NetarchiveSuite Developer Manual

Printer friendly version

Contents

Introduction

edit

This manual is intended to provide a starting point for writing Java code for the NetarchiveSuite system. After a rough overview of the main packages, it

describes two of the expected primary starting points, namely localization (making NetarchiveSuite speak your language) and the JSP pages involved in the graphical user interface. After that, we provide an introduction of the coding practices we have been using and how they may apply to external developers. We then describe the present plug-in architecture and the currently available plugging points, as well as the database design used for the harvest management system. Finally, we provide a tour of what happens when a harvest is performed and when data is stored in the archive. We hope that these descriptions will allow a developer to improve or adapt some functionality of NetarchiveSuite for their own needs. This manual does not describe how to install, run, or use NetarchiveSuite, for that look to the Installation Manual and the User Manual.

The reader is expected to be familiar with Java programming and have an understanding of the core issues involved in large-scale web harvesting. Having used Heritrix before is a definite plus, and an elementary understanding of SQL databases is required for some parts.

The code is available in the downloaded package (see Release Overview) or from our subversion repository.

Modules

edit

There are six main modules in the NetarchiveSuite software, though one of them (the Deploy module) is so specific to the Netarkivet installation that it's only included for compilability and should otherwise be ignored. This section gives an overview of what's contained in each module, and points out some of the most important packages. All sources are found in the `src` directory, and all packages start with `dk.netarkivet`. Units tests are similarly arranged, but under `tests` instead of `src`. The web interface definitions are found in the `webpages` directory.

Common

The `dk.netarkivet.common` package and its subpackages provide module-neutral code partly of a generic nature, partly specific to NetarchiveSuite.

Harvester

The `dk.netarkivet.harvester` package and its subpackages handle the definition and execution of harvests. Its main parts are the database containing the harvest definitions (the `datamodel` subpackage), the webinterface that the user can access the database with, the `scheduler` subpackage which handles scheduling and splitting info jobs, and the `harvesting` subpackage which encapsulates running Heritrix and sending the results off to the archive.

Archive

The `dk.netarkivet.archive` package and its subpackages provide redundant, distributed storage primarily for ARC files as well as Lucene indexing of same. The `arcrepository` subpackage contains the logic of keeping multiple bit archives synchronized. The `bitarchive` subpackage contains the application that stores the actual files and manages access to them. The `indexserver` subpackage handles merging CDX files and crawl.log files into a Lucene index used for deduplication and for viewerproxy access.

For more detailed description, please refer to ArchiveOverview

Access

The `dk.netarkivet.viewerproxy` package implements a simple access client to the archived data, based on web-page proxying.

Monitor

The `dk.netarkivet.monitor` package provides web-access to JMX-packaged information from all NetarchiveSuite applications.

Deploy

The Deploy module should be ignored until new assignment deploy assignment has been implemented (please refer to Deploy Assignment 1), as the current implementation is fairly specific to the Netarkivet setup. It is only distributed because it would be more bother to fix the compilation problems inherent in excluding it.

Harvesting roundtrip

edit

This section describes what goes on during a harvest, from the templates are uploaded and harvests created till the ARC files are uploaded to the archive and historical information about the harvest is stored.

Initial steps

To create a harvest in the first place, we need to have a template to base it on. Additionally, we need a schedule for a selective harvest or some domains to harvest for a snapshot harvest.

Uploading templates

Templates (Heritrix order.xml files) are uploaded using the `Definitions-upload-harvest-template` page. Templates are internally represented

with the HeritrixTemplate object, which as part of its constructor verifies that certain elements - later modified programmatically - exist in the template. The template is then stored in the `templates` table in the database.

Creating domains

Domains can either be auto-created through selective harvest definitions or mass upload, or they can be created manually through the interface. Domains are represented with a Domain object, which in turn contains a list of SeedList objects and a list of DomainConfiguration objects. These are stored in the `domains`, `seedlists`, and `domainconfigurations` tables respectively. New domains are created with a single configuration, a single seedlist containing `http://www.<domain>` as its only seed, and a limit on number of bytes to download.

Creating schedules

NetarchiveSuite comes with four schedules, repeating respectively hourly, daily, weekly and monthly. More schedules can be created using the web interface. Schedules are represented with the Schedule object and stored in the `schedules` table in the database.

Creating a selective harvest

Harvests are created using the web-based user interface, as described in the User Manual.

Scheduling and splitting

The scheduler, a part of the harvest definition application, is responsible for creating new jobs for harvests, including splitting jobs into manageable chunks and sending them off to the harvesters.

Scheduling

The HarvestScheduler class runs a TimerTask that executes every minute (which is the finest interval available for defining harvests anyway). It explicitly makes sure that only one scheduling thread runs at a time, as scheduling can take a while when a snapshot harvest is started.

When the scheduler activates, it selects all harvests that are ready to harvest, i.e. harvests that are activated and where the next harvest date is in the past (or, for snapshot harvest, they haven't run yet). For each harvest, a new thread is started to perform the operations required to send off the jobs -- this keeps snapshot harvests from blocking scheduling of selective harvests. Since the threads may run for a while, we keep a set of harvests currently undergoing scheduling and uses it to avoid that the same harvest gets scheduled several times concurrently.

As a side note, backups of embedded databases are performed by HarvestScheduler, too, as part of the regular scheduling check.

Splitting

Splitting a harvest into jobs and sending those jobs off to the harvesters happens, as mentioned, in a separate thread. The first part of splitting is to reduce the harvests into chunks that can be handled by the scheduler itself -- since the data structures for domain configurations and their associated domains contain a fair amount of information, we cannot keep all of them in memory at the same time. For that reason alone, we split harvests into arbitrary chunks with no more domain configurations per chunk than the configChunkSize setting allows. We use an iterator to avoid keeping all the domains and their configurations in memory for this operation, which iterates over all configurations in sorted order. The configurations are sorted by order.xml template, then by maximum number of bytes to harvest, and finally by the expected number of objects harvested. This makes sure that configurations that should be in the same job are sorted next to each other. A FilterIterator weeds out configurations in a snapshot harvest whose domains either are marked as aliases or were completely harvested in the snapshot harvest that the current snapshot harvest is based on.

For each chunk, we iteratively create new jobs by taking one domain configuration at a time and checking if it can be added to the job we're building. If it cannot, we store the job and start making a new one. Note that the jobs created are not submitted to the harvesters yet, that happens asynchronously as part of the scheduling check.

The check for whether a configuration can be added to a job is the most complex part of the scheduling system. It is based on the need to partition the domains into chunks such that all domains in a job take approximately the same amount of time to harvest and doesn't exceed memory limits of Heritrix. The estimation of the size of a domain is complicated by the facts that previously unharvested domains have an unknown size, and that domains can easily increase in size by several orders of magnitude by adding forums, image galleries or crawler traps. Furthermore, each Heritrix instance can only use one order.xml file.

Whether a domain configuration can be added to a job is a multi-stage check with the following stages:

1. The configuration must not already have been added to the job.
2. The job must not end up with more than configChunkSize configurations.
3. The configuration must use the same crawl template as the other configurations in the job.
4. If the byte limit for this job is determined by the harvest definition, the configuration must not have a smaller byte limit than the definition specifies. If the byte limit for the job is determined by the other configurations in the job, this configuration must have the same byte limit as the other configurations.
5. The expected number of objects harvested by all configurations in the job, based on previous harvests of the configurations, must not exceed the maxTotalSize setting.
6. The *relative* difference between the largest and smallest expected numbers of objects harvested by configurations in the job must be no more than the maxRelativeSizeDifference setting. Note that the default setting for this is 100, so expectations within a job differ by a factor 100, not just 100%. This prevents jobs from finishing many small configurations quickly and take a long time to finish a few, large configurations.
    - However, if the *absolute* difference between the largest and smallest expected numbers of objects harvested by configurations in the job is less than the .minAbsoluteSizeDifference setting, the relative difference is ignored. This allows the very smallest configurations to be lumped together in fewer jobs.

Note: Check on overrides.

The expected number of objects is found based on previous harvests of a given configuration and a few assumptions about the development of web sites. If a configuration hasn't been harvested before, defaults from the settings file are used. Expectations for previously harvested domains are calculated as follows:

1. The "best" previous harvest to estimate from is found by picking the most recent complete harvest using the configuration, or the harvest that harvested the most objects if the configuration never completed.
2. The expected size per object is found based on the average size in the "best" previous harvest, if that harvest got enough objects to be considered (at least 50), but at least as many as the expectedAverageBytesPerObject setting.
3. A maximum number of objects is found based on the current limits of the configuration and the harvest and the expected size per object. If neither configuration nor harvest imposes any limits, an artificial limit for estimation purposes is taken from the maxDomainSize setting.
4. A minimum number of objects is the number of objects found in the the "best" previous harvest, or is 0 if no previous harvest was found.
5. If the configuration had previously been completed, the estimated number of objects is the difference of minimum and maximum divided by the errorFactorPrevResult setting plus the minimum.
    - Otherwise, the estimated number of objects is the difference of minimum and maximum divided by the errorFactorBestGuess setting plus the minimum.
6. The expected number of objects is capped by the maximum based on the limits.

The errorFactorBestGuess setting should generally be smaller than the the errorFactorPrevResult setting, since there is more uncertainty about the actual number of objects when the harvest has never been completed. These two settings are best understood as the largest possible factor of error between our estimate and reality. If we use an error-factor of 10, we accept that while configurations could end up growing by as much as the hard limits allow, we split as if they only grow by one-tenth that amount. In most cases, growth will be limited, but it is likely that if a new archive or forum or somesuch is added to a site, the site can grow significantly between harvests. These settings determine the trade-off between the likelihood that some sites have grown a lot and the desire to keep similar-sized configurations in the same job.

Once the job does not get any more domain configurations added to it, it is added to the database with status 'New', and cannot change further except for status updates.

When all domain configurations for a harvest have been placed in jobs, the time for the next execution of the harvest is set. Note that the execution time is updated regardless of whether the jobs are actually successfull, or even have been run. Additionally, the counter of number of runs is updated.

If there are any errors in the scheduling process, including the creation of jobs, the harvest is deactivated to prevent the system from being overloaded with broken scheduling attempts.

Talking to the harvesters

Jobs created by the scheduler are sent to the harvesters as a DoOneCrawlMessage. This message contains not only the Job object, but also some metadata entries that are associated with the job. Currently, the metadata consists of a listing of the aliases that were used in the job creation and of a listing of the job IDs that should be used to get the deduplication index.

The DoOneCrawlMessages are placed on a JMS queue, either ANY_HIGHPRIORITY_HACO for selective/event harvests or ANY_LOWPRIORITY_HACO for snapshot harvests. At the same time, the job is set to status 'Submitted', indicating that it's in queue for being picked up by a harvester. The names of these queues is a historical artifact and does not indicate that "high priority" jobs can "get ahead" of "low priority" jobs, and there could potentially be just one or more than two queues. Notice that since the JMS messages are expected to be cleaned from the queues at system restart, we assume that any messages about jobs in state "Submitted" are lost after a restart, and they are therefore automatically resubmitted at system startup.

Each HarvestControllerServer application listens to just one of the two queues (unless it is out of disk space). When it receives a message (remember that JMS guarantees exactly-once delivery for queues), it immediately sends a message back that tells the scheduler that the job has been picked up and can be put in state 'Started'. The HarvestControllerServer then spins off a new thread that as the first thing stops the HarvestControllerServer from listening for more jobs (it is done this way due to limitations on what the thread that JMS started can do to its listeners). There is also a bit of logic to ensure that no job messages are accepted between the start of the thread and the time that it stops listening.

At this point, the HarvestControllerServer has accepted that it will attempt to run the job and can start to set up the files necessary for running Heritrix.

Harvest setup

The directory used in a crawl is created by the HarvestControllerServer, using the job id and timestamp in the directory name. Details on what Heritrix reads and writes can be found in the Heritrix "outside the GUI" page.

...

Running Heritrix

The HeritrixLauncher class sets the correct file paths in the Heritrix order.xml file and keeps an eye on the progress of the harvest. If Heritrix does not download any data for a period of time defined by the `noResponseTimeout` setting, HeritrixLauncher will stop the crawl. This is to avoid a single very slow web server from extending the crawl for very little gain. Also, if no crawler threads are active in Heritrix for a period of time defined by the `inactivityTimeout` setting, HeritrixLauncher will stop the crawl. This is a workaround for a subtle bug in Heritrix.

Since version 3.3.2, Heritrix is run by the harvester system as a standalone process. This allows access to Heritrix' web interface. The interfacing to the Heritrix process is controlled by JMXHeritrixController, an implementation of the HeritrixController interface. The old method of running Heritrix is implemented with the DirectHeritrixController, which is now deprecated. General documentation on JMX can be found as part of the Java documentation, on the Sun JMX Technologies pages, in the JMX Accelerated Howto, and via the JMX Wikipedia page. Heritrix' documentation of its JMX interface is partially described in the JMX feedback page, but can also be investigated in more depth via the Heritrix JMX command-line client, and in the source files Heritrix.java and CrawlJob.java (links for Heritrix version 1.12.1).

JMXHeritrixController starts a new process as part of its constructor, putting the jar files in `lib/heritrix/lib` and the NetarchiveSuite jar files in the

classpath. The process is started in the directory created by the HarvestControllerServer, and all files created as part of the crawl are put into that directory. Stdout and stderr from Heritrix, along with a dump of the startup environment, are put in the `heritrix.out` file. The full command line used for running Heritrix is put in the log file.

Before the process is started, a shutdown-hook is added to attempt proper cleanup in case the harvest controller is shut down prematurely. Notice that this hook is removed if the process finishes normally.

After constructing the JMXHeritrixController object, HeritrixLauncher calls the initialize() method on the JMXHeritrixController, which first checks that we're talking to the correct Heritrix instance (in case one was left over from earlier), then uses the addJob JMX command to create a job for the crawl. Before returning from initialize, we call getJobName() to extract from the job a unique name we can use to locate it by later. getJobName() also has the task to wait (using exponential back-off) until the job has actually been created, since the addJob command can return before the job actually exists.

After initialize() is done, the requestCallStart() method executes the JMX command requestCrawlStart to start the job, and we then enter a loop for the duration of the crawl. Inside the loop, we check for the two timeouts as well as for orderly termination of the job and log status reports every 20 seconds. These logs can be seen by the user in the System Overview web page.

Access to the Heritrix user interface can be had by connecting to the port specified by the `heritrixAdminGui` setting, using the admin name and password specified by the `heritrixAdminName` and `heritrixAdminPassword` settings, respectively.

The cleanup of the JMXHeritrixController involves issuing the shutdown JMX command to Heritrix, then waiting for a while (duration defined by the processTimeout setting) for Heritrix to end crawls and write its reports. If Heritrix doesn't stop within the timeout period, we forcibly kill it. After that, we collect the exit code and wait for the stdout/stderr collector processes to finish.

Creating metadata

After the heritrix has finished with the harvesting, the harvest is documented, and the result of this documentation is stored in a separate arcfile prefixed with the job id, and ending with "-metadata-1.arc". This metadata file contains all heritrix logs and reports associated with this harvest(crawl.log, local-errors.log, progress-statistics.log, runtime-errors.log, uri-errors.log, heritrix.out, crawl-report.txt, frontier-report.txt, hosts-report.txt, mimetype-report.txt, processors-report.txt, responsecode-report.txt, seeds-report.txt), some metadata about the job itself, and CDX'es of the contents of the arcfiles created by Heritrix. A CDX line points to where an object is located in an arcfile, its length and mimetype. This metadata arcfile is uploaded along with the rest of the arcfiles.

Uploading

When Heritrix is finished, and the metadata arcfile created, all arcfiles are uploaded to the archive using a ArcrepositoryClient.

Storing harvest statistics

When uploading is done, a status message is sent back to the scheduler, containing error reports and harvest statistics. Errors are split into harvest errors and upload errors, since upload is attempted even if the harvest fails. For each, a short error description and a longer, detailed description are sent. The statistics sent are the following for each domain harvested:

- Number of objects harvested
- Number of bytes harvested
- Reason the harvest stopped, one of completed (no more objects found), object-limit (hit maximum allowed number of objects), size-limit (hit maximum allowed number of bytes, as specified by the harvest), config-size-limit (hi maximum allowed number of bytes, as specified by the configuration), and unfinished (the harvest was interrupted before any of the other stop reasons applied).

...need to specify what gets counted within a domain...

...need to clarify the states of a harvest...

When the status message is received, the statistics from it is stored per domain in the database, along with the job number, the harvest number, the domain name, the configuration name, and a timestamp for receipt of the information.

After the harvest statistics have been sent to the database, the HarvestController application exits. This is for historical reasons, as Heritrix used to be run internally in the application and would leak memory. To restart the application for the next harvest, the SideKick application monitors a file (defined by the `isRunningFile` setting) that the HarvestController creates at startup and removes at shutdown (using File.deleteOnExit). When the SideKick has once seen the file and then finds it gone, it runs a shell script (given on the command line) that should restart the HarvestController. You can expect this restarting system to disappear in some later version, but for now it remains. Note that it is possible, though unlikely, that the HarvestController VM could crash in such a way that it doesn't get a chance to remove the monitored file and thus isn't restarted.

Old jobs

When a harvester application starts up, it checks whether any jobs are left from previous runs, in case the harvest or the upload was aborted. If there is, the last three steps described above are taken for the old jobs before the harvest application starts listening for new jobs.

Localization

edit

The NetarchiveSuite web pages are internationalized, that is they are ready to be translated into other languages. The default distribution only contains a default (English) version and a Danish version, but adding a new language does not take any coding. All translatable strings are collected in five resource bundles, one for each of the five main modules mentioned above. The default translation files are `src/dk/netarkivet/common`

`/Translations.properties`, `src/dk/netarkivet/archive/Translations.properties`, `src/dk/netarkivet/harvester/Translations.properties`, `src/dk/netarkivet/viewerproxy/Translations.properties`, and `src/dk/netarkivet/monitor/Translations.properties`.

To translate to a new language, first copy each of these files to a file in the same directory, but with _XX after Translations, where XX is the Unicode language code for the language you're going to translate into, e.g. if you're translating into Limburgish, use `Translations_li.properties`. If you're translating into a language that has different versions for different countries, you may need to use _XX_YY, where XX is the language code and YY is the ISO country code, e.g. `Translations_fr_CA.properties` for Canadian French. Then edit each of the new files to have your translation instead of the English translation for each line. Most of the important syntax should be evident from the original, but for details consult the XXX. Note that non-ASCII characters are illegal in a translation resource bundle, but some bundle-aware editors will do the translation between UTF-8 and escaped Unicode characters.

The translation has not been done throughout the code, only in the web-related parts. Thus log messages and unexpected error messages are in English and cannot be translated through the resource bundles.

JSP

edit

The webpages in NetarchiveSuite are written using JSP (Java Server Pages) with Apache I18N Taglib for internationalization. To support a unified look across pages from different modules, we have divided the pages into SiteSections as described in the next section. Any processing of requests happens in Java code before the web page is displayed, such that update errors can be handled with a uniform error page. Internationalization is primarily done with the taglib tags <fmt:message>, <fmt:date> etc.

The main feature of JSP is that ordinary Java (not JavaScript) can be used at server-side to generate HTML. The special tags <%...%> indicate a piece of Java code to run, while the tags <%=...> indicates a Java expression to run whose value will be inserted (as is, see escape mechanisms below) in the HTML. While it is possible to output to HTML from Java code using `out.print()`, it is discouraged as it a) is confusing to read, and b) does not allow for using taglibs for internationalization.

We use a number of standard methods defined in `dk.common.webinterface.HTMLUtils`. Of particular note are the following methods:

generateHeader()
: This method takes a `PageContext` and generates the full header part of the HTML page, including the starting <body> tag. It should always be used to create the header, as it also creates the menu and language selection links. After this method has been called, redirection or forwarding is no longer possible, so any processing that can cause fatal errors must be done before calling `generateHeader()`. The title of the page is taken from the `SiteSection` information based on the URL used in the request.

generateFooter()
: This closes the HTML page and should be called as the last thing on any JSP page.

setUTF8()
: This method must be called at the very start of the JSP page to ensure that reading from the request is handled in UTF-8.

encode()
: This encodes any character that is not legal to have in a **URL**. It should be used whenever an unknown string (or a string with known illegal characters) is made part of a URL. Note that it is not idempotent, calling it twice on a string is likely to create a mess.

escapeHTML()
: This escapes any character that has special meaning in **HTML** (such as < or &). It should be used any time a unknown string (or a string with known special characters) is being put into HTML. Note that it is **not** idempotent: If you escape something twice, you get a horrible-looking mess.

encodeAndEscape()
: This method combines `encode()` and `escapeHTML()` in one, which is useful when you're putting unknown strings directly into URLs in HTML.

The SiteSection system

Each part of the web site (as identified by the top-level menu items on the left side) is defined by one subclass of the SiteSection class. These sections are loaded through the <siteSection> settings, each of which connect one SiteSection class with its WAR file and the path it will appear under in the URL.

Each SiteSection subclass defines the name used in the left-hand menu, the prefix of all its pages, the number of pages visible in the left-hand menu when within this section, a suffix and title for each page in the section (including hidden pages), the directory that the section should be deployed under, and a resource bundle name for translations. Furthermore, the SiteSections have hooks for code that needs to be run on deployment and undeployment. If you want to add a new page to the section, you will only need to add a new line to the list of pages with a unique (within the SiteSection) suffix and a key for the page title, plus a default translation in the corresponding Translation.properties file. If you want it to appear in the left-hand menu, update the number of visible pages to n+1 and put your new pages as one of the first n+1 lines.

This is an example of what a simple SiteSection can look like. Note that only the first two pages from the list have entries in the left-hand menu. This class does no special initialisation and shutdown.

```
    public HistorySiteSection() {
        super("sitesection;history", "Harveststatus", 2,
              new String[][]{
                      {"alljobs", "pagetitle;all.jobs"},
                      {"perdomain", "pagetitle;all.jobs.per.domain"},
                      {"perhd", "pagetitle;all.jobs.per.harvestdefinition"},
                      {"perharvestrun", "pagetitle;all.jobs.per.harvestrun"},
                      {"jobdetails", "pagetitle;details.for.job"}
```

```
            }, "History",
               dk.netarkivet.harvester.Constants.TRANSLATIONS_BUNDLE);
    }

    public void intitialize() {}
    public void close() {}
```

Processing updates

Some JSP sites cause updates when posted with specific parameters. Such parameters should always be specified in the beginning of the JSP file. All updates of underlying file systems, databases etc should happen before `generateHeader()` is called, so processing errors can be properly redirected. The preferred way to process updates is to create a method processRequest() in a class corresponding to the web page, but under the `webinterface` package of the corresponding module. This method should take the `pageContext` and `I18N` parameters from the JSP page, together they contain all the information needed from there.

In case of processing errors, the processing method should call `HTMLUtils.forwardToErrorPage()` and then throw a `ForwardedToErrorPage` exception. The JSP code should always enclose the `processRequest()` call in a try-catch block and return immediately if `ForwardedToErrorPage` is thrown. This mechanism should be used for "expected" errors, mainly illegal parameters. Errors of the "this can never happen" nature should just cause normal exceptions. Like in other code, the `processRequest()` method should check its parameters, but it should also check the parameters posted in the request to check that they conform to the requirements. Some methods for that purpose can be found in `HTMLUtils`.

I18n

We use the Apache I18n taglib for most internationalization on the web pages. This means that instead of writing different versions of a web page for different languages, we replace all natural language parts of the page with special formatting instructions. These are then used to look up translations to the language in effect in translation resource bundles.

Normal strings can be handled with the <fmt:message/> tag. If variable paratemers are introduced, such as object names or domain names, they can be passed as parameters using <fmt:message key="translation.key"><fmt:param value="<%myVal>"/></fmt:message>. Note that while the message retrieved for the key gets any HTML-specific characters escaped, the values do not and should be manually escaped. It is possible if necessary to pass HTML as parameters.

Dates should in general be entered using <fmt:formatDate type="both">, though a few places use a more explicit handling of formats. This lets the date be expressed in the native language's favorite style.

Note the boilerplate code at the start of every page that defines output encoding, taglib usage, translation bundle, and a general-purpose I18N object. It is important that the translation bundles from the `Constants` class for the module you're in is used, or incomprehensible errors will occur.

```
    pageEncoding="UTF-8"
%><%@taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"
%><fmt:setLocale value="<%=HTMLUtils.getLocale(request)%>" scope="page"
/><fmt:setBundle scope="page" basename="<%=dk.netarkivet.archive.Constants.TRANSLATIONS_BUNDLE%>"/><%!
    private static final I18n I18N
            = new I18n(dk.netarkivet.archive.Constants.TRANSLATIONS_BUNDLE);
%><%
```

Pluggable parts

edit

Some points in NetarchiveSuite can be swapped out for other implementations, in a way similar to what Heritrix uses.

...

How pluggability works

... factories ...

...request for suggestions on pluggability areas ...

RemoteFile

The RemoteFile interface defines how large chunks of data are transferred between machines in a NetarchiveSuite installation. This is necessary because JMS has a relatively low limit on the size of messages, well below the several hundred megabytes to over a gigabyte that is easily stored in an ARC file. There are two current implementations available in the default distribution:

- FTPRemoteFile - this implementation uses one or more FTP servers for transfer. While this requires more setup and causes extra copying of data, the method has the advantage of allowing more protective network configurations.
- HTTPRemoteFile - this implementation uses an embedded HTTP server in each application that wants to send a RemoteFile. Additionally, it will detect when a file transfer happens within the same machine and use local copying or renaming as applicable. For single-machine installations, this is the implementation to use. In a multi-machine installation, it does require that all machines that can send RemoteFile objects (including the bitarchive machines) must have a port accessible from the rest of the system, which may go against security polices.
- HTTPSRemoteFile - This is an extension of HTTPRemoteFile that ensures that the communication is secure and encrypted. It is implemented with a shared certificate scheme, and only clients with access to the certificate will be able to contact the embedded HTTP server.

All three implementations will detect when 0 bytes are to be transferred and avoid creating unnecessary file in this case.

Describe interface...

JMSConnection

The JMSConnection provides access to a specific JMS connection. The default NetarchiveSuite distribution contains only one implementation, namely JMSConnectionSunMQ which uses Sun's OpenMQ. We recommend using this implementation, as other implementations have previously been found to violate some assumptions that NetarchiveSuite depends on.

Describe interface...

ArcRepositoryClient

The ArcRepositoryClient handles access to the Archive module, both upload and low-level access. There are two implementations in the default distribution:

- JMSArcRepositoryClient - this is a full-fledged distributed implementation using JMS for communication, allowing multiple locations with multiple machines each.
- TrivialArcRepositoryClient - as the name implies, this is the simplest possible implementation that can actually work: it stores all files in a single directory. This is usable for testing and small-scale harvests, or as the basis for a more complex implementation.

Describe interface...

IndexClient

The IndexClient provides the Lucene indices that are used for deduplication and for viewerproxy access. It makes use of the ArcRepositoryClient to fetch data from the archive and implements several layers of caching of these data and of Lucene-indices created from the data. It is advisable to perform regular clean-up of the cache directories.

Describe interface...

DBSpecifics

This DBSpecifics interface allows substitution of the database used to store harvest definitions. There are three implementations, one for MySQL, one for Derby running as a separate server, and one for Derby running embeddedly. Which is these to choose is mostly a matter of individual preference. The embedded Derby implementation has been in use at the Danish web archive for over two years.

Describe interface...

Notifications

The Notifications interface lets you choose how you want important error notifications to be handled in your system. Two implementations exist, one to send emails, and one to print the messages to `System.err`. Adding more specialised plugins should be easy.

Describe interface...

Database

edit

Database overview

Below you find a database diagram of the physical tables in the NetarchiveSuite database.

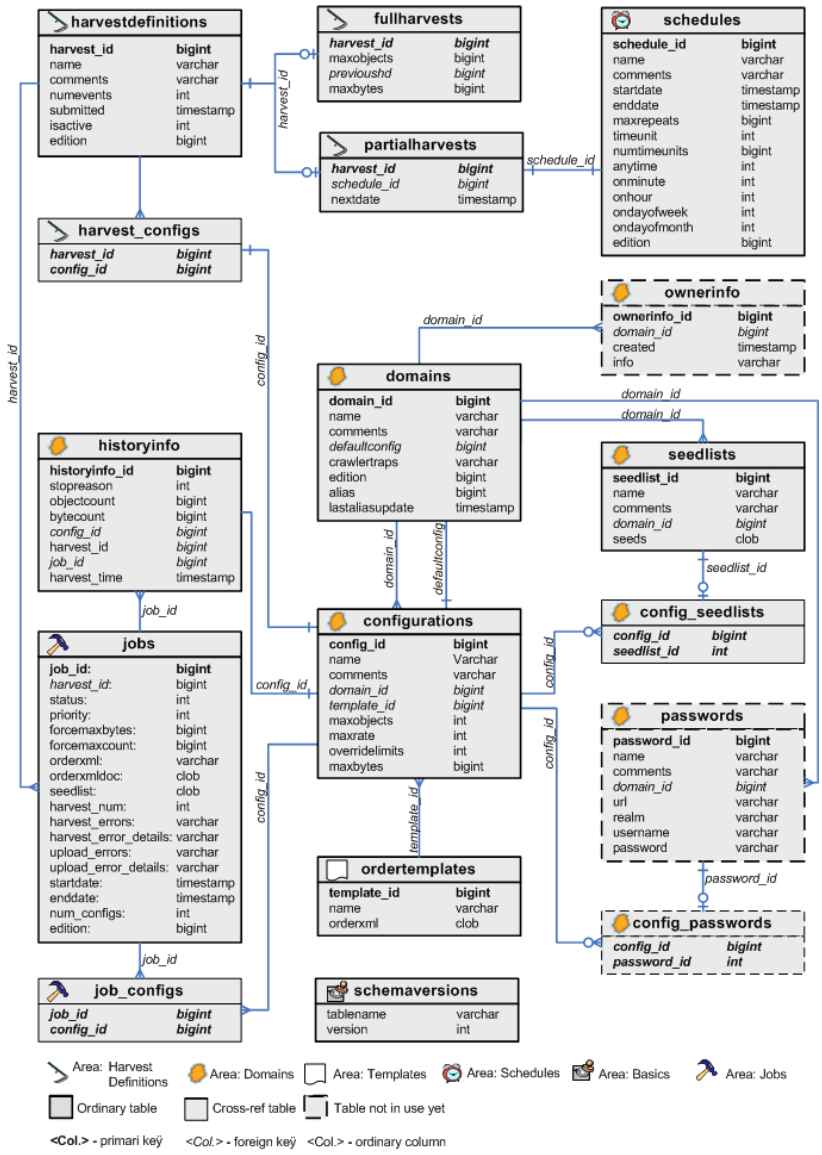

Table and Column Descriptions

Description of the individual tables and their columns can be now found in the Derby SQL create script *createfullhddb.sql*. Note that Derby automatically generates indexes for unique columns, other types of databases may need to set these indices manually.

The table and column descriptions is given on form:

```
<ALL> :== <AREA_WITH_TABLES>*

<AREA_WITH_TABLES> :==
  --************************************************--
  -- Area: <Area Name> -- <Area description>
  --************************************************--
  <TABLE_DEFINITION>*

<TABLE_DEFINITION> :==
  ------------------------------------------------------
  -- Name:    <Table name>
  -- Descr.:  <Table description>
 [-- Purpose: <Purpose description>]
 [-- Expected entry count: <Exp. count description>]
  <SQL_CREATE_TABLE_DEF>
  <Sql create index def>*
  <Sql insert data def>*

<SQL_CREATE_TABLE_DEF> :==
  create table <Table name>(
    <COLUMN_DEF>+
   [<Primary key definition over more columns>]
);

<COLUMN_DEF> :==
   <Sql column def> -- <Column description>
```

Getting data out

edit

Indexes and caching

The deduplication code and the viewer proxy both make use of an index generating system to extract Lucene indexes from the data in the archive. This system makes extensive use of caching to improve index generation performance. This section describes the default index generating system implemented as the IndexRequestClient plugin.

There are four parts involved in getting an index, each of them having their own cache. The first part resides on the client side, in the IndexRequestClient class, which caches unzipped Lucene indexes and makes them available for use. The IndexRequestClient receives its data from the CrawlLogIndexCache in the form of gzipped Lucene indexes. The CrawlLogIndexCache generates the Lucene indexes based on Heritrix crawl.log files and CDX files extracted from the ARC files, and caches the generated indexes in gzipped form. The crawl.log files and CDX files are in turn received through two more caches, both of which extract their data directly from the archive using batch jobs and store them in sorted form in their caches.

All four caches are based on the generic FileIndexCache class, which handles the necessary synchronization to ensure that not only separate threads but also separate processes can access the cache simultaneously without corrupting it. When a specific cache item is requested, the cache is first checked to see if it already exists. If it doesn't, a file indicating that work is being done is locked by the process. If this lock is acquired, the actual cache-filling operation can take place, otherwise another thread or process must be working on it already, and we can wait until it finishes and take its data.

The FileIndexCache class is generic on the type of the identifier that indicates which item to get. The higher-level caches (IndexRequestClient and CrawlLogIndexCache) use a Set<Long> type to allow indexes of multiple jobs based on their IDs. The two low-level caches just use a Long type, so they operate on just one job at a time.

The two caches that handle multiple job IDs as their cache item ID must handle a couple of special scenarios: Their cache item ID may consist of hundreds or thousands of job IDs, and part of the job data may be unavailable. To deal with the first problem, any cache item with more than four job IDs in the ID set is stored in a file whose name contains the four lowest-numbered IDs followed by an MD5 checksum of a concatenation of all the IDs in sorted order. This ensures uniqueness of the cache file without overflowing operating system limits.

CrawlLogIndexCache

The CrawlLogIndexCache guarantees that an index is always returned for a given request, regardless of whether part of the necessary data was available. This is done by performing a preparatory step where the data required to create the index is retrieved. If any of the data chunks are missing, a recursive attempt at generating an index for a reduced set is performed. Since the underlying data is always fetched from a cache, it is very likely that all the data for the reduced set is readily available, so no further recursion is typically needed. The set of job IDs that was actually found is returned from the request to cache data, while the actual data is stored in a file whose name can be requested afterwards. Note that future requests for the full set of job IDs will cause a renewed attempt at downloading the underlying data, which may take a while, especially if the lack of data is caused by a time-out.

The CrawlLogIndexCache is the most complex of the caches, but its various responsibilities are spread out over several superclasses.

- The top class is the generic FileBasedCache handles the locking necessary to have only one thread in one process at a time create the cached data. It also provides two helper methods: getIndex() is a forgiving cache lookup for complex cache items that handles the partial results described before, and the get(Set<I>) method allows for optimized caching of multiple simple cache requests.
- The MultiFileBasedCache handles the naming of files for caches that use sets as cache item identifiers.
- The CombiningMultiFileBasedCache extends the MultiFileBasedCache to have another, simpler cache as a data source, and providing an abstract method for combining the data from the underlying cache. It adds a step to the caching process of getting the underlying data, and only performs the combine action if all required data was found.
- The CrawlLogIndexCache is a CombiningMultiFileBasedCache whose underlying data is crawl.log files, but adds a simple CDX cache to provide data not found in the crawl.log. It also implements the combine method by creating a Lucene index from the crawl.log and CDX files, using code from Kristinn Sigurðsson. The other subclass of CombiningMultiFileBasedCache, which provides combined CDX indexes, is not currently used in the system, but is available at the IndexRequestClient level.
- The CrawlLogIndexCache is further subclasses into two flavors, FullCrawlLogIndexCache which is used in the viewer proxy, and DedupCrawlLogIndexCache which is used by the deduplicator in the harvester. The DedupCrawlLogIndexCache restricts the index to non-text files, while the FullCrawlLogIndexCache indexes all files.

The two caches used by CrawlLogIndexCache are !CDXDataCache and CrawlLogDataCache, both of which are simply instantiations of the RawMetadataCache. They both work by extracting records from the archived metadata files based on regular expressions, using batch jobs submitted through the ArcRepositoryClient. This is not the most efficient way of getting the data, as a batch job is submitted separately for getting the files for each job, but it is simple. It could be improved by overriding the get(Set<I>) method to collect all the data in one batch job, though some care has to be taken with synchronization and avoiding refetching unnecessary data.

Viewerproxy

The viewerproxy uses the Jetty HTTP server library to handle connections. Each incoming URL is sent through a pipeline of "resolvers", each of which can either process the URL or pass it on to the next resolver. The <tt>executeCommand</tt> method should be overridden to handle requests, and should return true if the requests was handled by this resolver. The resolver is responsible for calling <tt>response.setStatus</tt> to set the appropriate HTTP result code.

Incoming URLs are handled by the below resolvers in the order shown.

Viewerproxy control resolver

The HTTPControllerServer class manages index setup and missing URL collection for the viewerproxy. It is mainly used through the QA web interface. It

has the following commands:

Special access URLs

The GetDataResolver class provides some special URLs in the viewerproxy that can be used for more direct access to the stored data. To use them, your browser must be set up to access the viewerproxy in the same way as when browsing harvested data. The general format of the commands are <tt> http://viewerproxy.invalid/<command>?arg1=value1&arg2=value2...</tt> The commands are:

- getFile - gets a whole file from the archive
  - arcFile=<filename> - name of the file (without pathnames)
- getRecord - gets a single ARC record from the archive
  - arcFile=lt;filename> - name of the file to look up a record in (without pathnames)
  - arcOffset=<offset> - offset into the file the record starts at
- getMetadata - gets all metadata for a job from the archive
  - jobID=<id> - ID (numeric) of the job for which to fetch metadata

Observer resolver

The NotifyingURIResolver class provides means for logging what users access through the viewerproxy. It never processes any URLs itself, merely allows a URIObserver to monitor the URLs. It is currently used to record URLs that are not handled by other resolvers.

Coding guidelines

edit

This section gives some recommendations for those who want to adapt the current code and/or send in new plugins. They should be regarded as recommendations, not rules, but following them will make life easier for both parties.

Sending patches

We're always happy to receive patches, though we may choose not to apply them if the implemented features go against our purposes or the code quality is too low. Patches should be made by doing svn diff, either against a released version or against the newest svn. If sending patches by email, please send them as attachments rather than inline, as mail readers tend to mess up important formatting.

Coding style

Our overall coding style is based on Sun's guidelines with the following extensions:

Nested class definitions

Declare nested classes as static whenever possible. This avoids an unnecessary link back to the outer class, and in particular makes it possible to serialized the inner class even if the outer class is not serializable. The "invisible" link to the outer class found in a non-static inner class can also lead to unexpected memory leaks, as an inner instance may outlive its outer instance and keep it artificially alive through its implicit link. Nested class definitions appear at the beginning of the enclosing class before (static) variables.

Example:

```
class A {
    public static B {
        // B stuff
    }
    public static Integer ACONST=42;
    ...
}
```

Variable declarations

The general rule is "put declarations only at the beginning of blocks". We allow one exception from this rule, namely declarations that 1) initialize the variable and 2) depend on previous calculations are allowed be further down the block. Example:

```
void Foo() {
int i1=42;
int i2=0;
i2 = f(i1);
int i3 = g(i2);
}
```

Miscellaneous

Don't use * import statements, it clutters up the namespace and makes it hard to see what is intended. Good IDEs can do your imports automatically anyway.

Tabs should never be used in the source files. Most editors can use spaces instead.

Public methods should always check that their arguments follow the JavaDoc restriction with respect to being null, empty, non-negative etc. The

ArgumentNotValid class has a number of useful methods for this.

JavaDoc is strongly encouraged, as the code might explain what happens, but not the why; the JavaDoc must describe the **intent** of the function, including assumptions and invariants as well as expectations of the arguments.

Exceptions

At the outset of the project, we decided to use undeclared exceptions throughout our code to avoid to avoid cluttering method definitions with exceptions that are merely passing through, and to have more flexibility in what exceptions can be thrown in subclasses and interface implementations. Before you argue this decision, please read the arguments for and against. Notice that the fact that an exception is unchecked does not mean that you don't need to document its usage in JavaDoc for your methods, and you can still add it to the throws clause.

At any point where exceptions enter our code, we catch them and either handle them immediately or throw one of our exceptions instead, with the caught exception as the cause. All our exceptions inherit from `dk.netarkivet.common.exceptions.NetarkivetException`, and we try to keep the number of exceptions at a minimum. At the moment, the following exceptions exist:

- dk.netarkivet.common.exceptions.PermissionDenied - used when user rights are not sufficient to perform an operation, or authentication has failed.
- dk.netarkivet.common.exceptions.UnknownID - used when trying to look up an item that does not exist..
- dk.netarkivet.common.exceptions.IOFailure - used for a plethora of unpredictable file-system or network failures, when no better cause (like PermissionDenied) can be ascertained.
- dk.netarkivet.common.exceptions.ForwardedToErrorPage - used solely in JSP page support code to abort operations after forwarding to an error page. This should be caught in the JSP page and processing of the JSP stopped.
- dk.netarkivet.common.exceptions.ArgumentNotValid - used in all public methods for checking basic validity of arguments. Covers and provides methods for checking errors like passing null references or empty strings. Should not be used to indicate things like missing files.
- dk.netarkivet.common.exceptions.IllegalState - used when something can be in one of several states, and an operation is performed that is not appropriate for the current state.
- dk.netarkivet.common.exceptions.NotImplementedException - used as a placeholder in methods that are not implemented, or in a few system-specific places for instance trying to get the number of bytes free on a disk when running on a system that doesn't have that function implemented.

One standard example of how to catch outside exceptions and handle resource freeing is:

```
InputStream in;
try {
    try {
        in = new FileInputStream(file);
        in.readAll(...);
    } finally {
        if (in != null) {
            in.close();
        }
    }
} catch (IOException e) {
    throw new IOFailure("Failed to read file '" + file + "'", e);
}
```

Notice how the error message contains the file name in quotes (makes it easier to understand empty file error), and how the `IOFailure` gets the original exception passed in -- it is very important to never let the original exception vanish.

When it comes to handling our internal exceptions, the general rule is: Avoid catching exceptions unless you need to catch it. Also expressed as "Never catch an exception that you do not know how to handle" (with apologies to H. P. Lovecraft).

You need to catch an internal exception if:

- Resources must be released that cannot be properly releases with finally. Pay special attention to constructors.
- Your code can fix the problem and try again
- Your code must try an alternative execution strategy
- Your are implementing a toplevel method like main()

Logging

We use the apache.commons.logging framework for logging, which gives us one unified interface that can be realized with different underlying systems.

However, currently the monitoring component requires the underlying implementation to be Jdk14 logging, since it exposes log messages using a LogHandler implementation for the Jdk14 framework. To use another logging framework, this method would need to be redefined for that framework (for instance an appender for Log4J).

Unit tests

From the very start, a part of our development process has been to use unit tests to validate our coding. While we have had to learn some lessons about how to properly make unit tests (some of which lessons are not fully reflected in old tests yet), our overall experience is that unit tests have been a great boon to the stability of our code. We thus encourage others to make use of the unit test framework provided with NetarchiveSuite. See the "Practical matters" further down for instructions on how to run the unit tests that come with NetarchiveSuite.

What is a unit test?

A unit test is an automatically run test of a delimited part (unit) of the code -- a method. Unit tests should be small, run quickly and automatically, not depend on external resources, and not prevent other unit tests from running.

Each method, except for the most trivial getters and setters, should have a unit test. This test should check that the method does what it claims it does, and that it handles error situations in the way it claims it does. If the method changes an object's state, that state change should be checked. If the method temporarily changes an object's state, but claims to change it back, it should be checked that the state is changed back.

It is important that a unit test tests just one method. Firstly, it limits what goes into the unit test to a manageable size. Secondly, it provides a focus for what to test and what not to test -- other methods called from within the method need not be tested, as they have their own tests. Thirdly, it limits the amount of tests that will need changing if the methods interface changes. Lastly, it reduces the complexity of each test, making them more comprehensible and easier to maintain.

The JUnit framework helps streamlining unit tests, and is supported by a number of development environments (IDEs). With it, writing a unit test can be as easy as creating a method that compares the results of running the tested method against expected values. For instance, the below would be a reasonable test method for the `java.lang.String.substring(int, int)` method:

```
public void testSubstring() {
   String testString = "teststring";
   assertEquals("Legal substring should be allowed",
       "str", testString.substring(4, 7));
   assertEquals("Substring from start should be possible",
       "test", testString.substring(0, 4));
   assertEquals("Substring to end should be possible",
       "ring", testString.substring(6, testString.length()));
   assertEquals("Substring of the empty string should be possible",
       "", "".substring(0, 0));
   try {
      testString.substring(-1, 5);
      fail("Substring with negative start should be impossible");
   } catch (IndexOutOfBoundsException e) {
      assertTrue("Error message should contain illegal index value",
          e.getMessage().contains("-1"));
   }
   try {
      testString.substring(7, 5);
      fail("Substring with end before start should be impossible");
   } catch (IndexOutOfBoundsException e) {
      assertTrue("Error message should contain illegal index difference",
          e.getMessage().contains("-2"));
   }
   try {
      testString.substring(1, 100);
      fail("Substring with end too far out should be impossible");
   } catch (IndexOutOfBoundsException e) {
      assertTrue("Error message should contain illegal index value",
          e.getMessage().contains("100"));
   }
}
```

The standard method name testTestedMethodName is used by JUnit to find tests to run, and by IntelliJ/Eclipse to allow navigation to and direct execution of individual tests. This test first checks standard (successful) usage, on examples of increasing complexity, then goes on to check the error scenarios, making sure that the right exception with the right message is thrown. The `assertEquals`, `assertTrue` and `fail` methods are provided by the TestCase class in JUnit, and take care of formatting an error message in a readable manner. As an example, here is the (first part of the) output of running the testing with the third assertEquals only substringing out to `testString.length() - 1`:

```
junit.framework.ComparisonFailure: Substring to end should be possible
Expected:ring
Actual  :rin
   at dk.netarkivet.tools.UploadTester.testSubstring(UploadTester.java:44)
...
```

Why would you want to do unit tests?

Two words: **Saving time**. Unit tests increases your development time slightly, but decreases your debugging time significantly. Perhaps more importantly, it reduces the number of bugs that make it into the final code, decreasing customer dissatisfaction, support costs, re-release effort etc.

Unit tests provide a structured and simple way to continuously test your code. Large-scale (integration) tests of the entire system or significant subsystems are not good at pinpointing the reasons for failure, or at checking all possible modes of use of every single method. Large-scale tests typically are only possible late in the development cycle, when significant amounts of code have been written. Unit tests allow you to test much smaller parts of the code at a much earlier stage, letting you pinpoint errors with great accuracy and easing the task of testing extreme cases and error conditions.

A less obvious, but possibly more important, reason to do unit tests is that you get a clearer idea of what you code does (or should do). It's all too easy without unit tests to write "a method that extracts the domain name from a URL" in a way that seems to work, but that fails to even be clear about what a domain name is or what happens if the URL has no domain name. When writing the unit test, you have to ask yourself "how can I test what this method does?", and answering that question forces you to answer, in very exact terms, the question of "what does this method do?". Writing the unit test for the domain name extractor would raise questions of whether the domain name is the full hostname or a subset, which protocols are accepted (https? mailto? dns?), and importantly, how it handles malformed URLs or other bad input.

A third reason to create and maintain unit tests is that it provides a safety net for making changes to the code. In the Netarkivet project, we belatedly realized that XML doesn't scale to millions of files very well, and decided to move to using a proper database instead. The database involves 17 interrelated tables. The changeover was done in just a few man-weeks, partly because the data access was abstracted using DAO classes, but also significantly because the usages and assumptions were encoded in unit tests. Whenever code is changed, unit tests can catch unexpected side effects.

When do you write the unit tests?

In the Netarkivet project, we have used a code-unit-tests-first method of implementation. It may seem strange to test something that doesn't exist yet, but such code is actually the easiest to write unit tests for -- there is no implementation there to lead your thinking into specific paths and make you overlook the

special cases that cause bugs down the line. Typical method implementation has three steps:

1. Create the API as a stub method that is guaranteed not to work.
2. Write a unit test that uses that API -- this test will fail.
3. Implement the body of the API and see that the unit test passes.

Say that we want to create the method mentioned above that extracts domain names from URLs. The first step is to create the API and make sure it can compile:

```
public class DomainExtractor {
    /** This method extracts domain names from URLs.
     *
     * @param URL A string containing a URL, e.g. http://netarkivet.dk/index.html
     * @returns A string that contains the domain name found in the URL, e.g. netarkivet.dk
     */
    public String extract(String URL) {
        return null;
    }
}
```

Next, we create a test class for this method (using JUnit) and implement tests for the functionality. When implementing tests, we should be in the most evil mindtest possible, seeking any way we can think of to make the method do something other than it claims it does.

```
public class DomainExtractorTester extends TestCase {
    public void testExtract() {
        DomainNameExtractor dne = new DomainNameExtractor();
        assertEquals("Must extract simple domain", "netarkivet.dk",
            dne.extract("http://netarkivet.dk/index.html"));
        assertEquals("Must extract long domains", "news.bbc.co.uk",
            dne.extract("http://news.bbc.co.uk/frontpage"));
        assertEquals("Must not depend on trailing slash", "test.com",
            dne.extract("http://test.com"));
        assertEquals("Must keep www part", "www.test.com",
            dne.extract("http://www.test.com"));
    }
}
```

The `assertEquals` method inherited from test case takes three arguments: An explanatory message that tells us what we're testing for, the value that we expect to get from the test, and the actual value that the test gave us (in this case the return value of a method call).

At this point, we may realize that the method API does not specify what happens if we give it something that is not a URL, like "www.test.com". Does it throw an exception? Does it return null? Does it return some arbitrary part of the argument? Specifying error behaviour is as much a part of specifying the methods behaviour as saying what it does on the "good" cases. Also, what if the URL is not an HTTP URL, like "mailto:owner@test.com"? Possibly we were really just thinking of HTTP URLs, but then we need to specify that, too. These realizations should go into the javadoc at once, and the test should be expanded to check them (not shown here).

Tests should be written in such a manner that each test checks one thing (starting with the cases that would obviously work), and that no two tests check the same thing (e.g. checking both the URLs "http://test.com/foo" and "http://test.com/bar"). Knowing exactly what a "thing" is is not always trivial. To some extent, it can be derived from the API description, but it also depends on what the implementation will look like. An implementation using regular expressions would behave very differently from one splitting by characters, for example. Thus, the first tests should check the basic functionality, but then more can be added during implementation as special cases that might go wrong are noticed.

When the test is written to the point where basic functionality (and error cases) is tested, the test is run. This is merely a sanity check that the test compiles and works (for complex tests, there may be some setup prior to the first result being checked). The test, of course, will fail. This is clearly because the implementation is missing, so now we can go on to implementation.

Implementation will frequently seem very trivial once the tests are written. During the test writing, a lot of the special cases and error behaviours got defined, so writing the code that implements this is a much more straight-forward task. It can sometimes be beneficial to run the unit tests during implementation, when you think you've implemented some of the parts that are checked first. Also, even with a good unit test, you may still run into cases where redesign is needed, or where other code prevents you from doing what you thought you could (say, if a URL decoder library is used, and it doesn't provide you the functionality you were hoping for). Whenever the API is changed, the unit test should change too, reflecting this change -- otherwise it doesn't test that change.

Once the implementation is done, the unit test of course must pass.

This seems complex, why would you want to code unit-test-first?

The above example might look like there's a lot of coding to unit tests, and I cannot pretend that there isn't some coding. However, two factors ameliorate it: Firstly, a lot of the framework of the tests can be provided by a good IDE, secondly, unit test code is not production code and does not need to meet as rigorous a standard -- this can even make it quite fun to make unit tests, firing off one mean example after another.

Writing the unit tests before the implementation has the very real benefit of ensuring that the unit test gets written. All too often, once a method is implemented, adding more testing to it seems like a waste of time -- after all, you can just look at the implementation and see that it works, right? Our experience has shown that if the unit tests are left to be an afterthought, they simply do not get created.

Perhaps the greater benefit of writing the tests early is the way it forces you to think about what you're doing. Many programmers have an urge to get "down to the real stuff" and implement things as soon as possible. Starting with the unit tests allows the programmer to do some coding at once, but simultaneously forces him or her to think about the design before committing to implementation. Updating the API or extending the documentation while writing the first unit test is the rule rather than the exception. In particular, since the design choices found by making unit tests cannot be embodied in code yet, there is a greater tendency towards putting them in the Javadocs where they belong. One could say that there should be a correspondence between what the documentation states and what the unit tests test for -- if the tests test more, there is undocumented functionality, if they test for less, they are not complete. The unit tests come to do for code what double-entry bookkeeping does for accounting: Provide a way to double-check correctness.

A third advantage of doing unit tests first is that it forces the programmer to break the design down to manageable pieces. If a method is too complex to test, it is probably too complex to debug. If a method is hard to test due to complex interrelations with other methods, those same interrelations would be a source of hard-to-find bugs. On the other hand, a method that is easily tested can also more easily be reused in other contexts, as its behaviour is well known.

What are important things to keep in mind when making unit tests?

Make the test as simple as possible, but not simpler. Each test should test only one method, not the methods that the tested method calls. Look at what the method /itself/ does and test that. Also, check what the method promises in its JavaDoc and disregard that which is promised by those methods called in turn by the tested method.

Tests should take a short time to run, typically a fraction of a second. The Netarkivet system at the time of writing has 899 unit tests, and takes over three minutes to run on a fast development machine -- which is too long for frequent use. The longer the tests take to run, the less frequently they will be run. On the flip side, don't do "small-scale" optimizations that might save one or two instructions -- you can't tell what the Java run-time system optimizes anyway.

> Ideally, you **run the unit tests as a matter of course** during development, not as an afterthought, and slow tests are a hindrance for that. In many cases, especially with new code, you can run a subset of tests most of the time, but when changing old code, there could be cascading changes in other tests. These changes are important to catch, not only because failing tests would distract other coders, but because they indicate a dependency that might not be realized otherwise. Oftentime, when a test in another area of the project starts failing, the cause can be traced back to unclear design or lacking documentation.

Unit tests are **much more useful when they all pass**. If somebody has left some tests failing, it becomes difficult to see the effects of changing the code. If all tests pass when you start coding, you **know** that any tests that start failing are due to your changes.

You cannot always get your unit tests passing by the time you have to commit. **A halfway finished test should not disturb the other testers**, but should show up on reports. We have developed a system to allow developers to skip other developers' unfinished tests, but also have a list of the skipped tests which must be kept short and preferably contain a reason why the test is skipped. We do this by having a setting "dk.netarkivet.testutils.runningAs" on the JVM, which tells us who should be considered running the test. In an unfinished test, a check is added at the start, and if we're not running as either the developer mentioned in the check or "all", we skip the test.

You should **have a goal for coverage and measure against it**. Tools like Clover allow automatic calculation of which lines are reached by unit tests, summing up coverage by lines, statements and control points over classes, packages and the whole project. Measuring the coverage allows you to spot when you're slacking off on the testing, and can pinpoint critical areas that are not tested. In Netarkivet, we have a goal of 80% coverage of statements, and most of the time have been at 75% or higher. The non-covered part is typically error conditions and simple getter/setter methods. The former, while important to test, are difficult to set up correctly if you have error checking against "impossible" situations or exceptions caused by underlying libraries.

Always have a message on the `assertX()` or `fail()` call. **Without a failure message, you cannot tell what you're testing for** -- you end up testing things outside the target method or retesting the same thing in different contexts. The message should tell you what you're expecting to happen, e.g. `assertEquals("Should get imaginary number for square root of negative number", new Imaginary(0.0, 1.0), o.sqrt(` Not only does that make it easier to see what the problem is when the test fails, it also clarifies what the test actually tests, reducing the risk of redundant tests. Implicit in this should be to always test the results of an operation -- just running a method doesn't prove anything but that it doesn't throw an exception.

Some objects can take a long time to convert to a string, so including them in every message to `assertX()` can slow down the unit tests unacceptably. This can be ameliorated if you **make your own test utility classes** that define new `assertX()` methods, where you can delay the conversion until you know that the test has failed. Remember that each call to `acceptX()` is vastly more likely to pass than to fail, so reading an entire file into memory for each such call would slow things down a lot.

Many objects don't live in isolation, but depend on other objects and sometimes (unfortunately) on static state. A typical example of static state is a Singleton class. Even a test that does not make use of these other objects or state directly may cause them to be created as part of object construction. **Any such external object or state must be reverted to its original at the end of the test**. JUnit provides a guarantee that the `tearDown()` method is called at the end of a test, whether it succeeded, failed, or threw an exception (except if the AssertionFailed exception gets caught, which you should never do!). The `tearDown()` method, which in most cases will mirror the `setUp()` method, must ensure that the test has had no side-effects. Unfortunately, this is not always an easy job, as some side-effects can happen far away from the object itself and not be noticed until another test, far down the line, tries to use the same resource. Modular design and the use of mock objects can help isolate the test from side-effects, and usually makes the test easier to write, too.

Make sure to **vary the samples** that you test against, to avoid caches or cut-and-pasted code returning an old value that inadvertently passes for good. Also remember to make your test samples as evil as possible, doing the most obnoxious thing you can possibly do /within the parameters set/. This could include using empty strings, string with parts of regular expressions or other markup, integers that may overflow or underflow, etc.

While unit tests can point you in the direction of cleaner design, **avoid the temptation of making design decisions solely for the benefit of testability**. Use the unit tests as an indicator of potentially problematic design, not as the reason for the design. This includes setting access rights to what makes sense in the product code, even if it makes it harder to unit test. Java's security model is not exactly helpful here -- having a `friend` keyword would have made it much, much easier to test. This can be handled somewhat by using reflection (all Field and Method objects can be made accessible with the setAccessible method), but it is more cumbersome. It could be interesting to extend a compiler with a @Friend annotation that makes the compiler convert outside access to private members into calls to the reflection API.

Don't test your setup. If your setup statements (that is, statements required to make ready for testing the method in question) are so complex you need to assert their results, you're probably doing something wrong. Look into whether you are testing more than just the one method, or if the method itself needs to be split into several methods.

Don't try to prove a negative. It's tempting to test that a method call don't change things it's not supposed to, but you can't really do that. Any method can change all manner of things, if it really wants to, and you cannot check them all. Only if the JavaDoc or other design contract explicitly states that some parts are unchanged should that be checked.

How do you make a unit test for X?

We've run across many different kinds of code to make unit tests, and found solutions to at least some of them.

Interaction with external resources

When writing code that interfaces with external resources, the easiest way to test that *your* code works (don't attempt to unit test an Oracle database installation:) is to provide a mock object that emulates the resource. If you code is cleanly written, this is likely to be easy. A mock object is an object that can be used instead of the real thing, but has much reduced functionality. For instance, it may give pre-calculated answers to the specific calls that the unit tests make, or it may give dummy answers and count how many times it has been called. A generic system for mock objects is available at mockobjects.com.

Exceptions

Don't catch exceptions unless either the test should throw one, or catching is required for cleanup. The latter should be very rare, as cleanup properly is the province of `tearDown()`. Particularly, accidentally catching the exception thrown by `assertX()` or `fail()` will abort the tests with no explanation as to why. When a method specifies that it throws exceptions under certain circumstances, the correct way to test it is this:

```
try {
    myInstance.doWork(somethingAwful("green"));
    fail("Should have thrown GotSomethingAwfulException when given something awful to work on.");
} catch (GotSomethingAwfulException e) {
    assertEquals("Exception should remember color of awful thing", "green", e.getColor());
}
```

Trying to catch other exceptions leads to extra code with no gain, confusion about the interface, tests that fail in intractable ways, and incomprehensible tests. The above style should be used exactly for when the method *should* throw an exception according to its API.

`System.exit`

While calling `System.exit()` is frowned upon in server applications, you will also sometimes want to test command-line tools or other systems where `System.exit()` may reasonably be called. We have created a standard class that uses a SecurityManager to catch `System.exit()` calls, which would otherwise abort the entire test run. This can be extended to indicate whether a `System.exit()` call was expected or not.

What things are unit tests not good for?

There Is No Silver Bullet, of course. Unit tests can help you get better code, but it can only go so far. There are several types of problems that are difficult or even impossible to really test for in unit tests, and such untestable parts should be noted for testing in larger-scale tests.

Parallelization

Interactions of multiple threads, or worse, multiple processes, are difficult at best to test. Many of our attempts have ended with tests that pass only occasionally, or that sometimes hang the test system. We have a few ideas that work, though:

- Make sure the threads have recognizable names, and if the threads are expected to terminate, wait in a loop till they have. Make sure to have a timeout on it, though.

Complex interactions

Despite the best design efforts, some errors only occur when multiple components are put together. Even if each component does its part perfectly well, misunderstandings of designs and assumptions can cause unexpected behaviour. This is properly the field of integration tests. Some errors also come up because the unit test writer didn't think of every possible case, but in that case the unit test can later be extended to cover other cases.

External resources

Interactions with name servers, databases, web services or other resources that either are slow or unpredictable should be avoided, as it complicates the setup and makes spurious errors more likely. Such resources can sometimes be replaced with mock-ups that give the answers that the tested methods expect.

Hardware-dependent problems

Some bugs only occur on some platforms or when specific hardware is in use. For instance, Windows has mandatory locking that can make cause the `File.delete()` method to fail until the lock is released. This is not a problem under Unix, so our unit tests never attempted to test that problem. Much as Java would like to be truly platform-independent, there are always some differences.

Scaling

Scalability issues are typically hard to test for within the time constraints of unit tests.

Practical matters

All our unit tests are placed under the `tests` directory (along with some integrity tests), using a directory structure that mimics the classes they test (such that they can access package-private members). Each package contains an `XTesterSuite` class, where X is the last part of the package name. This class assembles the tests in that package as one bundle of tests, but also allows the tests to be run as a separate suite. Typically, each package also has a `TestInfo` class that contains various useful constants (names of test data files, for instance), and a `data` directory containing all test data for that package (but not its subpackages). The tests for a class `X` are placed in a class `XTester`, with each method `fooBar()` being tested by one or more methods whose name begins with `testFooBar` (incidentally, this format is understood by the UnitTest IntelliJ plug-in).

Running unit-tests

The unit tests can be run using the Ant target "unittest". If you want to run the unit tests in another manner, e.g. from within your development environment, run the class `dk.netarkivet.tests.UnitTesterSuite` with the parameters `-Xmx152m` and `-Djava.util.logging.config.file=tests/dk/netarkivet/testlog.prop`. It is recommended to also use the option `-Ddk.netarkivet.testutils.runningAs=NONE` to avoid running unit tests that are not expected to pass.

Excluded tests

We have a system for excluding unit tests from execution when they either depend on external issues or belong to code that is still under development. This prevents the unit test suite from being 'polluted' by tests that are not yet expected to work. A test can be excluded by using the `dk.netarkivet.testutils.runningAs` method:

```
if (!TestUtils.runningAs("LC")) {
    return;
}
```

This causes the test to end successfully unless the tests are run as the user `LC`. The tests can be run either as a specific user, as NONE or as ALL, by setting the property `dk.netarkivet.testutils.runningAs`, e.g. `-Ddk.netarkivet.testutils.runningAs=LC`. When run as ALL, every unit test regardless of exclusion is run -- this is used for the daily regression tests. When run as NONE, no excluded tests are run. Typically, a developer would run the unit test suite as him- or herself to see ones own failing tests but not be distracted by other developers failing tests.

In order to avoid excessive exclusion, it is a good idea to generate a list of which exclusions are in place by grepping for 'testutils.runningAs' in the source. At release time, only exclusions stemming from problems that cannot be solved yet and that are not blocking the release should be in place.

Private methods

Private methods are just as deserving as public methods of being tested, but due to Java's lack of a "friend" concept, they cannot be directly accessed from other classes. Instead, we have a utility class `ReflectUtils` that provides methods for accessing private methods as well as private fields (for easier setup). An example of using reflection for tests could be:

```
    hc = HarvestController.getInstance();
    Field arcRepController = ReflectUtils.getPrivateField(hc.getClass(),
         "arcRepController");
    final List<File> stored = new ArrayList<File>();
    arcRepController.set(hc, new MockupJMSArcRepositoryClient(stored));
    Method uploadFiles = ReflectUtils.getPrivateMethod(hc.getClass(),
          "uploadFiles", List.class);
    uploadFiles.invoke(hc, list(TestInfo.CDX_FILE, TestInfo.ARC_FILE2);
    assertEquals("Should have exactly two files uploaded",
                 2, stored.size()); // Set as sideeffect by mockup
    ...
```

JUnit assertions

JUnit comes with a base package of useful assertions, but we have over time crystallized out more assertions. These all live in the `dk.netarkivet.testutils` package, which is placed together with the tests. Along with a number of miscellaneous support utilities and mock-ups (described below), there are the following new asserts in the testutils package:

ClassAsserts

The assertions in here (`assertHasFactoryMethod`, `assertSingleton`, and `assertPrivateConstructor`) pertains mainly to singleton objects, of which there is a small handful in the system. The `assertEquals` method tests via reflection that the `equals` method obeys the requirements from `Object.equals`.

CollectionAsserts

The `assertIteratorEquals` and `assertListEquals` methods provide more detailed messages about differences in iterators and lists than just doing `equals`. The `assertIteratorNamedEquals` is for specific use by `HarvestDefinitionDAOTester`.

FileAsserts

These methods help inspecting test results that reside in files. The `assertFileNumberOfLines` method checks the number of lines in a file without holding the whole file in memory. The other methods are utility methods that provide more informative error messages for tests of whether files contain strings or match regular expressions.

MessageAsserts

The one assert method here checks the generic JMS message class NetarkivetMessage for whether a reply was successful and outputs the error message from it if not.

StringAsserts

The three utility methods here are similar to those of `FileAsserts` in that the provide better error messages for string/regexp matching.

XmlAsserts

These assertions help test XML structures. The `assertElementHasAttribute` and `assertElementHasNotAttribute` check for the presence of a given attribute and whether it does or does not have a given text. Similarly, the `assertNoNodeWithXPath` and `assertNodeWithXPath` methods test whether or not a node exists in a document corresponding to a particular XPath string, and the `assertNodeTextInXPath` checks if an existing node contains a specific test.

Mock-ups

As using objects in their normal contexts became more and more difficult in an increasingly complex system, we turned to mock objects to simplify unit tests. Additionally, we have standardized some of our most common set-up/tear-down procedures into objects of their own.

...

Settings

Almost all configuration of NetarchiveSuite is done through the `dk.netarkivet.common.Settings` class. It provides a simple interface to the `settings.xml` file as well as definitions of all current configuration settings. The `settings.xml` file itself is an XML file with a structure reminiscent of the package structure. All settings can also be set on the command line by using the `-D` option, this will override anything listed in the `settings.xml`

file.

Settings are referred to inside NetarchiveSuite by their path in the XML structure. For instance, the `storeRetries` setting in `arcrepositoryClient` under `common` is referred to with the string `"settings.common.arcrepositoryClient.storeRetries"`. However, to avoid typos, each known setting has its path defined as a String constant in the `Settings` class, which is used throughout the code.

To add a new general setting, the following steps need to be taken:

1. The `Settings` class should get a definition for the path of the setting.
2. The XML Schema for the `settings.xml` should be updated to allow the new setting.
3. An XSLT script should be made to add the setting to current `settings.xml` files.
4. `settings.xml` files should be updated (including those in the unit tests).

... add description of XML Schema options for pluggable parts ...