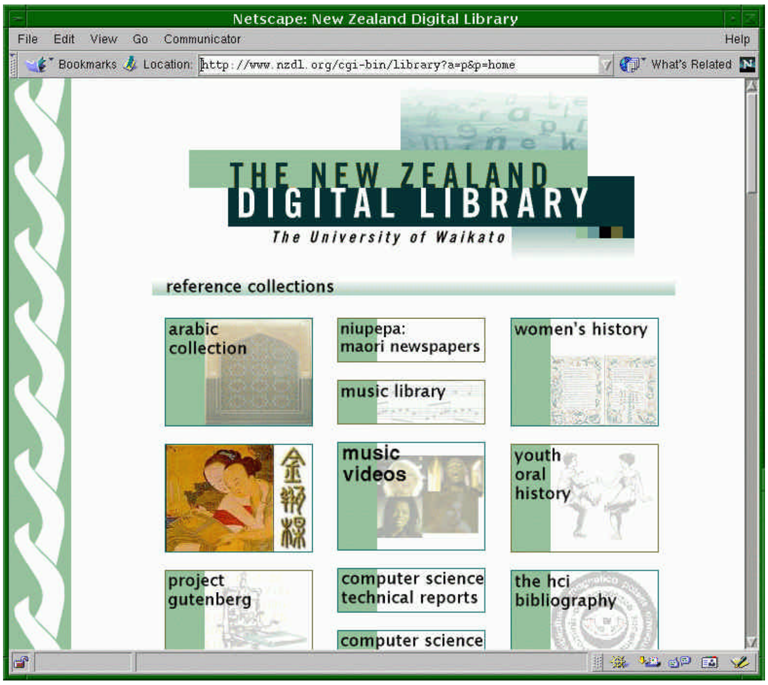

Netscape: Titles A–Z
File Edit View Go Communicator Help
Bookmarks Location: http://www.nzdl.org/cgi-bin/library?c=gberg&a=d&cl=CL1.11
What's Related
HOME HELP PREFERENCES
PROJECT GUTENBERG
"Fine Literature Digitally Re-Published" ©
titles a–z
search
titles a–z
authors a–z
A B C D E F G H I J K L M N O P Q–R S T U V W Y–Z 0–9
100%

Kai Lung's Golden Hours by Bramah
Kansas Women in Literature by Nettie Garmer Barker
Kenilworth by Walter Scott
Kidnapped by R. L. Stevenson
A Kidnapped Santa Claus by L. Frank Baum
King Henry IV (second part) by William Shakespeare
King Henry the Eighth by William Shakespeare
King Henry the Fourth (first part) by William Shakespeare
King Henry the Sixth (first part) by William Shakespeare
King Henry the Sixth (second part) by William Shakespeare
King Henry the Sixth (third part) by William Shakespeare
The King James Bible
King John by William Shakespeare
The King of the Golden River by John Ruskin
King Richard III by William Shakespeare
King Richard the Second by William Shakespeare
The Kingdom of the Blind by Oppenheim
The King's Jackal by Davis

lines or so, and relevant information for each page is stored in the indexes and collection information database. The top of Figure 26 shows that this book contains 104 computer-generated pages, and below it is the beginning of page one: who entered it, the title, the author, and the beginnings of a table of contents (this table forms part of the Gutenberg source text, and was not generated by Greenstone). At the top left are buttons that control the document's appearance: just one page or the whole document; whether query term highlighting is on or off; and whether or not the book should be displayed in its own window, detached from the main searching and browsing activity. At the top right is a navigation aid that supports direct access to any page in the book: simply type in the page number and press the "go to page" button. Alternatively, the next and previous pages are retrieved by clicking on the arrow icons either side of the page selection widget.

The action for retrieving documents, *documentaction*, is specified by setting *a=d* and takes several additional arguments. Most important is the document to retrieve: this is specified through the *d* variable. In Figure 26 it is set to *d=HASH51e598821ed6cbbdf0942b.1* to retrieve the first page of the document with the identifier *HASH51e598821ed6cbbdf0942b*, known in more friendly terms as *The Golf Course Mystery*. There are further variables: whether query term highlighting is on or off (*hl*) and

Figure 28
The New Zealand Digital Library home page



Location: http://www.nzdl.org/cgi-bin/library?a=p&p=home

which page within a book is displayed (*gt*). These variables are used to support the activities offered by the buttons on the page in Figure 26, described above. Defaults are used if any of these variables are omitted.

The action follows a similar procedure to *queryaction*: appraise the CGI arguments, access the collection server using the protocol, and use the result to generate a web page. Options relating to the document are decoded from the CGI arguments and stored in the object for further work. To retrieve the document from the collection server, only the document identifier is needed to set up the protocol call to *get_document()*. Once the text is returned, considerable formatting must be done. To achieve this, the code for *documentaction* accesses the stored arguments and makes use of the Format object and the Macro Language.

Browsing a hierarchical classifier

Figure 27 shows an example of browsing, where the user has chosen *Titles A-Z* and pressed the button for the letter *K*. The action that supports this is also *documentaction*, given by the CGI argument *a=d* as before. However, whereas before a *d* variable was included, this time there is none. Instead, the node within the browsable classification hierarchy to display is specified in the variable *cl*. In our case this represents titles grouped under the letter *K*. This list was formed at build time and stored in the collection information database.

Records that represent classifier nodes in the database use the prefix *CL*, followed by numbers separated by periods (.) to designate where they lie within the nested structure. Ignoring the search button (leftmost in the navigation bar), classifiers are numbered sequentially in increasing order, left to right, starting at 1. Thus the top level classifier node for titles in our example is *CL1* and the page sought is generated by setting *cl=CL1.11*. This can be seen in the URL at the top of Figure 27.

To process a *cl* document request, the Filter object is used to retrieve the node over the protocol. Depending on the data returned, further protocol calls are made to retrieve document metadata. In this case, the titles of the books are retrieved. However, if the node were an interior one whose children are themselves nodes, the titles of the child nodes would be retrieved. From a coding point of view this amounts to the same thing, and is handled by the same mechanism.

Finally, all the retrieved information is bound together, using the macro language, to produce the web page shown in Figure 27.

Generating the home page

As a final example, we look at generating the Greenstone home page. Figure 28 shows the New Zealand Digital Library home page, and its URL includes the arguments *a=p* and *p=home*. Thus, like the "about this collection" page, it is generated by a *pageaction* (*a=p*), but this time the page to produce is *home* (*p=home*). The macro language, therefore, accesses the content of *home.dm*. There is no need to specify a collection (with the *c* variable) in this case.

The purpose of the home page is to show what collections are available. Clicking on an icon takes the user to the "about this collection" page for that collection. The menu of collections is dynamically generated every time the page is loaded, based on the collections that are in the file system at that time. When a new one comes online, it automatically appears on the home page when that page is reloaded (provided the collection is stipulated to be "public").

To do this the receptionist uses the protocol (of course). As part of appraising the CGI arguments, *pageaction* is programmed to detect the special case when *p=home*. Then, the action uses the protocol call *get_collection_list()* to establish the current set of online collections. For

Table 17 Standalone programs included in Greenstone

Subdirectory	Role
setpasswd/	Password support for Windows.
getpw/	Password support for Unix.
txt2db/	Convert an XML-like ASCII text format to GNU's database format.
db2txt/	Convert the GNU database format to an XML-like ASCII text format.
phind/	Hierarchical phrase browsing tool.
hashfile/	Compute unique document ID based on content of file.
mgpp/	Rewritten and updated version of Managing Gigabytes package in C++.
w32server/	Local library server for Windows.
checkis/	Specific support for installing Greenstone under Windows.

each of these it calls *get_collectinfo()* to obtain information about it. This information includes whether the collection is publicly available, what the URL is for the collection's icon (if any), and the collection's full name. This information is used to generate an appropriate entry for the collection on the home page.

3.4 Source code

The source code for Greenstone resides in *GSDLHOME/src*. It occupies two subdirectories, *recpt* for the receptionist's code and *colservr* for the collection server's. Greenstone runs on Windows systems right down to Windows 3.1, and unfortunately this imposes an eight-character limit on file and directory names. This explains why cryptic abbreviations like *recpt* and *colservr* are used. The remaining subdirectories include standalone utilities, mostly in support of the building process. They are listed in Table 17.

Another directory, *GSDLHOME/lib*, includes low-level objects that are used by both receptionist and collection server. This code is described in Section 3.5.

Greenstone makes extensive use of the Standard Template Library (STL), a widely used C++ library from Silicon Graphics (*www.sgi.com*) that is the result of many years of design and development. Like all programming libraries it takes some time to learn. Appendix A gives a brief overview of key parts that are used throughout the Greenstone code. For a fuller description, consult the official STL reference manual, available online at *www.sgi.com*, or one of the many STL textbooks, for example Josuttis (1999).

Figure 29
The *text_t* API
(abridged)

```
typedef vector<unsigned short> usvector;

class text_t {
protected:
usvector text;
unsigned short encoding; // 0 = unicode, 1 = other

public:
  // constructors
  text_t ();
  text_t (int i);
  text_t (char *s); // assumed to be a normal c string

  void setencoding (unsigned short theencoding);
  unsigned short getencoding ();

  // STL container support
  iterator begin ();
  iterator end ();

  void erase(iterator pos);
  void push_back(unsigned short c);
  void pop_back();

  void reserve (size_type n);

  bool empty () const {return text.empty();}
  size_type size() const {return text.size();}

  // added functionality
  void clear ();
  void append (const text_t &t);

  // support for integers
  void appendint (int i);
  void setint (int i);
  int getint () const;

  // support for arrays of chars
  void appendcarr (char *s, size_type len);
  void setcarr (char *s, size_type len);
};
```

3.5 Common Greenstone types

The objects defined in *GSDLHOME/lib* are low-level Greenstone objects, built on top of STL, which pervade the entire source code. First we describe *text_t*, an object used to represent Unicode text, in some detail. Then we summarize the purpose of each library file.

The *text_t* object

Greenstone works with multiple languages, both for the content of a collection and its user interface. To support this, Unicode is used throughout the source code. The underlying object that realises a Unicode string is *text_t*.

Unicode uses two bytes to store each character. Figure 29 shows the main features of the *text_t* Application Program Interface (API). It fulfils the

Figure 30
The overloaded operators to *text_t*

```
 1 class text_t {
 2   // ...
 3   public:
 4   text_t &operator=(const text_t &x);
 5   text_t &operator+= (const text_t &t);
 6   reference operator[](size_type n);
 7
 8   text_t &operator=(int i);
 9   text_t &operator+= (int i);^ \\
10   text_t &operator= (char *s);
11   text_t &operator+= (char *s);
12
13   friend inline bool operator!=(const text_t& x, const text_t& y);
14   friend inline bool operator==(const text_t& x, const text_t& y);
15   friend inline bool operator< (const text_t& x, const text_t& y);
16   friend inline bool operator> (const text_t& x, const text_t& y);
17   friend inline bool operator>=(const text_t& x, const text_t& y);
18   friend inline bool operator<=(const text_t& x, const text_t& y);
19   // ...
20 };
```

two-byte requirement using the C++ built-in type *short*, which is defined to be a two byte integer. The data type central to the *text_t* object is a dynamic array of unsigned shorts built using the STL declaration *vector<unsigned short>* and given the abbreviated name *usvector*.

The constructor functions (lines 10–12) explicitly support three forms of initialisation: construction with no parameters, which generates an empty Unicode string; construction with an integer parameter, which generates a Unicode text version of the numeric value provided; and construction with a *char** parameter, which treats the argument as a null-terminated C++ string and generates a Unicode version of it.

Following this, most of the detail (lines 17–28) is taken up maintaining an STL vector-style container: *begin()*, *end()*, *push_back()*, *empty()* and so forth. There is also support for clearing and appending strings, as well as for converting an integer value into a Unicode text string, and returning the corresponding integer value of text that represents a number.

There are many overloaded operators that do not appear in Figure 29. To give a flavour of the operations supported, these are shown in Figure 30. Line 4 supports assignment of one *text_t* object to another, and line 5 overloads the += operator to provide a more natural way to append one *text_t* object to the end of another. It is also possible, through line 6, to access a particular Unicode character (represented as a *short*) using array subscripting [ ]. Assign and append operators are provided for integers and C++ strings. Lines 12–18 provide Boolean operators for comparing two *text_t* objects: equals, does not equal, precedes alphabetically, and so on.

Member functions that take *const* arguments instead of non-*const* ones are also provided (but not shown here). Such repetition is routine in C++

objects, making the API fatter but no bigger conceptually. In reality, many of these functions are implemented as single in-line statements. For more detail, refer to the source file *GSDLHOME/lib/text_t.h*.

Files in *GSDLHOME/lib*

The header files in *GSDLHOME/lib* include a mixture of functions and objects that provide useful support for the Greenstone runtime system. Where efficiency is of concern, functions and member functions are declared *inline*. For the most part, implementation details are contained within a header file's *.cpp* counterpart.

cfgread.h Functions to read and write configuration files. For example, *read_cfg_line()* takes as arguments the input stream to use and the *text_tarray* (shorthand for *vector<text_t>*) to fill out with the data that is read.

display.h A sophisticated object used by the receptionist for setting, storing and expanding macros, plus supporting types. Section 3.9 gives further details.

fileutil.h Function support for several file utilities in an operating system independent way. For example, *filename_cat()* takes up to six *text_t* arguments and returns a *text_t* that is the result of concatenating the items together using the appropriate directory separator for the current operating system.

gsdlconf.h System-specific functions that answer questions such as: does the operating system being used for compilation need to access *strings.h* as well as *string.h*? Are all the appropriate values for file locking correctly defined?

gsdltimes.h Function support for date and times. For example, *time2text()* takes computer time, expressed as the number of seconds that have elapsed since 1 January 1970, and converts it into the form YYYY/MM/DD hh:mm:ss, which it returns as type *text_t*.

gsdltools.h Miscellaneous support for the Greenstone runtime system: clarify if littleEndian or bigEndian; check whether Perl is available; execute a system command (with a few bells and whistles); and escape special macro characters in a *text_t* string.

gsdlunicode.h A series of inherited objects that support processing Unicode *text_t* strings through IO streams, such as Unicode to UTF-8 and *vice versa*; and the removal of zero-width spaces. Support for map files is also provided through the *mapconvert* object, with mappings loaded from *GSDLHOME/mappings*.

text_t.h Primarily the Unicode text object described above. It also provides two classes for converting streams: *inconvertclass* and *outconvertclass*. These are the base classes used in *gsdlunicode.h.*

3.7 The collection server

Now we systematically explain all the objects in the conceptual framework of Figure 24. We start at the bottom of the diagram—which is also the foundations of the system—with Search, Source and Filter, and work our way up through the protocol layer and on to the central components in the receptionist: Actions, Format and Macro Language. Then we focus on object initialisation, since this is easier to understand once the role of the various objects is known.

Most of the classes central to the conceptual framework are expressed using virtual inheritance to aid extensibility. With virtual inheritance, inherited objects can be passed around as their base class, but when a member function is called it is the version defined in the inherited object that is invoked. By ensuring that the Greenstone source code uses the base class throughout, except at the point of object construction, this means that different implementations—using, perhaps, radically different underlying technologies—can be slotted into place easily.

For example, suppose a base class called *BaseCalc* provides basic arithmetic: add, subtract, multiply and divide. If all its functions are declared virtual, and arguments and return types are all declared as strings, we can easily implement inherited versions of the object. One, called *FixedPrecisionCalc*, might use C library functions to convert between strings and integers and back again, implementing the calculations using the standard arithmetic operators: +, –, *, and /. Another, called *InfinitePrecisionCalc*, might access the string arguments a character at a time, implementing arithmetic operations that are in principal infinite in their precision. By writing a main program that uses *BaseCalc* throughout, the implementation can be switched between fixed precision and infinite precision by editing just one line: the point where the calculator object is constructed.

The Search object

Figure 31 shows the base class API for the Search object in Figure 24. It defines two virtual member functions: *search()* and *docTargetDocument()*. As signified by the *=0* that follows the argument

Figure 31
The Search base class API

```
class searchclass {
public:
  searchclass ();
  virtual ~searchclass ();
  // the index directory must be set before any searching
  // is done
  virtual void setcollectdir (const text_t &thecollectdir);
  // the search results are returned in queryresults
  // search returns 'true' if it was able to do a search
  virtual bool search(const queryparamclass &queryparams,
             queryresultsclass &queryresults)=0;
  // the document text for 'docnum' is placed in 'output'
  // docTargetDocument returns 'true' if it was able to
  // try to get a document
  // collection is needed to see if an index from the
  // collection is loaded. If no index has been loaded
  // defaultindex is needed to load one
  virtual bool docTargetDocument(const text_t &defaultindex,
                        const text_t &defaultsubcollection,
                        const text_t &defaultlanguage,
                        const text_t &collection,
                        int docnum,
                        text_t &output)=0;
protected:
  querycache *cache;
  text_t collectdir; // the collection directory

};
```

declaration, these are *pure* functions—meaning that a class that inherits from this object must implement both (otherwise the compiler will complain).

The class also includes two protected data fields: *collectdir* and *cache*. A Search object is instantiated for a particular collection, and the *collectdir* field is used to store where on the file system that collection (and more importantly its index files) resides. The *cache* field retains the result of a query. This is used to speed up subsequent queries that duplicate the query (and its settings). While identical queries may seem unlikely, in fact they occur on a regular basis. The Greenstone protocol is stateless. To generate a results page like Figure 25 but for matches 11–20 of the same query, the search is transmitted again, this time specifying that documents 11–20 are returned. Caching makes this efficient, for the fact that the search has already been performed is detected and the results are lifted straight from the cache.

Both data fields are applicable to every inherited object that implements a searching mechanism. This is why they appear in the base class, and are declared within a protected section of the class so that inherited classes can access them directly.

Figure 32
API for direct access to MG (abridged)

```
enum result_kinds {
  result_docs,      // Return the documents found in last search
  result_docnums,   // Return document id numbers and weights
  result_termfreqs, // Return terms and frequencies
  result_terms      // Return matching query terms
};
int mgq_ask(char *line);
int mgq_results(enum result_kinds kind, int skip, int howmany,
                int (*sender)(char *, int, int, float, void *),
                void *ptr);
int mgq_numdocs(void);
int mgq_numterms(void);

int mgq_equivterms
             (unsigned char *wordstem,
              int (*sender)(char *, int, int, float, void *),
              void *ptr);

int mgq_docsretrieved (int *total_retrieved, int *is_approx);

int mgq_getmaxstemlen ();

void mgq_stemword (unsigned char *word);
```

Figure 33
The Source base class API

```
class sourceclass {
public:
  sourceclass ();
  virtual ~sourceclass ();

  // configure should be called once for each configuration line
  virtual void configure
                          (const text_t &key,
                            const text_tarray &cfgline);

  // init should be called after all the configuration is done but
  // before any other methods are called
  virtual bool init (ostream &logout);

  // translate_OID translates OIDs using ".pr", ."fc" etc.
  virtual bool translate_OID (const text_t &OIDin, text_t &OIDout,
                            comerror_t &err, ostream &logout);

  // get_metadata fills out the metadata if possible, if it is not
  // responsible for the given OID then it will return false.
  virtual bool get_metadata
                            (const text_t &requestParams,
                              const text_t &refParams,
                             bool getParents,
                             const text_tset &fields,
                             const text_t &OID,
                             MetadataInfo_tmap &metadata,
                             comerror_t &err, ostream &logout);

  virtual bool get_document (const text_t &OID, text_t &doc,
                             comerror_t &err, ostream &logout);
};
```

Search and retrieval with MG

Greenstone uses MG (short for Managing Gigabytes, see Witten *et al.*, 1999) to index and retrieve documents, and the source code is included in the *GSDLHOME/packages* directory. MG uses compression techniques to maximise disk space utilisation without compromising execution speed.

For a collection of English documents, the compressed text and full text indexes together typically occupy one third the space of the original uncompressed text alone. Search and retrieval is often quicker than the equivalent operation on the uncompressed version, because there are fewer disk operations.

MG is normally used interactively by typing commands from the command line, and one way to implement *mgsearchclass* would be to use the C library *system()* call within the object to issue the appropriate MG commands. A more efficient approach, however, is to tap directly into the MG code using function calls. While this requires a deeper understanding of the MG code, much of the complexity can be hidden behind a new API that becomes the point of contact for the object *mgsearchclass*. This is the role of *colserver/mgq.c*, whose API is shown in Figure 32.

The way to supply parameters to MG is via *mgq_ask()*, which takes text options in a format identical to that used at the command line, such as:

```
mgq_ask(".set casefold off");
```

It is also used to invoke a query. Results are accessed through *mgq_results*, which takes a pointer to a function as its fourth parameter. This provides a flexible way of converting the information returned in MG data structures into those needed by *mgsearchclass*. Calls such as *mgq_numdocs()*, *mgq_numterms()*, and *mgq_docsretrieved()* also return information, but this time more tightly prescribed. The last two give support for stemming.

The Source object

The role of Source in Figure 24 is to access document metadata and document text, and its base class API is shown in Figure 33. A member function maps to each task: *get_metadata()* and *get_document()* respectively. Both are declared *virtual*, so the version provided by a particular implementation of the base class is called at runtime. One inherited version of this object uses GDBM to implement *get_metadata()* and MG to implement *get_document()*: we detail this version below.

Other member functions seen in Figure 33 are *configure()*, *init()*, and *translate_OID()*. The first two relate to the initialisation process described in Section 3.10.

The remaining one, *translate_OID()*, handles the syntax for expressing document identifiers. In Figure 26 we saw how a page number could be appended to a document identifier to retrieve just that page. This was possible because pages were stored as “sections” when the collection was

Figure 34
The GDBM database for the Gutenberg collection (excerpt)

```
────────────────────────-
[HASH01d7b30d4827b51282919e9b]
<doctype>     doc
<hastxt>      0
<Title>       The Winter's Tale
<Creator>     William Shakespeare
<archivedir>  HASH01d7/b30d4827.dir
<thistype>    Invisible
<childtype>   Paged
<contains>
".1;".2;".3;".4;".5;".6;".7;".8;".9;".10;".11;".12;".13;".14; \

".15;".16;".17;".18;".19;".20;".21;".22;".23;".24;".25;".26;  \
              ".27;".28;".29;".30;".31;".32;".33;".34;".35
<docnum>      168483
────────────────────────-
[CL1]
<doctype>     classify
<hastxt>      0
<childtype>   HList
<Title>       Title
<numleafdocs> 1818
<thistype>    Invisible
<contains>
".1;".2;".3;".4;".5;".6;".7;".8;".9;".10;".11;".12;".13;".14; \
              ".15;".16;".17;".18;".19;".20;".21;".22;".23;".24
────────────────────────-
[CL1.1]
<doctype>     classify
<hastxt>      0
<childtype>   VList
<Title>       A
<numleafdocs> 118
<contains>  HASH0130bc5f9f90089b3723431f;HASH9cba43bacdab5263c98545; \
            HASH12c88a01da6e8379df86a7;HASH9c86579a83e1a2e4cf9736;   \
            HASHdc2951a7ada1f36a6c3aca;HASHea4dda6bbc7cdeb4abfdee;   \
            HASHce55006513c47235ac38ba;HASH012a33acaa077c0e612b9351; \
            HASH010dd1e923a123826ae30e4b;HASHaf674616785679fed4b7ee; \
            HASH0147eef4b9d1cb135e096619;HASHe69b9dbaa83ffb045d963b; \
            HASH01abc61c646c8e7a8ce88b10;HASH5f9cd13678e21820e32f3a; \
            HASHe8cbba1594c72c98f9aa1b;HASH01292a2b7b6b60dec96298bc; \
          ...
```

built. Appending “.1” to an OID retrieves the first section of the corresponding document. Sections can be nested, and are accessed by concatenating section numbers separated by periods.

As well as hierarchical section numbers, the document identifier syntax supports a form of relative access. For the current section of a document it is possible to access the *first child* by appending *.fc*, the *last child* by appending *.lc*, the *parent* by appending *.pr*, the *next sibling* by appending *.ns*, and the *previous sibling* by appending *.ps*.

The *translate_OID()* function uses parameters *OIDin* and *OIDout* to hold the source and result of the conversion. It takes two further parameters, *err* and *logout*. These communicate any error status that may arise during the translation operation, and determine where to send logging information. The parameters are closely aligned with the protocol, as we shall see in Section 3.8.

Figure 35
API for MG and GDBM based version of *sourceclass* (abridged)

```
class mggdbmsourceclass : public sourceclass {
protected:
  // Omitted, data fields that store:
  //   collection specific file information
  //   index substructure
  //   information about parent
  //   pointers to gdbm and mgsearch objects

public:
  mggdbmsourceclass ();
  virtual ~mggdbmsourceclass ();

  void set_gdbmptr (gdbmclass *thegdbmptr);
  void set_mgsearchptr (searchclass *themgsearchptr);

  void configure (const text_t &key, const text_tarray &cfgline);
  bool init (ostream &logout);
  bool translate_OID (const text_t &OIDin, text_t &OIDout,
                      comerror_t &err, ostream &logout);
  bool get_metadata (const text_t &requestParams,
                      const text_t &refParams,
                     bool getParents, const text_tset &fields,
                     const text_t &OID, MetadataInfo_tmap &metadata,
                     comerror_t &err, ostream &logout);
  bool get_document (const text_t &OID, text_t &doc,
                     comerror_t &err, ostream &logout);
};
```

DATABASE RETRIEVAL WITH GDBM

GDBM is the GNU database manager program. It implements a flat record structure of key/data pairs, and is backwards compatible with DBM and NDBM. Operations include storage, retrieval and deletion of records by key, and an unordered traversal of all keys.

Figure 34 shows an excerpt from the collection information database that is created when building the Gutenberg collection. The excerpt was produced using the Greenstone utility *db2txt*, which converts the GDBM binary database format into textual form. Figure 34 contains three records, separated by horizontal rules. The first record is a document entry, the other two are part of the hierarchy created by the *AZList* classifier for titles in the collection. The first line of each one is its key.

The document record stores the book’s title, author, and any other metadata provided (or extracted) when the collection was built. It also records values for internal use: where files associated with this document reside (<*archivedir*>) and the document number used internally by MG (<*docnum*>).

The field named <*contains*> stores a list of elements, separated by semicolons, that point to related records in the database. For a document record, <*contains*> is used to point to the nested sections. Subsequent record keys are formed by concatenating the current key with one of the child elements (separated by a period).

Figure 36
API for the Filter base class

```
class filterclass {
protected:
  text_t gsdlhome;
  text_t collection;
  text_t collectdir;

  FilterOption_tmap filterOptions;

public:
  filterclass ();
  virtual ~filterclass ();

  virtual void configure
                         (const text_t &key,
                           const text_tarray &cfgline);
  virtual bool init (ostream &logout);

  // returns the name of this filter
  virtual text_t get_filter_name ();

  // returns the current filter options
  virtual void get_filteroptions
                               (InfoFilterOptionsResponse_t &response,
                                comerror_t &err, ostream &logout);

  virtual void filter (const FilterRequest_t &request,
                       FilterResponse_t &response,
                       comerror_t &err, ostream &logout);
};
```

The second record in Figure 34 is the top node for the classification hierarchy of *Titles A–Z*. Its children, accessed through the *<contains>* field, include *CL1.1*, *CL1.2*, *CL1.3* and so on, and correspond to the individual pages for the letters *A*, *B*, *C* etc. There are only 24 children: the *AZList* classifier merged the *Q–R* and *Y–Z* entries because they covered only a few titles.

The children in the *<contains>* field of the third record, *CL1.1*, are the documents themselves. More complicated structures are possible—the *<contains>* field can include a mixture of documents and further *CL* nodes. Keys expressed relative to the current one are distinguished from absolute keys because they begin with a quotation mark (").

USING MG AND GDBM TO IMPLEMENT A SOURCE OBJECT

The object that puts MG and GDBM together to realise an implementation of *sourceclass* is *mggdbmsourceclass*. Figure 35 shows its API. The two new member functions *set_gdbmptr()* and *set_mgsearchptr()* store pointers to their respective objects, so that the implementations of *get_metadata()* and *get_document()* can access the appropriate tools to complete the job.

Figure 37
How a filter option is stored

```
struct FilterOption_t {
  void clear (); \  void check_defaultValue ();
  FilterOption_t () {clear();}

  text_t name;

  enum type_t {booleant=0, integert=1, enumeratedt=2, stringt=3};
  type_t type;

  enum repeatable_t {onePerQuery=0, onePerTerm=1, nPerTerm=2};
  repeatable_t repeatable;

  text_t defaultValue;
  text_tarray validValues;
};

struct OptionValue_t {
  void clear ();

  text_t name;
  text_t value;
};
```

The Filter object

The base class API for the Filter object in Figure 24 is shown in Figure 36. It begins with the protected data fields *gsdlhome*, *collection*, and *collectdir*. These commonly occur in classes that need to access collection-specific files.

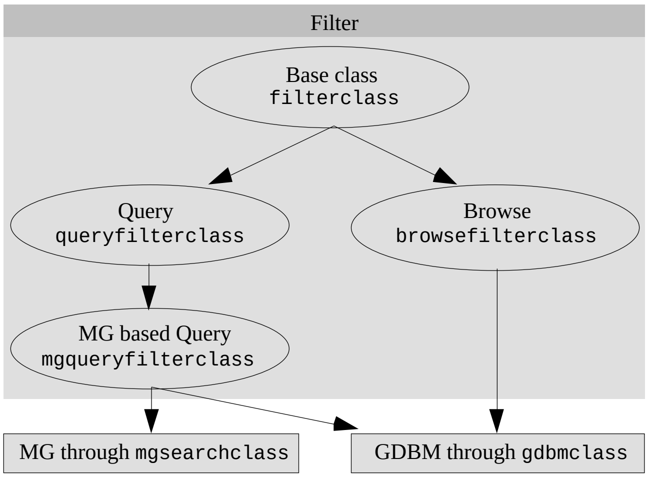
- *gsdlhome* is the same as *GSDLHOME*, so that the object can locate the Greenstone files.
- *collection* is the name of the directory corresponding to the collection.
- *collectdir* is the full pathname of the collection directory (this is needed because a collection does not have to reside within the *GSDLHOME* area).

mggdbsourceclass is another class that includes these three data fields.

The member functions *configure()* and *init()* (first seen in *sourceclass*) are used by the initialisation process. The object itself is closely aligned with the corresponding filter part of the protocol; in particular *get_filteroptions()* and *filter()* match one for one.

Central to the filter options are the two classes shown in Figure 37. Stored inside *FilterOption_t* is the *name* of the option, its *type*, and whether or not it is *repeatable*. The interpretation of *validValues* depends on the option type. For a Boolean type the first value is *false* and the second is *true*. For an integer type the first value is the minimum number, the second the maximum. For an enumerated type all values are listed. For a string type the value is ignored. For simpler situations, *OptionValue_t* is used, which records as a *text_t* the *name* of the option and its *value*.

Figure 38
The inheritance hierarchy for Filter



The request and response objects passed as parameters to *filterclass* are constructed from these two classes, using associative arrays to store a set of options such as those required for *InfoFilterOptionsResponse_t*. More detail can be found in *GSDLHOME/src/recpt/comtypes.h*.

Inherited Filter objects

Two levels of inheritance are used for filters, as illustrated in Figure 38. First a distinction is made between Query and Browse filters, and then for the former there is a specific implementation based on MG. To operate correctly, *mgqueryfilterclass* needs access to MG through *mgsearchclass* and to GDBM through *gdbmclass*. *browsefilterclass* only needs access to GDBM. Pointers to these objects are stored as protected data fields within the respective classes.

Files in *GSDLHOME/src/colservr*

Here are the header files in *GSDLHOME/src/colservr*, with a description of each. The filename generally repeats the object name defined within it.

browsefilter.h Inherited from *filterclass*, this object provides access to GDBM. (Described above.)

collectserver.h This object binds Filters and Sources for one collection together, to form the Collection object depicted in Figure 24.

colservrconfig.h Function support for reading the collection-specific files *etc/collect.cfg* and *index/build.cfg*. The former is the collection's configuration file. The

Table 18 List of protocol calls

Protocol call	Role
get_protocol_name()	Returns the name of this protocol. Choices include *nullproto*, *corbaproto*, and *z3950proto*. Used by protocol-sensitive parts of the runtime system to decide which code to execute.
get_collection_list()	Returns the list of collections that this protocol knows about.
has_collection()	Returns *true* if the protocol can communicate with the named collection, i.e. it is within its collection list.
ping()	Returns *true* if a successful connection was made to the named collection. In the null protocol the implementation is identical to *has_collection()*.
get_collectinfo()	Obtains general information about the named collection: when it was last built, how many documents it contains, and so on. Also includes metadata from the collection configuration file: "about this collection" text; the collection icon to use, and so on.
get_filterinfo()	Gets a list of all Filters for the named collection.
get_filteroptions()	Gets all options for a particular Filter within the named collection.
filter()	Supports searching and browsing. For a given filter type and option settings, it accesses the content of the named collections to produce a result set that is filtered in accordance with the option settings. The data fields returned also depend on the option settings: examples include query term frequency and document metadata.
get_document()	Gets a document or section of a document.

latter is a file generated by the building process that records the time of the last successful build, an index map list, how many documents were indexed, and how large they are in bytes (uncompressed).

filter.h The base class Filter object *filterclass* described above.

maptools.h Defines a class called *stringmap* that provides a mapping that remembers the original order of a *text_t* map, but is fast to look up. Used in *mggdbmsourceclass* and *queryfilterclass*.

mggdbmsource.h Inherited from *sourceclass*, this object provides access to MG and GDBM. (Described above.)

mgppqueryfilter.h Inherited from *queryfilterclass*, this object provides an implementation of *QueryFilter* based upon MG++, an improved version of MG written in C++. Note that Greenstone is set up to use MG by default, since MG++ is still under development.

mgppsearch.h Inherited from *searchclass*, this object provides an implementation of Search using MG++. Like *mgppqueryfilterclass*, this is not used by default.

Figure 39
Null protocol API (abridged)

```
class nullproto : public recptproto {
public:

  virtual text_t get_protocol_name ();

  virtual void get_collection_list (text_tarray &collist,
                     comerror_t &err, ostream &logout);

  virtual void has_collection (const text_t &collection,
                     bool &hascollection,
                     comerror_t &err, ostream &logout);

  virtual void ping (const text_t &collection,
                     bool &wassuccess,
                     comerror_t &err, ostream &logout);

  virtual void get_collectinfo (const text_t &collection,
                     ColInfoResponse_t &collectinfo,
                     comerror_t &err, ostream &logout);

  virtual void get_filterinfo (const text_t &collection,
                     InfoFiltersResponse_t &response,
                     comerror_t &err, ostream &logout);

  virtual void get_filteroptions (const text_t &collection,
                     const InfoFilterOptionsRequest_t &request,
                     InfoFilterOptionsResponse_t &response,
                     comerror_t &err, ostream &logout);


   virtual void filter (const text_t &collection,
                     FilterRequest_t &request,
                     FilterResponse_t &response,
                      comerror_t &err, ostream &logout);

  virtual void get_document (const text_t &collection,
                      const DocumentRequest_t &request,
                      DocumentResponse_t &response,
                      comerror_t &err, ostream &logout);
};
```

mgq.h Function-level interface to the MG package. Principal functions are *mg_ask()* and *mg_results()*.

mgqueryfilter.h Inherited from *queryfilterclass*, this object provides an implementation of *QueryFilter* based upon MG.

mgsearch.h Inherited from *searchclass*, this object provides an implementation of Search using MG. (Described above.)

phrasequeryfilter.h Inherited from *mgqueryclass*, this object provides a phrase-based query class. It is not used in the default installation. Instead *mgqueryfilterclass* provides this capability through functional support from *phrasesearch.h*.

phrasesearch.h Functional support to implement phrase searching as a post-processing operation.

querycache.h Used by *searchclass* and its inherited classes to cache the results of a

Table 19 Actions in Greenstone

Action	Role
action	Base class for virtual inheritance.
authenaction	Supports user authentication: prompts the user for a password if one has not been entered; checks whether it is valid; and forces the user to log in again if sufficient time lapses between accesses.
collectoraction	Generates the pages for the Collector.
documentaction	Retrieves documents, document sections, parts of the classification hierarchy, or formatting information.
extlinkaction	Takes a user directly to a URL that is external to a collection, possibly generating an alert page first (dictated by the Preferences).
pageaction	Generates a page in conjunction with the macro language.
pingaction	Checks to see whether a collection is online.
queryaction	Performs a search.
statusaction	Generates the administration pages.
tipaction	Brings up a random tip for the user.
usersaction	Supports adding, deleting, and managing user access.

query, in order to make the generation of further search results pages more efficient. (Described above.)

queryfilter.h Inherited from the Filter base class *filterclass*, this object establishes a base class for Query filter objects. (Described above.)

queryinfo.h Support for searching: data structures and objects to hold query parameters, document results and term frequencies.

search.h The base class Search object *searchclass*. (Described above.)

source.h The base class Source object *sourceclass*. (Described above.)

3.8 The protocol

Table 18 lists the function calls to the protocol, with a summary for each entry. The examples in Section 3.3 covered most of these. Functions not previously mentioned are *has_collection()*, *ping()*, *get_protocol_name()* and *get_filteroptions()*. The first two provide yes/no answers to the questions “does the collection exists on this server?” and “is it running?” respectively. The purpose of the other two is to support multiple protocols within an architecture that is distributed over different computers, not just the null-protocol based single executable described here. One of these distinguishes which protocol is being used. The other lets a receptionist interrogate a collection server to find what options are supported, and so

Figure 40
Using the *cgiargsinfoclass* from *pageaction.cpp*

```
cgiarginfo arg_ainfo;
arg_ainfo.shortname = “a”;
arg_ainfo.longname = “action”;
arg_ainfo.multiplechar = true;
arg_ainfo.argdefault = “p”;
arg_ainfo.defaultstatus = cgiarginfo::weak;
arg_ainfo.savedarginfo = cgiarginfo::must;
argsinfo.addarginfo (NULL, arg_ainfo);

arg_ainfo.shortname = “p”;
arg_ainfo.longname = “page”;
arg_ainfo.multiplechar = true;
arg_ainfo.argdefault = “home”;
arg_ainfo.defaultstatus = cgiarginfo::weak;
arg_ainfo.savedarginfo = cgiarginfo::must;
argsinfo.addarginfo (NULL, arg_ainfo);
```

dynamically configure itself to take full advantage of the services offered by a particular server.

Figure 39 shows the API for the null protocol. Comments, and certain low level details, have been omitted (see the source file *recpt/nullproto.h* for full details).

This protocol inherits from the base class *recptproto*. Virtual inheritance is used so that more than one type of protocol—including protocols not even conceived yet—can be easily supported in the rest of the source code. This is possible because the base class object *recptproto* is used throughout the source code, with the exception of the point of construction. Here we specify the actual variety of protocol we wish to use—in this case, the null protocol.

With the exception of *get_protocol_name()*, which takes no parameters and returns the protocol name as a Unicode-compliant text string, all protocol functions include an error parameter and an output stream as the last two arguments. The error parameter records any errors that occur during the execution of the protocol call, and the output stream is for logging purposes. The functions have type *void*—they do not explicitly return information as their final statement, but instead return data through designated parameters such as the already-introduced error parameter. In some programming languages, such routines would be defined as procedures rather than functions, but C++ makes no syntactic distinction.

Most functions take the collection name as an argument. Three of the member functions, *get_filteroptions()*, *filter()*, and *get_document()*, follow the pattern of providing a Request parameter and receiving the results in a Response parameter.

Figure 41
Action base class API

```
class action {
protected:
  cgiargsinfoclass argsinfo;
  text_t gsdlhome;

public:
  action ();
  virtual ~action ();

  virtual void configure (const text_t &key,
                          const text_tarray &cfgline);
  virtual bool init (ostream &logout);

  virtual text_t get_action_name ();
  cgiargsinfoclass getargsinfo ();

  virtual bool check_cgiargs (cgiargsinfoclass &argsinfo,
                         cgiargsclass &args,
                         ostream &logout);
  virtual bool check_external_cgiargs (cgiargsinfoclass &argsinfo,
                         cgiargsclass &args,
                         outconvertclass &outconvert,
                         const text_t &saveconf,
                         ostream &logout);
  virtual void get_cgihead_info (cgiargsclass &args,
                         recptprotolistclass *protos,
                         response_t &response,
                         text_t &response_data,
                         ostream &logout);
   virtual bool uses_display (cgiargsclass &args);

  virtual void define_internal_macros (displayclass &disp,
                         cgiargsclass &args,
                         recptprotolistclass *protos,
                         ostream &logout);
  virtual void define_external_macros (displayclass &disp,
                         cgiargsclass &args,
                         recptprotolistclass *protos,
                         ostream &logout);

  virtual bool do_action (cgiargsclass &args,
                         recptprotolistclass *protos,
                         browsermapclass *browsers,
                         displayclass &disp,
                         outconvertclass &outconvert,
                         ostream &textout,
                         ostream &logout);
};
```

3.9 The receptionist

The final layer of the conceptual model is the receptionist. Once the CGI arguments are parsed, the main activity is the execution of an Action, supported by the Format and Macro Language objects. These are described below. Although they are represented as objects in the conceptual framework, Format and Macro Language objects are not strictly objects in the C++ sense. In reality, Format is a collection of data structures with a set of functions that operate on them, and the Macro Language object is built around *displayclass*, defined in *lib/display.h*, with stream conversion support from *lib/gsdlunicode.h*.

Figure 42
Core data structures in Format

```
enum command_t {comIf, comOr, comMeta, comText, comLink, comEndLink,
                                       comNum, comIcon, comDoc,
                                       comHighlight, comEndHighlight};
enum pcommand_t {pNone, pImmediate, pTop, pAll};
enum dcommand_t {dMeta, dText};
enum mcommand_t {mNone, mCgiSafe};

struct metadata_t {
  void clear();
  metadata_t () {clear();}

 text_t metaname;
  mcommand_t metacommand;
  pcommand_t parentcommand;
  text_t parentoptions;
};

// The decision component of an {If}{decision,true-text,false-text}
// formatstring. The decision can be based on metadata or on text;
// normally that text would be a macro like
// _cgiargmode_.

struct decision_t {
  void clear();
  decision_t () {clear();}

  dcommand_t command;
  metadata_t meta;
  text_t text;
};

struct format_t {
  void clear();
  format_t () {clear();}

  command_t command;
  decision_t decision;
  text_t text;
  metadata_t meta;
  format_t *nextptr;
  format_t *ifptr;
  format_t *elseptr;
  format_t *orptr;
};
```

Actions

Greenstone supports the eleven actions summarised in Table 19.

The CGI arguments needed by an action are formally declared in its constructor function using *cgiarginfo* (defined in *recpt/cgiargs.h*). Figure 40 shows an excerpt from the *pageaction* constructor function, which defines the size and properties of the CGI arguments *a* and *p*.

For each CGI argument, the constructor must specify its short name (lines 2 and 10), which is the name of the CGI variable itself; a long name (lines 3 and 11) that is used to provide a more meaningful description of the action; whether it represents a single or multiple character value (lines 4 and 12); a possible default value (lines 5 and 13); what happens when more than one default value is supplied (lines 6 and 14) (since defaults

Figure 43
Data structures built for sample *format* statement

can also be set in configuration files); and whether or not the value is preserved at the end of this action (lines 7 and 15)

Since it is built into the code, web pages that detail this information can be generated automatically. The *statusaction* produces this information. It can be viewed by entering the URL for the Greenstone administration pages, which is *http://localhost/gsdl/cgi-bin/library?a=status&p=frameset.*

The twelve inherited actions are constructed in *main()*, the top-level function for the *library* executable, whose definition is given in *recpt/librarymain.cpp*. This is also where the *receptionist* object (defined in *recpt/receptionist.cpp*) is constructed. Responsibility for all the actions is passed to the *receptionist*, which processes them by maintaining, as a data field, an associative array of the Action base class, indexed by action name.

Figure 41 shows the API for the Action base class. When executing an action, *receptionist* calls several functions, starting with *check_cgiargs()*. Most help to check, set up, and define values and macros; while *do_action()* actually generates the output page. If a particular inherited object has no definition for a particular member function, it passes through to the base class definition which implements appropriate default behaviour.

Explanations of the member functions are as follows.

- *get_action_name()* returns the CGI *a* argument value that specifies this action. The name should be short but may be more than one character long.
- *check_cgiargs()* is called before *get_cgihead_info()*, *define_external_macros()*, and *do_action()*. If an error is found a message will be written to *logout*; if it is serious the function returns *false* and no page content is produced.
- *check_external_cgiargs()* is called after *check_cgiargs()* for all

Figure 44
Illustration of macro precedence

```
package query
_header_ []               {_querytitle_}
_header_ [l=en]           {Search page}
_header_ [c=demo]         {<table
bgcolor=green><tr><td>_querytitle_</td></tr></table>}
_header_ [v=1]            {_textquery_}
_header_ [l=fr,v=1,c=hdl] {HDL Page de recherche}
```

actions. It is intended for use only to override some other normal behaviour, for example producing a login page when the requested page needs authentication.

- *get_cgihead_info()* sets the CGI header information. If *response* is set to *location*, then *response_data* contains the redirect address. If *response* is set to *content*, then *response_data* contains the content type.
- *uses_display()* returns *true* if the *displayclass* is needed to output the page content (the default).
- *define_internal_macros()* defines all macros that are related to pages generated by this action.
- *define_external_macros()* defines all macros that might be used by other actions to produce pages.
- *do_action()* generates the output page, streamed through the macro language object *display* and the output conversion object *textout*. Returns *false* if there was an error which prevented the action from outputting anything.

At the beginning of the class definition, *argsinfo* is the protected data field (used in the code excerpt shown in Figure 40) that stores the CGI argument information specified in an inherited Action constructor function. The other data field, *gsdlhome*, records *GSDLHOME* for convenient access.[2] The object also includes *configure()* and *init()* for initialisation purposes.

Formatting

Although formatting is represented as a single entity in Figure 24, in reality it constitutes a collection of data structures and functions. They are gathered together under the header file *recpt/formattools.h*. The core data structures are shown in Figure 42.

The implementation is best explained using an example. When the format statement

[2] The value for *gsdlhome* comes from *gsdlsite.cfg* located in the same directory as the CGI executable `library`, whereas `GSDLHOME` is set by running the *setup* script which accesses a different file, so technically it is possible for the two values to be different. While possible, it is not desirable, and the above text is written assuming they are the same.

Figure 45
Data structures that represent the default macros

```
format CL1VList "[link][Title]{If}{[Creator], by
[Creator]}[/link]}"
```

is read from a collection configuration file, it is parsed by functions in *formattools.cpp* and the interconnected data structure shown in Figure 43 is built. When the format statement needs to be evaluated by an action, the data structure is traversed. The route taken at *comIf* and *comOr* nodes depends on the metadata that is returned from a call to the protocol.

One complication is that when metadata is retrieved, it might include further macros and format syntax. This is handled by switching back and forth between parsing and evaluating, as needed.

Macro language

The Macro Language entity in Figure 24, like Format, does not map to a single C++ class. In this case there is a core class, but the implementation of the macro language also calls upon supporting functions and classes.

Again, the implementation is best explained using an example. First we give some sample macro definitions that illustrate macro precedence, then—with the aid of a diagram—we describe the core data structures built to support this activity. Finally we present and describe the public member functions to *displayclass*, the top-level macro object.

In a typical Greenstone installation, macro precedence is normally: *c* (for the collection) takes precedence over *v* (for graphical or text-only interface), which takes precedence over *l* (for the language). This is accomplished by the line

```
macroprecedence c,v,l
```

Figure 46
Displayclass API
(abridged)

```
class displayclass
{
public:
  displayclass ();
  ~displayclass ();

  int isdefaultmacro (text_t package, const text_t &macroname);
  int setdefaultmacro (text_t package, const text_t &macroname,
                       text_t params, const text_t &macrovalue);
  int loaddefaultmacros (text_t thisfilename);

  void openpage (const text_t &thispageparams,
                 const text_t &thisprecedence);
  void setpageparams (text_t thispageparams,
                      text_t thisprecedence);

  int setmacro (const text_t &macroname,
                text_t package,
                const text_t &macrovalue);

  void expandstring (const text_t &inputtext, text_t &outputtext);
  void expandstring (text_t package, const text_t &inputtext,
                     text_t &outputtext, int recursiondepth = 0);

  void setconvertclass (outconvertclass *theoutc) {outc = theoutc;}
  outconvertclass *getconvertclass () {return outc;}
  ostream *setlogout (ostream *thelogout);
};
```

in the main configuration file *main.cfg*. The macro statements in Figure 44 define sample macros for *_header_* in the *query* package for various settings of *c*, *v*, and *l*. If the CGI arguments given when an action is invoked included *c=hdl*, *v=1*, and *l=en*, the macro *_header_[v=1]* would be selected for display. It would be selected ahead *of _content_[l=en]* because *v* has a higher precedence than *l*. The *_content_[l=fr,v=1,c=hdl]* macro would not be selected because the page parameter for *l* is different.

Figure 45 shows the core data structure built when reading the macro files specified in *etc/main.cfg*. Essentially, it is an associative array of associative arrays of associative arrays. The top layer (shown on the left) indexes which package the macro is from, and the second layer indexes the macro name. The final layer indexes any parameters that were specified, storing each one as the type *mvalue* which records, along with the macro value, the file it came from. For example, the text defined for *_header_[l=en]* in Figure 44 can be seen stored in the lower of the two *mvalue* records in Figure 45.

The central object that supports the macro language is *displayclass*, defined in *lib/display.h*. Its public member functions are shown in Figure 46. The class reads the specified macro files using *loaddefaultmacros()*, storing in a protected section of the class (not shown) the type of data structure shown in Figure 45. It is also permissible for macros to be set by the runtime system using *setmacro()* (in an example of Section 3.2, *pageaction* set *_collectionextra_* to be the dynamically generated table of available collections using this function.) This is supported by a set of associative arrays similar to those used to represent macro files (it is not identical, because the former does not require the “parameter” layer). In *displayclass*, macros read from the file are referred to as *default macros*. Local macros specified through *setmacro()* are referred to as *current macros*, and are cleared from memory once the page has been generated.

When a page is to be produced, *openpage()* is first called to communicate the current settings of the page parameters (*l=en* and so on). Following that, text and macros are streamed through the class—typically from within an *actionclass*—using code along the lines of:

```
cout << text_t2ascii << display << "_amacro_"
                                << "_anothermacro_";
```

The result is that macros are expanded according to the page parameter settings. If required, these settings can be changed partway through an action by using *setpageparams()*. The remaining public member functions provide lower level support.

Files in *GSDLHOME/src/recpt*

The principal objects in the receptionist have now been described. Below we detail the supporting classes. Except where efficiency is paramount—in which case definitions are in-line—implementation details are contained within a header file’s *.cpp* counterpart. Supporting files often include the word *tool* as part of the file name, as in *OIDtools.h* and *formattools.h*.

A second set of lexically scoped files include the prefix *z3950*. The files provide remote access to online databases and catalogs that make their content publicly available using the Z39.50 protocol.

Another large group of supporting files include the term *browserclass*. These files are related through a virtual inheritance hierarchy. As a group they support an abstract notion of browsing: serial page generation of compartmentalised document content or metadata. Browsing activities include perusing documents ordered alphabetically by title or chronologically by date; progressing through the titles returned by a query ten entries at a time; and accessing individual pages of a book using the “go to page” mechanism. Each browsing activity inherits from *browserclass*, the base class:

- *datelistbrowserclass* provides support for chronological lists;
- *hlistbrowserclass* provides support for horizontal lists;
- *htmlbrowserclass* provides support for pages of HTML;
- *invbrowserclass* provides support for invisible lists;

- *pagedbrowserclass* provides go to page support;
- *vlistbrowserclass* provides support for vertical lists.

Actions access *browserclass* objects through *browsetools.h*.

OIDtools.h Function support for evaluating document identifiers over the protocol.

action.h Base class for the Actions entity depicted in Figure 24.

authenaction.h Inherited action for handling authentication of a user.

browserclass.h Base class for abstract browsing activities.

browsetools.h Function support that accesses the *browserclass* hierarchy. Functionality includes expanding and contracting contents, outputing a table of contents, and generating control such as the "go to page" mechanism.

cgiargs.h Defines *cgiarginfo* used in Figure 40, and other data structure support for CGI arguments.

cgiutils.h Function support for CGI arguments using the data structures defined in *cgiargs.h*.

cgiwrapper.h Function support that does everything necessary to output a page using the CGI protocol. Access is through the function

```
void cgiwrapper (receptionist &recpt, text_t collection);
```

which is the only function declared in the header file. Everything else in the *.cpp* counterpart is lexically scoped to be local to the file (using the C++ keyword *static*). If the function is being run for a particular collection then *collection* should be set, otherwise it should be the empty string "". The code includes support for Fast-CGI.

collectoraction.h Inherited action that facilitates end-user collection-building through the Collector. The page generated comes from *collect.dm* and is controlled by the CGI argument *p=page*.

comtypes.h Core types for the protocol.

converter.h Object support for stream converters.

datelistbrowserclass.h Inherited from *browserclass*, this object provides browsing support for chronological lists such as that seen in the Greenstone Archives collection under "dates" in the navigation bar.

documentaction.h Inherited action used to retrieve a document or part of a classification hierarchy.

extlinkaction.h Inherited action that controls whether or not a user goes straight to an external link or passes through a warning page alerting the user to the fact that they are about to move outside the digital library system.

formattools.h Function support for parsing and evaluating collection configuration *format* statements. Described in more detail in Section 3.9.2 above.

historydb.h Data structures and function support for managing a database of previous queries so a user can start a new query that includes previous query terms.

hlistbrowserclass.h Inherited from *browserclass*, this object provides browsing support for horizontal lists.

htmlbrowserclass.h Inherited from *browserclass*, this object provides browsing support for HTML pages.

htmlgen.h Function support to highlight query terms in a *text_t* string.

htmlutils.h Function support that converts a *text_t* string into the equivalent HTML. The symbols ", &, <, and > are converted into *&quot;*, *&amp;*, *&lt;* and *&gt;* respectively.

infodbclass.h Defines two classes: *gdbmclass* and *infodbclass*. The former provides the Greenstone API to GDBM; the latter is the object class used to store a record entry read in from a GDBM database, and is essentially an associative array of integer-indexed arrays of *text_t* strings.

invbrowserclass.h Inherited from *browserclass*, this object provides browsing support for lists that are not intended for display (invisible).

nullproto.h Inherited from *recptproto*, this class realises the null protocol, implemented through function calls from the receptionist to the collection server.

pageaction.h Inherited action that, in conjunction with the macro file named in *p=page*, generates a web page.

pagedbrowserclass.h Inherited from *browserclass*, this object provides browsing support for the “go to page” mechanism seen (for example) in the Gutenberg collection.

pingaction.h Inherited action that checks to see whether a particular collection is responding.

queryaction.h Inherited action that takes the stipulated query, settings and preferences and performs a search, generating as a result the subset of *o=num* matching documents starting at position *r=num*.

querytools.h Function support for querying.

receptionist.h Top-level object for the receptionist. Maintains a record of CGI argument information, instantiations of each inherited action, instantiations of each inherited browser, the core macro language object *displayclass*, and all possible converters.

recptconfig.h Function support for reading the site and main configuration files.

recptproto.h Base class for the protocol.

statusaction.h Inherited action that generates, in conjunction with *status.dm*, the various administration pages.

tipaction.h Inherited action that produces, in conjunction with *tip.dm*, a web page containing a tip taken at random from a list of tips stored in *tip.dm*.

userdb.h Data structure and function support for maintaining a GDBM database of users: their password, groups, and so on.

usersaction.h An administrator action inherited from the base class that supports adding and deleting users, as well as modifying the groups they are in.

vlistbrowserclass.h Inherited from *browserclass*, this object provides browsing support for vertical lists, the mainstay of classifiers. For example, the children of the node for titles beginning with the letter *N* are stipulated to be a *VList*.

z3950cfg.h Data structure support for the Z39.50 protocol. Used by *z3950proto.cpp*, which defines the main protocol class (inherited from the base class *recptproto*), and configuration file parser *zparse.y* (written using YACC).

z3950proto.h Inherited from *recptproto,* this class realises the Z39.50 protocol so that a Greenstone receptionist can access remote library sites running Z39.50 servers.

z3950server.h Further support for the Z39.50 protocol.

3.10Initialisation

Initialisation in Greenstone is an intricate operation that processes configuration files and assigns default values to data fields. In addition to inheritance and constructor functions, core objects define *init()* and *configure()* functions to help standardise the task. Even so, the order of execution can be difficult to follow. This section describes what happens.

Greenstone uses several configuration files for different purposes, but all follow the same syntax. Unless a line starts with the hash symbol (#) or consists entirely of white space, the first word defines a keyword, and the remaining words represent a particular setting for that keyword.

The lines from configuration files are passed, one at a time, to *configure()* as two arguments: the keyword and an array of the remaining words. Based on the keyword, a particular version of *configure()* decides whether the information is of interest, and if so stores it. For example, *collectserver* (which maps to the Collection object in Figure 24) processes the format statements in a collection's configuration file. When the keyword *format* is passed to *configure()*, an *if* statement is triggered that stores in the object a copy of the function's second argument.

After processing the keyword and before the function terminates, some versions of *configure()* pass the data to *configure()* functions in other objects. The Receptionist object calls *configure()* for Actions, Protocols, and Browsers. The NullProtocol object calls *configure()* for each Collection object; Collection calls Filters and Sources.

In C++, data fields are normally initialized by the object's constructor function. However, in Greenstone some initialisation depends on values read from configuration files, so a second round of initialisation is needed. This is the purpose of the *init()* member functions, and in some cases it leads to further calls to *configure()*.

Figure 47 shows diagnostic statements generated from a version of Greenstone augmented to highlight the initialisation process. The program starts in the *main()* function in *recpt/librarymain.cpp*. It constructs a Receptionist object and a NullProtocol object, then scans *gsdlsite.cfg* (located in the same directory as the *library* executable) for *gsdlhome* and stores its value in a variable. For each online collection—as established by reading in the directories present in *GSDLHOME/collect*—it constructs a Collection object, through the NullProtocol object, that includes within it Filters, Search and Source, plus few hardwired calls to

Figure 47
Initialising Greenstone using the null protocol

```
============
Main program
============
Statically construct Receptionist
Statically construct NullProtocol
Establish the value for 'gsdlhome' by reading gsdlsite.cfg
Foreach directory in GSDLHOME/collect that isn't "modelcol":
  Add directory name (now treated as collection name) to NullProtocol:
    Dynamically construct Collection
    Dynamically construct Gdbm class
    Dynamically construct the Null Filter
    Dynamically construct the Browse Filter
    Dynamically construct MgSearch
    Dynamically construct the QueryFilter
    Dynamically construct the MgGdbmSource
    Configure Collection with 'collection'
      Passing 'collection' value on to Filters and Sources:
    Configure Receptionist with 'collectinfo':
      Passing 'collectinfo' value on to Actions, Protocols, and Browsers:

Add NullProtocol to Receptionist
Add in UTF-8 converter
Add in GB converter
Add in Arabic converter

Foreach Action:
  Statically construct Action
  Add Action to Receptionist
Foreach Browsers:
  Statically construct Browser
  Add Browser to Receptionist

Call function cgiwrapper:
  =================
  Configure objects
  =================
  Configure Receptionist with 'collection'
    Passing 'collection' value on to Actions, Protocols, and Browsers:
    NullProtocol not interested in 'collection'
  Configure Receptionist with 'httpimg'
    Passing 'httpimg' value on to Actions, Protocols, and Browsers:
    NullProtocol passing 'httpimg' on to Collection
    Passing 'httpimg' value on to Filters and Sources:
  Configure Receptionist with 'gwcgi'
    Passing 'gwcgi' value on to Actions, Protocols, and Browsers:
    NullProtocol passing 'gwcgi' on to Collection
      Passing 'gwcgi' value on to Filters and Sources:

  Reading in site configuration file gsdlsite.cfg
    Configure Recptionist with 'gsdlhome'
      Passing 'gsdlhome' value on to Actions, Protocols, and Browsers:
      NullProtocol passing 'gsdlhome' on to Collection
        Passing 'gsdlhome' value on to Filters and Sources:
    Configure Recptionist with ...
    ... and so on for all entries in gsdlsite.cfg
  Reading in main configuration file main.cfg
    Confiugre Recptionist with ...
    ... and so on for all entries in main.cfg

  ====================
  Initialising objects
  ====================
  Initialise the Receptionist
    Configure Receptionist with 'collectdir'
      Passing 'collectdir' value on to Actions, Protocols, and Browsers:
      NullProtocol not interested in 'collectdir'
    Read in Macro files
    Foreach Actions
      Initialise Action

    Foreach Protocol
      Initialise Protocol

      When Protocol==NullProtocol:
        Foreach Collection
          Reading Collection's build.cfg
          Reading Collection's collect.cfg
            Configure Collection with 'creator'
              Passing 'creator' value on to Filters and Sources:
            Configure Collection with 'maintainer'
              Passing 'maintainer' value on to Filters and Sources:
            ... and so on for all entries in collect.cfg

    Foreach Browsers
      Initialise Browser

  =============
  Generate page
  =============
  Parse CGI arguments
  Execute designated Action to produce page
 End.
```

configure().

Next *main()* adds the NullProtocol object to the Receptionist, which keeps a base class array of protocols in a protected data field, and then sets up several converters. *main()* constructs all Actions and Browsers used in the executable and adds them to the Receptionist. The function concludes by calling *cgiwrapper()* in *cgiwrapper.cpp*, which itself includes substantial object initialisation.

There are three sections to *cgiwrapper()*: configuration, initialisation and page generation. First some hardwired calls to *configure()* are made. Then *gsdlsite.cfg* is read and *configure()* is called for each line. The same is done for *etc/main.cfg*.

The second phase of *cgiwrapper()* makes calls to *init()*. The Receptionist makes only one call to its *init()* function, but the act of invoking this calls *init()* functions in the various objects stored within it. First a hardwired call to *configure()* is made to set *collectdir*, then the macro files are read. For each action, its *init()* function is called. The same occurs for each protocol stored in the receptionist—but in the system being described only one protocol is stored, the NullProtocol. Calling *init()* for this object causes further configuration: for each collection in the NullProtocol, its collection-specific *build.cfg* and *collect.cfg* are read and processed, with a call to *configure()* for each line.

The final phase of *cgiwrapper()* is to parse the CGI arguments, and then call the appropriate action. Both these calls are made with the support of the Receptionist object.

The reason for the separation of the configuration, initialisation, and page generation code is that Greenstone is optimised to be run as a server (using Fast-cgi or the Corba protocol). In this mode of operation, the configuration and initialisation code is executed once, then the program remains in memory and generates many web pages in response to requests from clients, without requiring re-initalisation.

4 Configuring your Greenstone site

There are two configuration files in Greenstone that are used for configuring various aspects of your Greenstone site. These files are *main.cfg*, found in *GSDLHOME/etc*, and *gsdlsite.cfg* found in *GSDLHOME/cgi-bin*. These files each control specific aspects of site-wide configuration, and are quite powerful. Both of these files can be viewed from the Greenstone administration page.

4.1 main.cfg

This file is used to configure the receptionist—that part of Greenstone that fields queries and displays pages. With this file you can control everything from the languages that the interface can use to what logs are kept.

4.2 Site maintenance and logging

There are certain options that control how your Greenstone site is maintained and whether there is a nominated maintainer. Table 20 describes the lines in the configuration file used to control these things.

4.3 Language support

There are two different lines in the *main.cfg* configuration file that allow for support of different languages. One is the *Encoding* line, which includes a specific type of character encoding, the other is the *Language* line, which includes a user interface language as an option for the user interface (provided that there is a language macro for that language). These lines determine which languages and encodings will be available from the preferences page.

Table 20 Configuration options for site maintenance and logging

Directory	Contents	Section
maintainer	NULL or an E-mail address	The E-mail address of the site maintainer to be used for certain notification purposes. If this is set to NULL, E-mail events (see below) are disabled
MailServer	NULL or a server name	An outgoing mail server for this Greenstone site. If this is not set it will resolve to *mail.maintainer's-domain* (so for example if the maintainer is *help@example.com* the server will default to *mail.example.com*. If this doesn't resolve to a valid SMTP server, E-mail events will not work
status	*enabled* or *disabled*	Determines whether the "Maintenance and administration" page is to be made available or not
collector	*enabled* or *disabled*	Determine whether or not the end user collection building "collector" facility will be available or not
logcgiargs	*true* or *false*	If set to *true*, a log of the usage of the site will be kept in *usage.txt*.
usecookies	*true* or *false*	If set to *true*, cookies will collect information about the site users. The information collected will be written to *usage.txt* (this only works if *logcgiargs* is *true*)
LogDateFormat	*LocalTime* or *UTCTime* or *Absolute*	The format for the time information to be written to the log. *LocalTime* will produce the format "Thu Dec 07 12:34 NZDT 2000", *UTCTime* is the same format but in GMT time, and *absolute* is an integer representing the number of seconds since 00:00:00 01/01/1970 GMT
LogEvents	*AllEvents* or *CollectorEvents* or *disabled*	Write a log of certain events to *events.txt*. Currently only collector events (such as creating and deleting collections) are enabled. *AllEvents* will log all Greenstone events, *CollectorEvents* will log only events to do with the collector and disabled won't log any events
EmailEvents	*enabled* or *disabled*	E-mail the maintainer (if there is one—see the maintainer option) every time an event occurs
EmailUserEvents	*enabled* or *disabled*	E-mail the user on certain events—such as the collector finishing a collection build
macrofiles	a list of macro filenames	Determine what macros are available for Greenstone's user interface software

The *Encoding* line has four options: *shortname*, *longname*, *map* and *multibyte*. The *shortname* option is mandatory for all encodings, it is the standard charset label. The *longname* option is the encoding name displayed on the preferences page. It is not mandatory, and will default to the *shortname* if not present. The *map* encoding is mandatory for all encodings except utf8, which is handled internally (and should always be enabled. The *multibyte* option should be set for all character sets that require more than one byte per character. There are many encodings listed in *main.cfg*, many of which are commented out with a preceding #. To enable an encoding, remove the #.

The *Language* line has three options, *shortname*, *longname*, and *default_encoding*. The *shortname* option is the ISO 639 two-letter language symbol, and it is a required option. The *longname* option is the display name for the language on the preferences page, and like the *Encoding longname* option is not compulsory amd will default to the *shortname* of the language if it is not present. The *default_encoding* option is used to define the encoding preferred for this language.

4.4 Page parameters and CGI arguments

Page parameters and CGI arguments may be defined from within the *main.cfg* configuration file. Recall from Figure 40 that most CGI arguments are defined from within the library C++ code itself. However, it is occasionally useful to define new arguments or edit existing ones at configuration time (thus avoiding the need to recompile the library).

To do this you use the *cgiarg* configuration option. *Cgiarg* may take up to six arguments; *shortname*, *longname*, *multiplechar*, *argdefault*, *defaultstatus* and *savedarginfo*. These arguments correspond to the CGI argument options described in Section 3.9. As an example, in Greenstone's default *main.cfg* file, the *cgiarg* configuration option is used to set the default values of the existing *a* and *p* CGI arguments to *p* and *home* respectively.

Page parameters are special cases of CGI arguments which correspond to parameters in Greenstone's macro files. For example, the *l* CGI argument directly corresponds to the *l*= parameter in the macro files. To define a CGI argument as also being a page parameter you use the *pageparam* configuration option.

The best way to learn about the various configuration options accepted in the *main-cfg* configuration file is to experiment with the file itself. Note that if you are using the Windows local library version of Greenstone you must restart the server before any changes made to configuration files will

Table 21 Lines in *gsdlsite.cfg*

Line	**Function**
gsdlhome	A path to the *GSDLHOME* directory.
httpprefix	The web address of *GSDLHOME*. If the document root on your web server is set to *GSDLHOME* you will not need this line.
httpimage	The web address of the directory containing the images for the user interface. If your web-server's dpcument root is set to *GSDLHOME* this will be */images*.
gwcgi	The web address of this cgi script (usually ends in *library*). This is not needed if your web server sets the environment variable *SCRIPT_NAME*.
maxrequests	(Only applies if *fast-cgi* is in use.) The number of requests fast-cgi should process before it exits. If you are debugging the library this should be set to a small number, if everything is fine it should be a large number.

take effect.

4.5 gsdlsite.cfg

This file sets variables that will be used by the library software and web-server at run-time. It should be in the same directory as the *library* program. Table 21 describe the lines used in this file.

Appendix A: The C++ Standard Template Library

As the word “template” suggests, the Standard Template Library (STL) is not just a plug-and-use object library. Coupled with the template mechanism in C++, it provides a forum for programmers to concisely develop their own objects that tap into the algorithmic capabilities embedded within STL. This adds an extra layer of complexity, but it’s worth it.

To help understand the Greenstone code excerpts given in this manual, we give a few tutorial level examples that use STL.

STL lists

First we study two programs that implement an integer list. One uses basic C++ types (the “old fashioned” way), the other uses STL. Figure 48 shows the source code implementation that does *not* use STL. Lines 5–8 define the basic data structure we are going to use: the field *val* stores the integer value, and *next* points to the next element in the list—a classic implementation of a linked list.

To demonstrate use of the data structure, the main program (lines 23–32) sets up an integer list with elements [5, 4]. It then calls the function *total_int_list* (defined over lines 10–21) which takes as its only parameter a pointer to the head of a list and sums the values in it. The returned answer (9 in our example) is printed to the screen.

The main work is accomplished by lines 12–18. First some initialisation: the local variable *total* is set to zero, and *curr* to point to the start of the list. Then a while loop adds the current integer element in the list to the running total (*total += curr->val;*) before moving on to the next element (*curr = curr->next;*). The while loop terminates when *curr* becomes

Figure 48
Programming a list of integers from scratch

```
 1 #include <iostream.h>
 2
 3 #define nil 0
 4
 5 struct intlist {
 6   int val;
 7   struct intlist* next;
 8 };
 9
10 int total_int_list(intlist* head)
11 {
12   int total = 0;
13   intlist* curr = head;
14   while (curr!=nil)
15   {
16     total += curr->val;
17     curr = curr->next;
18   }
19
20   return total;
21 }
22
23 void main()
24 {
25   intlist node1 = { 5, nil };
26   intlist node2 = { 4, nil };
27   node2.next = &node1;
28
29   int total = total_int_list(&node2);
30  cout << "List items total to: " << total << endl;
31 }
```

equal to *nil*, signifying that there are no more elements left to process.

Figure 49 shows an equivalent program using STL. It is no longer necessary to define a suitable data structure in the code; all that is necessary is the *#include <list>* directive on line 2 that includes the template version for a list defined in STL. The object is called a “container class” because when we declare a variable of this type we also specify the type we want it to store. On line 19 an integer list is realised with the statement *list<int> vals;*. Elements can be added to the object using the member function *push_back()*, as is done on lines 20–21.

The main work is done by lines 6–12. There are still two initialisations and a while loop, but other than that the new syntax has little in common with the old. Central to this new way of processing is a variable of type *iterator* (line 7). In STL many classes include *iterator* types to provide a uniform way of working through a sequence of container objects. The first element is returned with *begin()* and the element just past the last one with *end()*. Moving to the next element is accomplished by the increment operation ++, which is overloaded by the iterator class to implement this task, and the value stored there is accessed through dereferencing (**curr* on line 10), which is also overloaded.

The STL implementation of this program is slightly smaller (25 lines

Figure 49
Programming a list of integers using STL

```
#include <iostream.h>
#include <list>

int total_int_list(list<int>* head)
{
  int total = 0;
  list<int>::iterator curr = head->begin();
  while (curr!=head->end())
  {
    total += *curr;
    curr++;
  }

  return total;
}

void main()
{
  list<int> vals;
  vals.push_back(5);
  vals.push_back(4);

  int total = total_int_list(&vals);
  cout << "List items total to: " << total << endl;
}
```

verses 31) than the conventional code. The gains are more noticeable in larger projects, because the STL *list* object is more powerful than the example here illustrates. It is, for instance, a doubly linked list that supports several forms of insertion and deletion—something that would require additional programming effort to add to the basic integer list version.

Note that the parameter to *total_int_list* in Figure 49 was implemented as a pointer, to correspond with the pointer used in *total_int_list* in Figure 48. In STL it is often more natural (and more desirable) to use references rather than pointers. Then the parameter becomes *list<int>& head*, and its member functions are called with the syntax *head.begin();* and so on.

STL maps

When implementing a digital library system, it is useful to be able to store elements in an array indexed by text strings rather than by numeric indexes. In Greenstone, for example, this greatly simplifies storing the macro files once they have been read; and the various configuration files. A data type that supports such access is called an *associative array*, and is often built in to modern high-level languages. It is also known by the name *hash array* (most notably in Perl), since hashing is the normal technique used to implement the text index.

In STL, associative arrays are accomplished using the *map* object. Figure 50 shows a somewhat contrived example that stores the age of three people (Alice, Peter and Mary) in an associative array under their own

Figure 50
Using associative arrays in STL

```
 1 #include <iostream.h>
 2 #include <map>
 3
 4 int total_int_table(map<char*, int>& table)
 5 {
 6   int total = 0;
 7   map<char*, int>::iterator curr = table.begin();
 8   while (curr!=table.end())
 9   {
10 1 total += (curr->second);
11  curr++;
12   }
13
14   return total;
15 }
16
17 int main()
18 {
19   map<char*, int> table;
20   table["Alice"] = 31;
21   table["Peter"] = 24;
22   table["Mary"]  = 47;
23
24   int total = total_int_table(table);
25   cout << "Age total: " << total << endl;
26  }
```

names (lines 19–22). The problem is to write a function that calculates the total age of the people present, without knowing how many there are or what their names are. Of course, this could be solved with a standard numeric array of integers. The example is contrived to demonstrate the features of the *map* object and bring out the similarity of processing with the *list* object with an *iterator*.

Like *list*, *map* is a container class. However, when declaring a variable of this type we must specify two[3] things: the index type, and the element type. As can be seen on line 19, we obtain an associative array that stores integers using *char** (which is how a string is declared in C++) as the index type followed by *int* as the element type.

There are several ways to store elements in the associative array. In the example on lines 21–22 the overloaded array subscript [ ] is used to initialise the table with the ages of the three people. The similarity of *total_int_table*—which performs the main calculation in the program—to *total_int_list* in Figure 48 is striking. In fact, they are nearly identical, and this is no coincidence. STL makes heavy use of inheritance so that different objects still use the same fundamental operations. This is particularly true with iterators. The small differences between the two functions are that the iterator is now derived from *map<char*, int>*, and access to its elements is with *curr->second()*—because dereferencing the variable (**curr*) is defined to return an object of type *pair*. This records

[3] Technically there are four types, but the last two are optional. Since we are only giving a basic introduction to this STL class, details about these last two types are omitted.

both the index name (*first*) and the element value (*second*), but we only want the latter. Other than that, the code remains the same. The only remaining difference—changing the function's only argument from a pointer to a reference—is superficial.

Two other STL types widely used in the Greenstone code are *vector* and *set*. The former facilitates dynamic arrays, and the latter supports mathematical set operations such as union, intersection and difference.

References

Bainbridge, D., Buchanan, G., McPherson, J., Jones, S., Mahoui, A. and Witten, I.H. (submitted) "Greenstone: A platform for distributed digital library development." Submitted to *Joint Conference on Digital Libraries*, Virginia, 2001.

Deitel, H.M. and Deitel, P.J. (1994) *C++: How to Program*. Prentice Hall, Englewood Cliffs, New Jersey.

Josuttis, N.M. (1999) *The C++ standard library: a tutorial and reference.* Addison-Wesley, 1999.

Slama, D., Garbis, J. and Russell, P. (1999) *Enterprise CORBA*. Prentice Hall, Englewood Cliffs, New Jersey.

Stroustroup, B. (1997) *The C++ Programming Language*. Addison-Wesley.

Witten, I.H., Moffat, A. and Bell, T.C. (1999) *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann, San Francisco.