VISTA DATA FLOW SYSTEM (VDFS)

----

for VISTA & WFCAM data

Science Archive Software Architecture Design

author
R.S. Collins (WFAU Edinburgh)
Software Developer

number
VDF-WFA-VSA-009
issue
1.0
date
September 2006

co-authors
I. Bond, N.J. Cross, N.C. Hambly, E.T.W. Sutorius

Contents

SCOPE

This document presents the design of the curation software architecture for the WFCAM Science Archive (WSA), which will also be used for the VISTA Science Archive (VSA). The intended audience are those members of the VSA team directly involved in the construction of the archive. This document is also expected to be informative to interested external parties.

The exact implementation of the software will inevitable change with time, so rather than providing a detailed documentation of each individual code module here, we instead refer the reader to the on-line code documentation [2], which is automatically generated from the latest source code. Only curation software design is detailed in this document. For web access software design please refer to the User Interface Document (VDF-WFA-VSA-008).

Also included in this document is a description of the data flow in the science archive, from the initial input of individual FITS catalogue and multiframe metadata products provided by CASU to the final prepared database product that is presented to the users.

The requirement of this document is a consequence of one of the findings of the Critical

Design Review that expressed a need for a design of the software architecture. The external requirements on the design follow on from the content of the Hardware/OS/DBMS Design Document (VDF-WFA-VSA-006), the Database Design Document (VDF-WFA-VSA-007), and the User Interface Document (VDF-WFA-VSA-008). Also relevant is the Interface Control Document (VDF-WFA-VSA-004) which determines the requirements on the format of data to be transfered from CASU to WFAU and into the archive.

OVERVIEW

The software architecture is constrained to work within the confines of the entire archive design from hardware to database to end-user interface design. Therefore, this document begins with a background to the software architecture design in section 3 reminding the reader of how the software architecture fits in with the rest of the archive design. This includes a recap of the revelant hardware design and description of the architecture from an operating system standpoint.

In section 4 we present the overall design of the software architecture, which is an overview of the code designed for developers, but also provides the context for section 5. This section provides descriptions of the procedures performed by each science archive curation use case, detailing how these fit in with the overall software architecture.

Finally, section 6 describes the data flow in the science archive, from the initial individual FITS catalogue and multiframe metadata products provided by CASU to the final prepared database product that is presented to the users.

FUNDAMENTALS

Rationale

The software architecture design for the VSA follows on from the work documented previously. Here we briefly recap what has been done so far:

- The Hardware/OS/DBMS Design Document (VDF-WFA-VSA-006) described the hardware on which all of the above will be deployed. The various computer systems and their operating systems were described along details on how they will be networked.
- In the Database Design Document (VDF-WFA-VSA-007), the scenarios performed by the archive scientist were expanded upon and developed into the identification of 20 curation use cases. Details on how pixel, catalogue, and other data are stored in the VSA were given with particular reference to design of database schemas and

tables that will be managed by the DBMS.

- The User Interface Document (VDF-WFA-VSA-008) described how data will be delivered to the end user and described the underlying software that will be involved.

In this document, all this is taken to the next level by presenting detailed designs of the software architecture. Here, it is necessary to identify real software entities (as opposed to abstract modelling concepts), show their inter-dependencies and inter-relationships, and show where they will be deployed within the VSA. In designing the software architecture, we are taking a component oriented view. The goals of this is to enable re-use of individual software components and also to be able to replace existing components with newer, perhaps upgraded, ones with a minimum of disruption to the system as a whole.

OS Level Components and Deployment

In this section a top level view of the system architecture is given. Here we consider operating systems to be used along with high level enabling systems such as web servers, database management systems, and Java containers. These are listed in table 1. The overall role of these components is to run the computer systems involved and to provide a framework which enables development of components that required functionality of the VSA. Figure 1 shows the deployment of the top level components on the hardware described in the Hardware Design Document (VDF-WFA-VSA-006).

As can be seen in figure 1, and as described in the Hardware Design Document, the software architecture of the VSA will be deployed on a number of servers.

- The Web Server is a PC running Linux which provides the entry point for users to access data products from the VSA. All user interface components will be deployed here.

- The Catalogue Server is a PC running under Windows which contains a copy of the latest release of databased products. It is from here that catalogue data products are provided to the user, via the web server.

- The Load Server is a PC running under Windows which is used by the archive scientists for managing the VSA.

- The Curation Server is a PC running Linux onto which the VSA curation use cases are deployed. This server will run all the software requiring pixel and catalogue data processing involved in generating archived data products as well as handling those tasks requiring connections to the database management system deployed on the load and catalogue servers. This server will be one of the servers that form one node of the mass storage system.

Table 1: Operating system level components deployed on the VSA.

Component Name	Stereotype	Deployment	Function
Linux	OS	Curation Server, Web Server	operating system
Microsoft Windows	OS	Catalogue Server, Load Server	operating system
MS SQL Server	DBMS	Catalogue Server, Load Server	DB management system
Apache	HTTP server	Web Server	HTTP server
Tomcat	Java code	Web Server	Servelet container classes

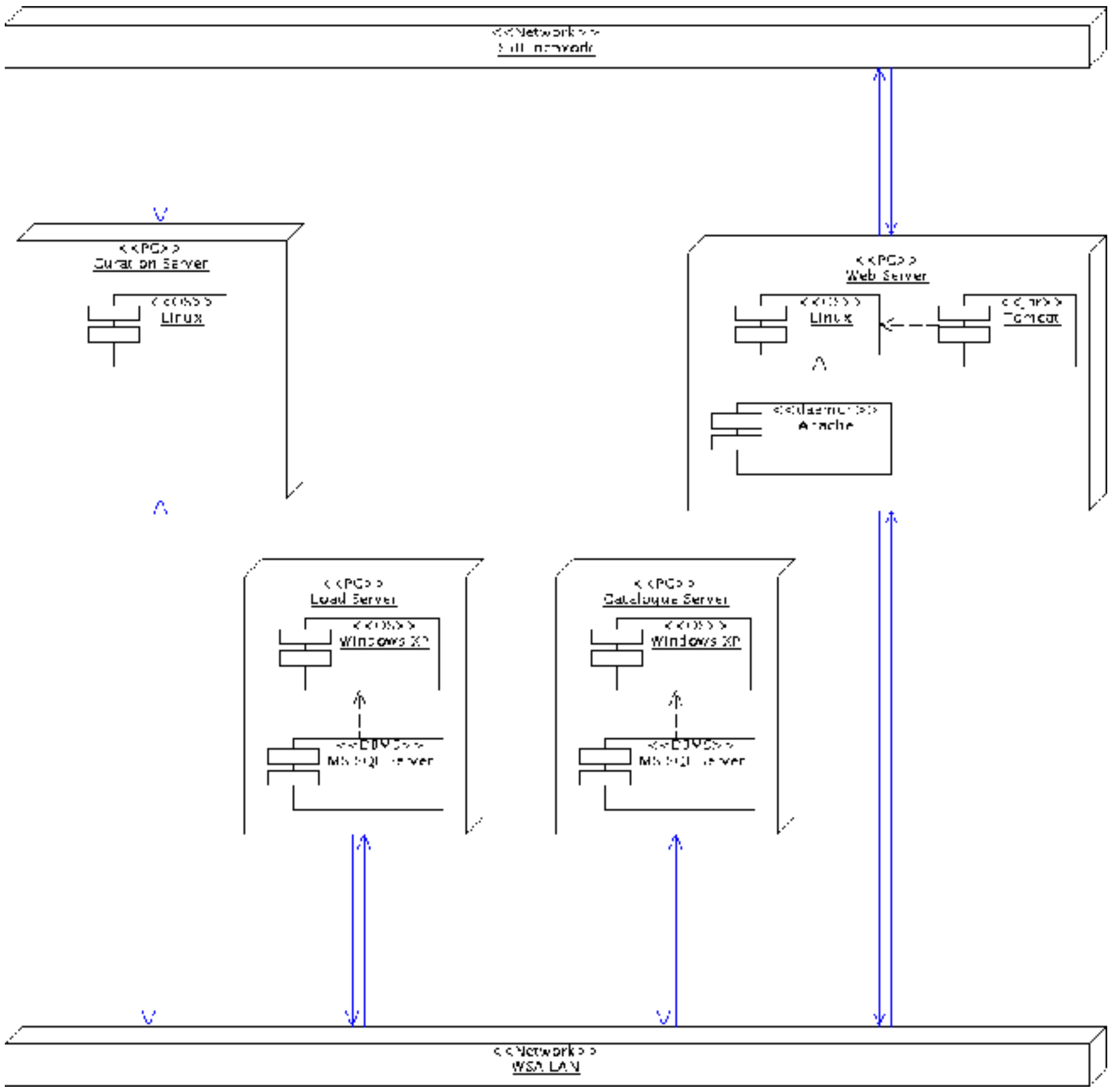

Figure 1: UML deployment diagram showing the main hardware and operating system components of the VISTA Science Archive.

SOFTWARE DESIGN

Introduction

Several developers contribute to the VSA software source code and so a central CVS repository [3] is utilised to managed code changes. The formal directory structure for the software is presented in figure 2. Wherever possible the archive software is written in the Python language, with all curation tasks scripted in Python. For the more intensive data processing tasks the software is written in C++. Either a Python/C++ interface wrapper is used to directly call the C++ functions or the C++ code is executed as an external process.

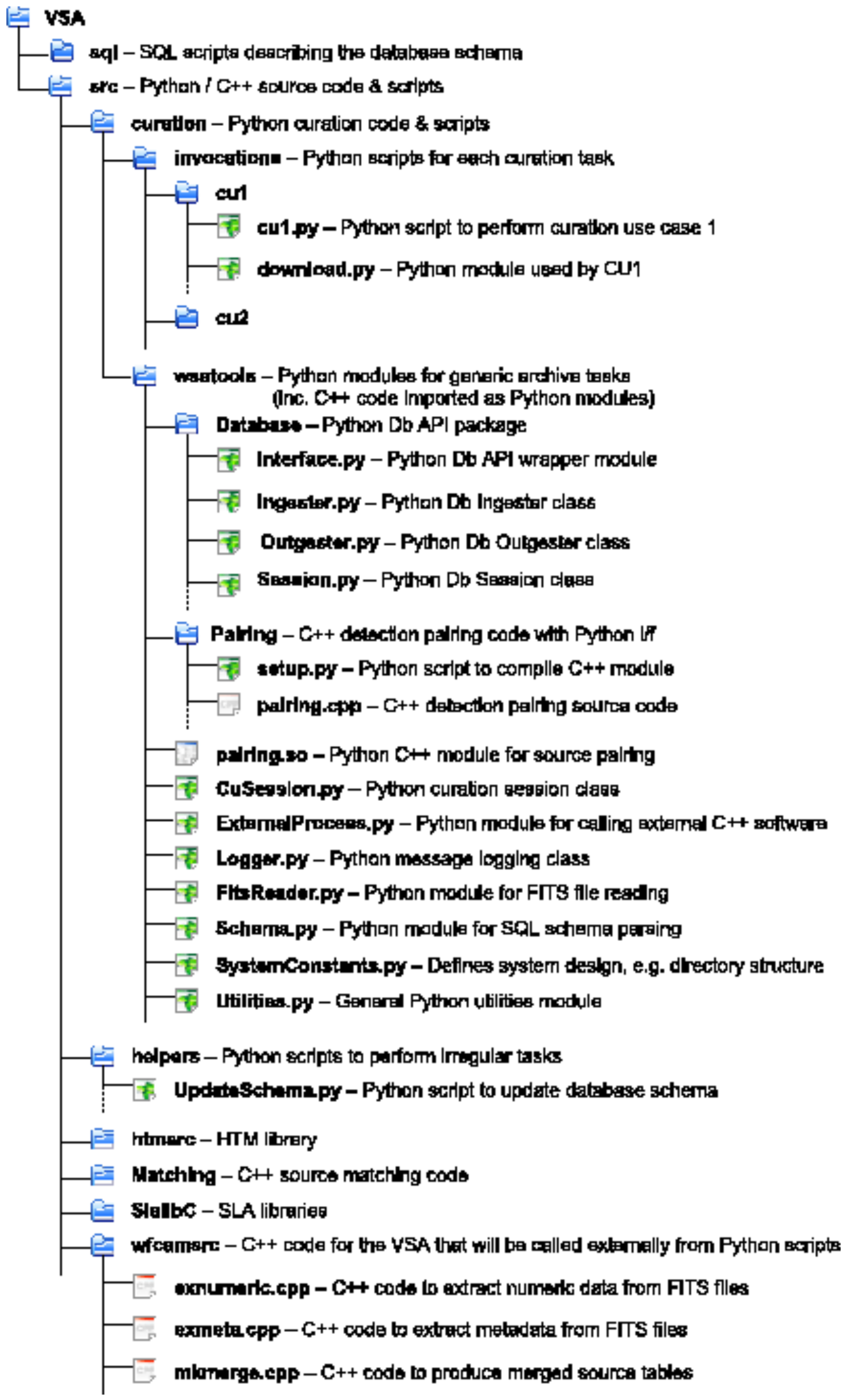
VSA
sql – SQL scripts describing the database schema
src – Python / C++ source code & scripts
curation – Python curation code & scripts
invocations – Python scripts for each curation task
cu1
cu1.py – Python script to perform curation use case 1
download.py – Python module used by CU1
cu2
wsatools – Python modules for generic archive tasks (inc. C++ code imported as Python modules)
Database – Python Db API package
Interface.py – Python Db API wrapper module
Ingester.py – Python Db Ingester class
Outgester.py – Python Db Outgester class
Session.py – Python Db Session class
Pairing – C++ detection pairing code with Python I/f
setup.py – Python script to compile C++ module
pairing.cpp – C++ detection pairing source code
pairing.so – Python C++ module for source pairing
CuSession.py – Python curation session class
ExternalProcess.py – Python module for calling external C++ software
Logger.py – Python message logging class
FitsReader.py – Python module for FITS file reading
Schema.py – Python module for SQL schema parsing
SystemConstants.py – Defines system design, e.g. directory structure
Utilities.py – General Python utilities module
helpers – Python scripts to perform irregular tasks
UpdateSchema.py – Python script to update database schema
htmsrc – HTM library
Matching – C++ source matching code
SlalibC – SLA libraries
wfcamsrc – C++ code for the VSA that will be called externally from Python scripts
exnumeric.cpp – C++ code to extract numeric data from FITS files
exmeta.cpp – C++ code to extract metadata from FITS files
mkmerge.cpp – C++ code to produce merged source tables

Figure 2: The directory structure of the software CVS repository highlighting a selection of important modules that are discussed in this document.

The primary task of the curation software is to prepare a database with data sourced from FITS files. To access the FITS files from the Python software we use the `PyFits` library, and from the C++ software we use `CFITSIO`.

Interaction with the DBMS by our software is made possible by ODBC, the industry standard API for DBMS connections. The obvious choice for an ODBC implementation to be used by our Python software was `mxODBC`, for which we were provided with a free `non-commercial use' license. `mxODBC` is the most reliable and highest quality ODBC implementation, having been designed in collaboration with the developers of the Python language. As the DBMS is hosted on a Windows PC, and the Python software is served by a Linux PC, the ODBC connection consists of many layers. On the Linux PC the Python code calls `mxODBC`, which makes use of `unixODBC`, which in turn uses `FreeTDS` to connect to the ODBC API on the Microsoft SQL Server PC.

Alternatively flat-file table data may be transfered into and out of the database through files located in the load server's file share directory. These files may either be native binary files or ASCII text files containing comma-separated values. This method is used for transferring to and from the database large quantities of data, typically produced by one of the C++ software packages that perform more intensive data processing tasks. The C++ software does not have direct access to the database through an API, so relies upon the information stored within the SQL script files to determine the schema format for both creating and reading these flat-file tables. Irrespective of the file format, for the flat-file table data to be inserted into a database table (an `ingest') it must match the schema of that table. For the outgest of table data into binary files it is also vital that the C++ code expects the correct format to read the data. Therefore, it is imperative that the schema described by the SQL script files always match the database schema. This aspect is also vital in many other aspects of the software design where the code is `schema' driven (section 6 provides further detail to the `schema' driven design of the code).

All specific aspects of the system design described in section 3.2 are encapsulated in the Python module, `wsatools.SystemConstants'. Herein lies the translation of the conceptual entities such as the load server, its share directory, and the mass-storage disk array to real device names and paths. By referencing these constants rather than, say, literal path names, the conceptual purpose of the code becomes more clear, and the task of porting the software to an alternative system will be more simple. Likewise a `wsatools.DbConstants' Python module maintains the translations of conceptual database entities such as the load database to their actual names. This module is automatically generated from the database schema script, (`VSA_InitiateArchive.sql`).

Database Package

The database software package is a set of Python modules and classes developed for the VSA that provide structured access to database data. The most significant components of the database package are the interface module, the Session class, the Table class and the Ingester and Outgester classes.

The interface module maintains a clean interface between the curation code and the database API. In principle modifications to the interface module only are required to allow the code to use an alternate database API.

The Session class maintains the connection to a specified database, and provides a set of useful access methods to the database that are of a higher level than simply passing raw SQL statement scripts to the database API. This class provides the public interface between the curation software and the database. Upon instantiation of a Session class object a connection is made to the requested database if such a connection does not already exist (the class keeps track of all open connections, and reuses an existing connection if an identical connection request is made). Upon destruction of the Session class object the connection will be closed if it is the latest such connection to the database.

Providing an additional database access wrapper interface, in the form of the Session class, not only ensures correct use of the database connection and API, but also presents the software developer with a convenient set of methods to interact with the database by supplying data in Python objects rather than in long, obscure, SQL statements. Also, should the SQL standard or our DBMS change, then only this API would need to be altered. Additionally, along this line of design, a Table class has been designed to store the complete contents of a small curation database table, such as the survey programme definition table, in a Python object for efficient access.

This package also provides Ingester and Outgester classes that handle the transfer of flat-file table data into and out of the database via files located in the load server's file share directory. In addition to performing the file transfer and ingest, the Ingester class upon initialisation checks that the database schema and that described by the SQL files are consistent. The Outgester class allows the software to outgest the results of any database query to a binary or ASCII file, and contains a work around for delays in the mounted NFS file share that can cause only partial data transfer to occur.

Curation Tasks

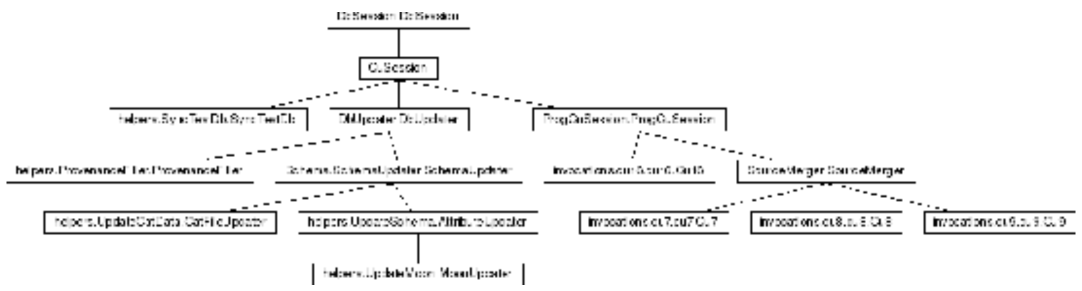
DbSession.DbSession
CuSession
helpers.SyncTestDb.SyncTestDb
DbUpdater.DbUpdater
ProgCuSession.ProgCuSession
helpers.ProvenanceFiller.ProvenanceFiller
Schema.SchemaUpdater.SchemaUpdater
invocations.cu6.cu6.Cu6
SourceMerger.SourceMerger
helpers.UpdateCatData.CatFileUpdater
helpers.UpdateSchema.AttributeUpdater
invocations.cu7.cu7.Cu7
invocations.cu8.cu8.Cu8
invocations.cu9.cu9.Cu9
helpers.UpdateMoon.MoonUpdater

Figure 3: Inheritance diagram for the CuSession class.

The generic software design of all curation tasks is encapsulated with the CuSession class (see figure 3). This class inherits the functionality of the Database package's Session class, and extends it, providing message logging and exception handling. All curation scripts inherit the parent class CuSession, which automatically ensures all essential pre-/post-curation tasks are performed. For example, upon initialisation of the curation session object:

- A database connection is opened (*inherited*).
- The database is locked.
- The Curation Use Case (CU) description is logged.
- A unique CU Event ID is created.
- A temporary working directory is created.

Upon completion of a curation session:

- A row describing this session and its outcome is written to the

ArchiveCurationHistory table.
- The message log is written to a file in the curation log directory.
- The database connection is closed. (*inherited*)
- The database lock is removed.
- Temporary files are removed from the work space directory and the load server share directory.

Furthermore, curation tasks that perform programme specific curation derive from the ProgCuSession class that inherits the CuSession class and extends it to also updates the ProgrammeCurationHistory table. If the curation session failed for some reason a clear error message is presented to the operator.

Each curation use case script implements a class derived from a CuSession object, overriding the `_onRun()` method to define the implementation of that use case. The procedural flow of the CuSession class design is presented in figure 4.

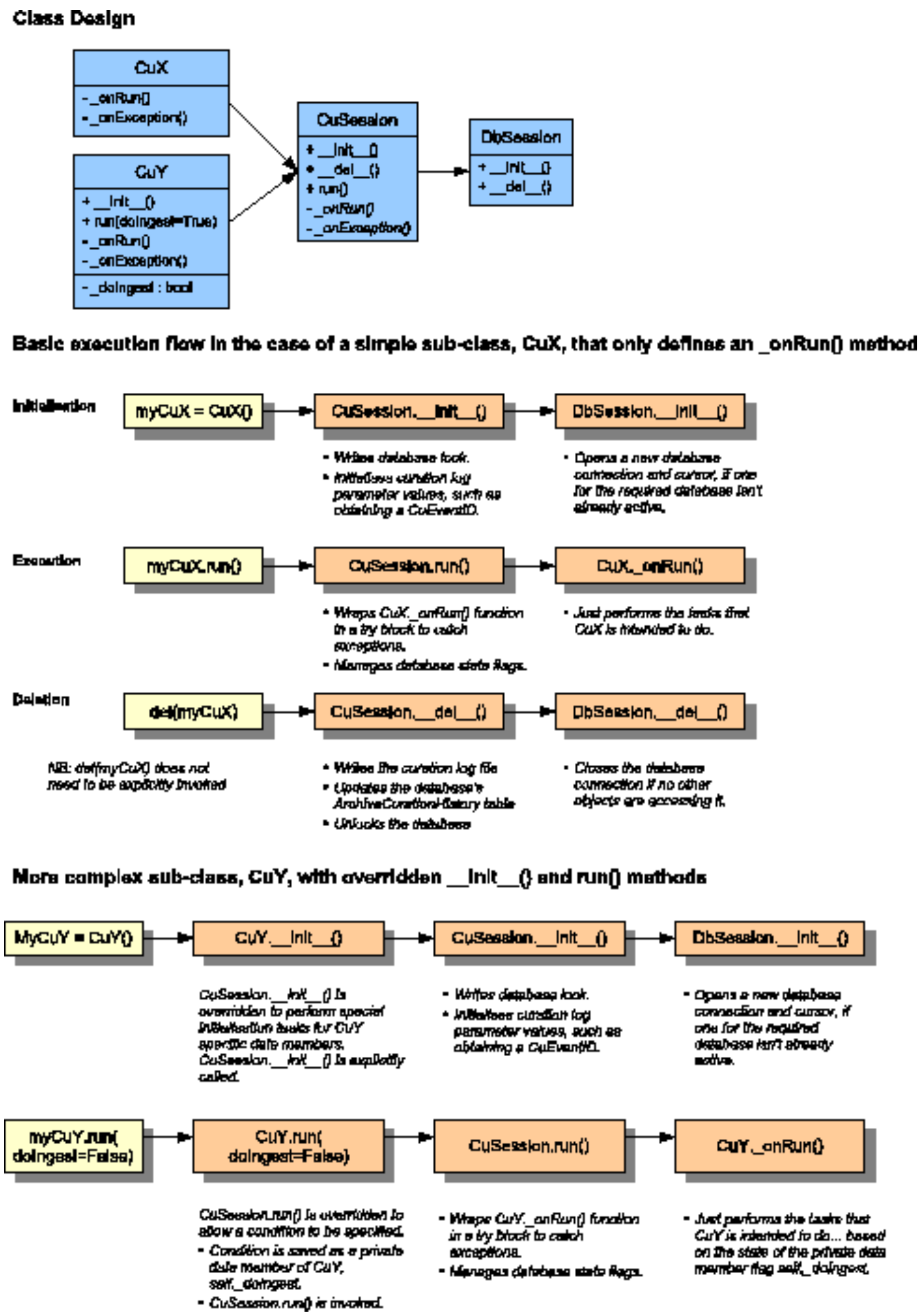
Class Design
CuX
- _onRun()
- _onException()
CuY
+ __init__()
+ run(doIngest=True)
- _onRun()
- _onException()
- _doIngest : bool
CuSession
+ __init__()
+ __del__()
+ run()
- _onRun()
- _onException()
DbSession
+ __init__()
+ __del__()
Basic execution flow in the case of a simple sub-class, CuX, that only defines an _onRun() method
Initialisation
myCuX = CuX()
CuSession.__init__()
DbSession.__init__()
• Writes database lock.
• Initialises curation log parameter values, such as obtaining a CuEventID.
• Opens a new database connection and cursor, if one for the required database isn't already active.
Execution
myCuX.run()
CuSession.run()
CuX._onRun()
• Wraps CuX._onRun() function in a try block to catch exceptions.
• Manages database state flags.
• Just performs the tasks that CuX is intended to do.
Deletion
del(myCuX)
CuSession.__del__()
DbSession.__del__()
NB: del(myCuX) does not need to be explicitly invoked
• Writes the curation log file
• Updates the database's ArchiveCurationHistory table
• Unlocks the database
• Closes the database connection if no other objects are accessing it.
More complex sub-class, CuY, with overridden __init__() and run() methods
MyCuY = CuY()
CuY.__init__()
CuSession.__init__()
DbSession.__init__()
CuSession.__init__() is overridden to perform special initialisation tasks for CuY specific data members. CuSession.__init__() is explicitly called.
• Writes database lock.
• Initialises curation log parameter values, such as obtaining a CuEventID.
• Opens a new database connection and cursor, if one for the required database isn't already active.
myCuY.run(doIngest=False)
CuY.run(doIngest=False)
CuSession.run()
CuY._onRun()
CuSession.run() is overridden to allow a condition to be specified.
• Condition is saved as a private data member of CuY, self._doIngest.
• CuSession.run() is invoked.
• Wraps CuY._onRun() function in a try block to catch exceptions.
• Manages database state flags.
• Just performs the tasks that CuY is intended to do... based on the state of the private data member flag self._doIngest.

Figure 4: A diagram detailing the run-time procedural flow of a curation session, from actual use case script implementation, through to the CuSession class, and the Database.Session class.

All scripts at present are invoked from the command-line, and utilise the `wsatools.CLI' module, which provides a `--help`/`-h` option documenting the complete command-line interface for the script. Using the object-oriented model given above, all scripts automatically inherit command-line arguments and options from their parent classes. For example, all programme curation sessions inherit from the ProgCuSession class, which provides a command-line argument to specify the unique ID of the programme to be curated for this session.

Database Table Preparation

The C++ module `wfcamsrc' has been developed as an object-oriented solution to preparing large data tables for ingest into the database. It consists of three parts at present: `exmeta' for preparing tables of metadata, primarily from FITS header keyword values. `exnumeric' for preparing detection tables from FITS binary table data, and `mkmerge' for preparing source tables from detection table data. It is designed to work for data intensive tasks where very large flat-file tables are constructed as binary or ASCII files for subsequent ingest into our relational database. A polymorphic design to the code has been adopted through the creation of a series of different classes each designed to handle the data in different, standard ways. The software is completely driven by the schema script files that are discussed in section 6, which describe the data source and how the data should be manipulated to be correctly formatted in the table for ingest. Hence, the schema describes which class to select to format the data for a given table attribute.

Code Reference Documentation

A complete, interactive, API documentation for our software is provided on-line [2]. Please refer to this documentation for a more detailed description of the software design. It is generated from the latest source code, using `epydoc` for the Python software, and `Doxygen` for the C++ software. The Python code is divide into two main packages: `wsatools', the set of generic tools for archive curation, and `invocations' (see figure 5), which contains the scripts to perform each specific curation use case (see figure 6).

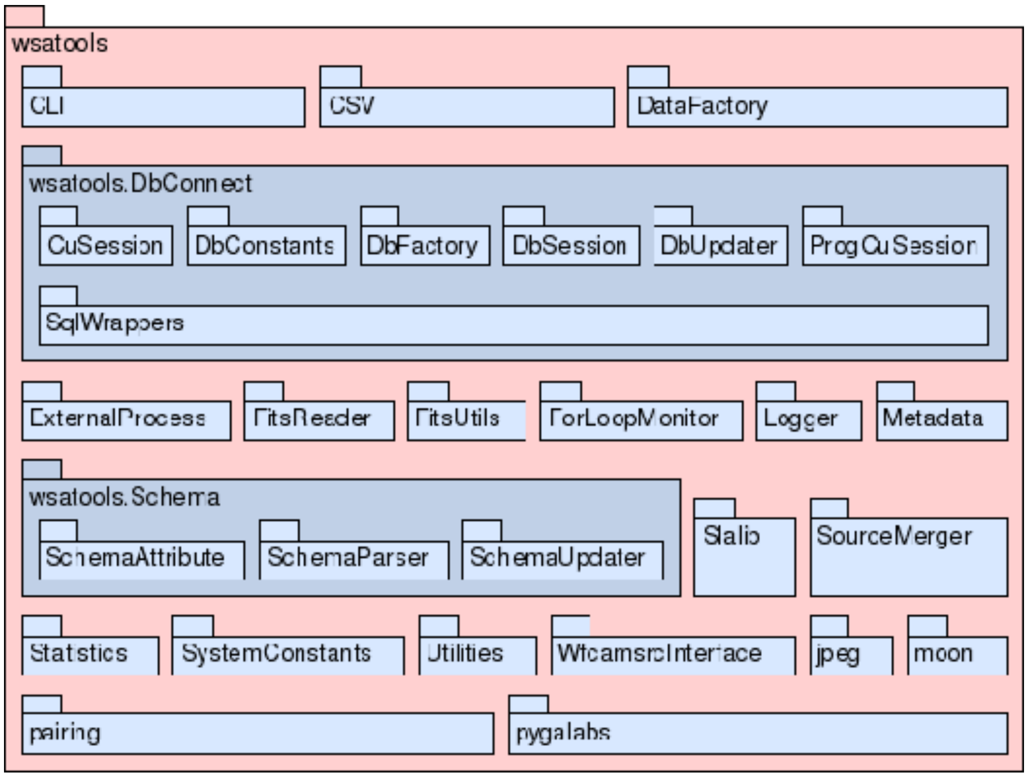

Figure 5: Package structure UML diagram for generic archive tools package: wsatools. See [2] for the latest version on-line.

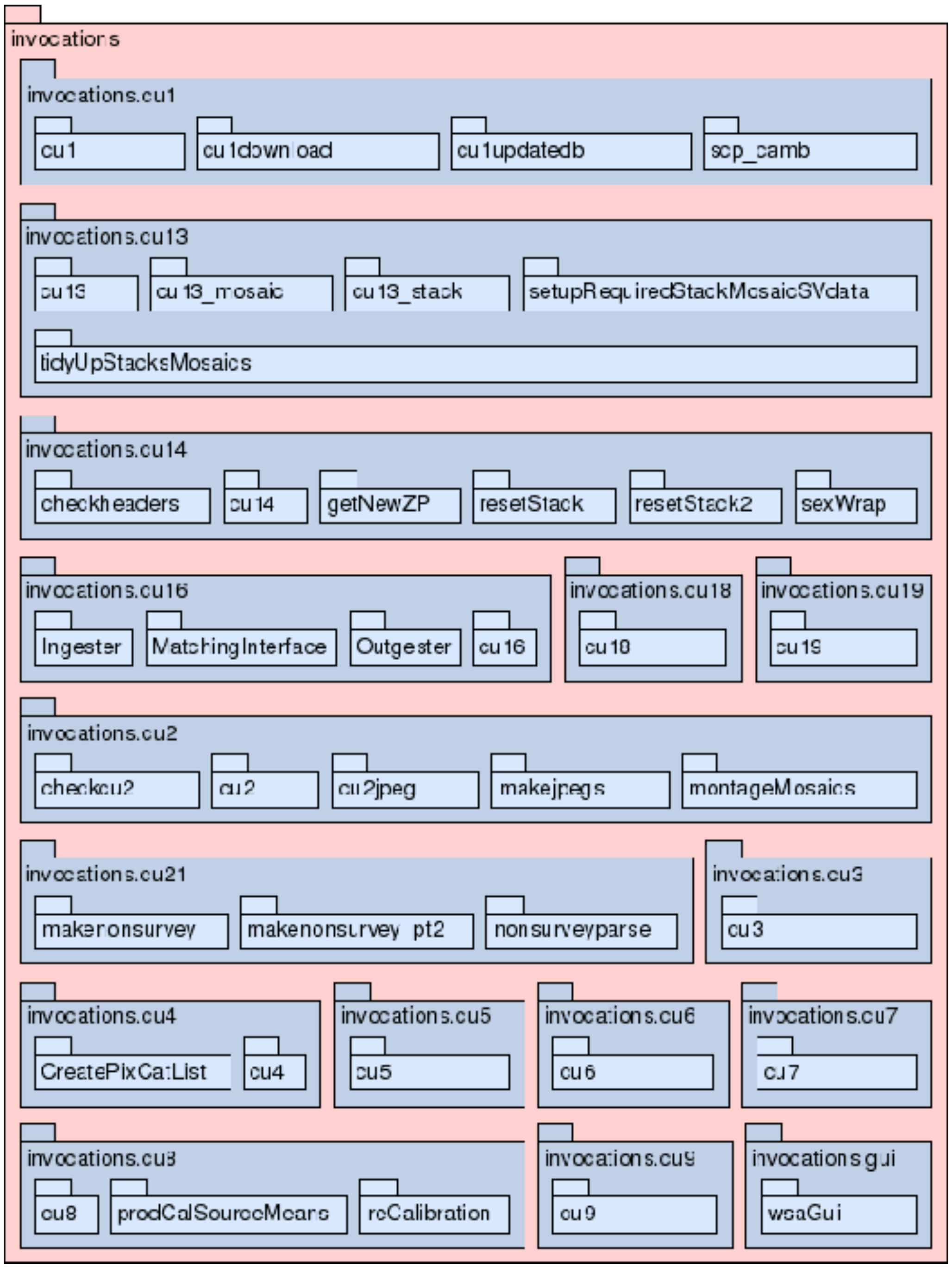

Figure 6: Package structure UML diagram for curation use case invocation scripts package: invocations. See [2] for the latest version on-line.

CURATION USE CASES

The VSA curation use cases, identified in the Database Design Document, are a set of tasks carried out by the archive operator. These encompass all processes by which data is ingested, processed and rendered in a form suitable for user access. For every use case that is to implemented as a software component we have developed a Python script. Table 2 provides a summary of every use case, highlighting their current development status. In this section we describe the design of each completed software component for performing curation use cases.

Table 2: Curation task components for the VSA, denoting current status. NB: CU19 & CU20 are combined into a single software component.

Curation Use Case	Current Status	Function
CU1	In WSA production use	Obtain science data from CASU
CU2	In WSA production use	Create library compressed image frame products
CU3	In WSA production use	Ingest science and compressed image metadata
CU4	In WSA production use	Ingest single frame source detections
CU5	Implemented	Create library difference image frame products
CU6	In development	Create/update synoptic source catalogues
CU7	In WSA production use	Create/update merged source catalogues
CU8	In development	Recalibrate photometry
CU9	In development	Produce list measurements between passbands
CU10	To be developed for final phase	Compute/update proper motions
CU11	To be developed for final phase	Recalibrate astrometry
CU12	No software component required	Get publicly released and/or consortium supplied external catalogues
CU13	In WSA production use	Create library stacked/mosaic image frame products
CU14	In WSA production use	Create standard source detection list from any new stacked/mosaiced image frame product
CU15	No software component required	Run periodic curation tasks CU6-CU9
CU16	In WSA production use	Create joins with external catalogues

CU17	To be developed for final phase	Produce list driven measurements between WFCAM/VISTA and non-WFCAM/VISTA imaging data
CU18	In WSA production use	Create/recreate table indices
CU19/CU20	In WSA production use	Release - place on-line new database product
CU21	In WSA production use	Release non-survey data

Included within the use cases are tasks involving database queries and operations that take place on the Load Server along with image processing and data analysis applications that need to be invoked on a Linux workstation. It was decided to implement all curation use cases on one workstation providing a single point of entry for management of these tasks. As discussed in section 3.2, one of the PCs that form a node of the mass storage system is used as the Curation Server.

A special curation use case is the `discovery' mechanism whereby the operator identifies which curation use cases need to be carried out at a particular time or when a given use case needs to be carried out. This `Curation Discovery Tool' will be implemented as a GUI tool for the operator, as the final stage to the development of the VSA software. For more details and current progress see section 7.

Table 3: Software components implemented by each CU.

CU	Python Components	C++ Components	External Libraries
CU1	ProgCuSession	None	mxODBC, HPN-SSH
CU2	ProgCuSession	wsatools.jpeg	mxODBC, CFITSIO, jpeglib
CU3	ProgCuSession, Database.Ingester, helpers.ProvenanceFiller	wfcamsrc.exmeta	mxODBC, PyFITS, CFITSIO, HTM, AST, SLALIB
CU4	ProgCuSession, Database.Ingester	wfcamsrc.exnumeric	mxODBC, CFITSIO, HTM, AST, SLALIB
CU7	SourceMerger, Database.Ingester, Database.Outgester	wfcamsrc.mkmerge, wsatools.pairing	mxODBC, SLALIB, Lambert
CU13	ProgCuSession	CASU.fitsio or SWARP	mxODBC, PyFITS, SciPy, NumPy, PyRAF, iraf.imstatistics
CU14	ProgCuSession	CASU.cirdr or SExtractor	mxODBC, PyFITS, SciPy, NumPy, PyRAF

CU16	ProgCuSession, Database.Ingester, Database.Outgester	Matching	mxODBC
CU18	ProgCuSession	None	mxODBC
CU19	ProgCuSession	None	mxODBC
CU21	ProgCuSession	None	mxODBC

CU1

The CU1 script transfers data marked as `OK_TO_COPY` from the CASU servers to the VSA archive file system. The software components implemented by this Python script are listed in table 3. A special version of the publically available SSH tools, called High-Performance SSH (or HPN-SSH), is utilised to the copy the data as it maximises the available network bandwith to maintain the best possible transfer speed. The script has the option to run multiple download threads to optimise the data transfer speed.

CU2

From a list of FITS images files, for a given date range and release version, the CU2 script, using the C++ module `wsatools.jpeg', produces compressed JPEG thumbnail images for every detector of the multi-extension FITS images (see figure 7). Following CU3, the location of the compressed image product is then inserted into the record for that detector in table MultiframeDetector. The software components implemented by this Python script are listed in table 3.

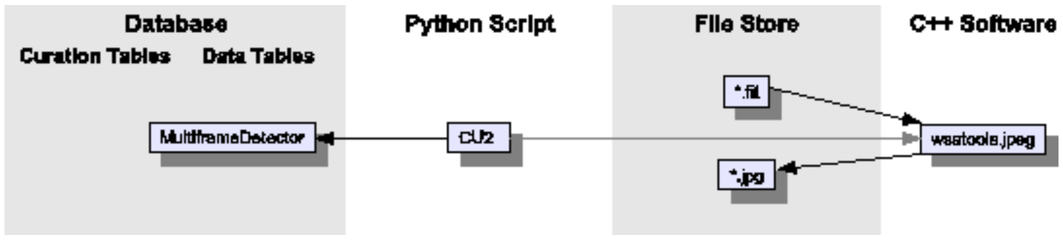

Figure 7: A flow-chart for CU2. Procedural flow is denoted by grey arrows, and data flow between the database and Python/C++ software components by black arrows.

CU3

CU3 takes the list of new files transfered to the archive file system by CU1 and extracts the file metadata to produce new records to insert into the Multiframe, MultiframeDetector, ProgrammeFrame and CurrentAstrometry database tables (see figure 8). For each of these files the invocation Python script calls the C++ programme exmeta (part of the wfcamsrc module, see section 4.4), which uses the schemas for each of these tables to determine how to construct the table data based on the data contained within the FITS headers (see section 6 for more information concerning schema driven features, and table 4 for the list of schema dependencies for CU3). The data is constructed in a ASCII file of comma-separated values, and then ingested into the database. The software components implemented by this Python script are listed in table 3.

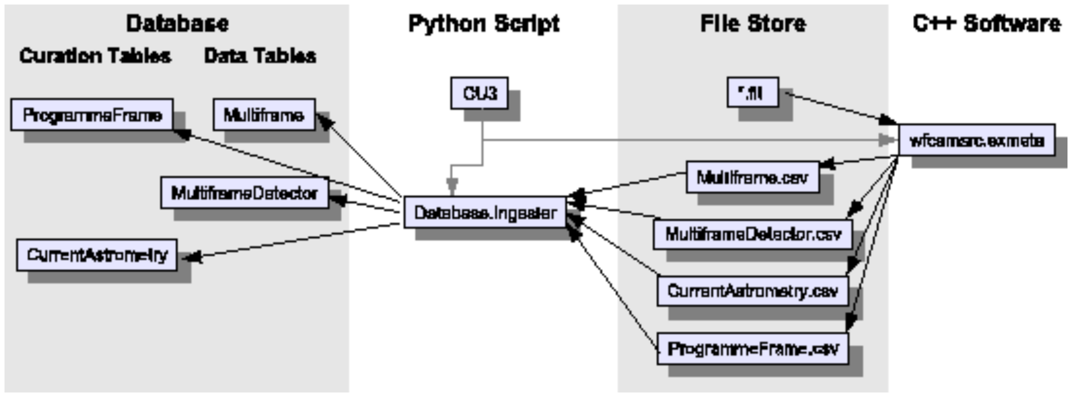

Figure 8: A flow-chart for CU3. Procedural flow is denoted by grey arrows, and data flow between the database and Python/C++ software components by black arrows.

Table 4: Schemas that CU3 depends upon.

Schema Name	Schema File	Schema Tables
Multiframe Schema	`VSA_MultiframeSchema.sql`	Multiframe, MultiframeDetector, Provenance
Curation Logs Schema	`VSA_CurationLogsSchema.sql`	ProgrammeFrame
Calibrations Schema	`VSA_CalibSchema.sql`	CurrentAstrometry

ProvenanceFiller

The set of multiframe data being ingested into the database at the time of CU3 invocation will not necessarily include every component multiframe used to produce a particular combined multiframe that is ingested into the database. Therefore, the provenance information for the combined multiframe is not added to the Provenance database table until a later stage when the complete set of component multiframes are available.

For this purpose a helper script, helpers.ProvenanceFiller, has been implemented to update the Provenance table. It searches the database (very quickly - less than one minute) to determine which of the existing combined multiframes listed in table Multiframe, have incomplete, or no, corresponding information in table Provenance. It then reads through every FITS file that corresponds to this list of multiframes to extract the provenance information from the FITS header keywords.

Combined multiframes are defined as those with a frame type equal to *stack*, *stackconf*, *leav*, *leavstack*, *leavconf*, *leavstackconf*, *deepleavstack*, or *deepleavstackconf*. We do not maintain provenance information for *dark* and *sky* frames in the database. This is because they may be combined from intermediate frames that are not ingested into the database.

CU4

CU4 takes the list of new files transfered to the archive file system by CU1 and extracts the binary catalogue data to produce new records to insert into the programme detection database tables (see figure 9). For each of these files the invocation Python script calls the C++ programme exnumeric (part of the wfcamsrc module, see section 4.4), which uses the schemas for each of these tables to determine how to construct the table data based on the data contained within the FITS binary table (see section 6 for more information concerning schema driven features, and table 5 for the list of schema dependencies for CU4). The data is constructed in a binary file for ingest into the database. The software components implemented by this Python script are listed in table 3.

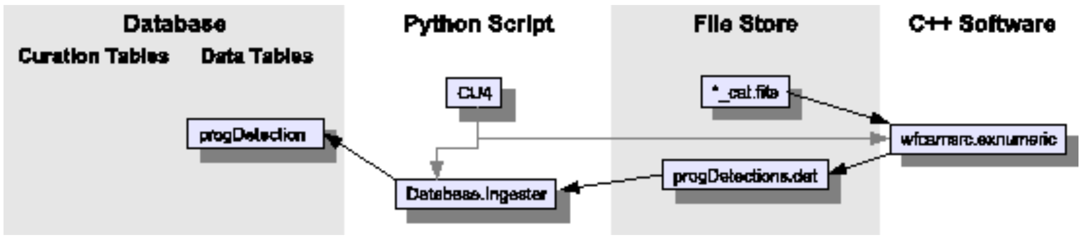

Figure 9: A flow-chart for CU4. Procedural flow is denoted by grey arrows, and data flow between the database and Python/C++ software components by black arrows. Here the detection table for a given programme `prog' is created.

Table 5: Schemas that CU4 depends upon.

Schema Name	Schema File	Schema Tables
Programme Schema	`VSA_progSchema.sql`	progDetection

CU7

The CU7 script is an implementation of the SourceMerger class (see figure 10, to produce the merged source tables for each survey programme. It queries the database to produce a list of multiframes corresponding to the same field of observation but in different passbands (a unique combination of filter and observational pass). The Database.Outgester class then outgests the programme detections for each passband for each frame set in this list. The individual detections in these tables are then paired by the wsatools.pairing C++ module, to produce a file listing the corresponding set of pointers between detections in each passband. These pointers are then used by the wfcamsrc.mkmerge C++ module to produce a merged source table, that can then by ingested into the database. A programme merge log table is then updated with the list of frame sets used to produce the programme source table. All CU7 components (listed in table 3), and especially the C++ source table creator software, mkmerge, are schema driven and depend upon the schema scripts listed in table 6.

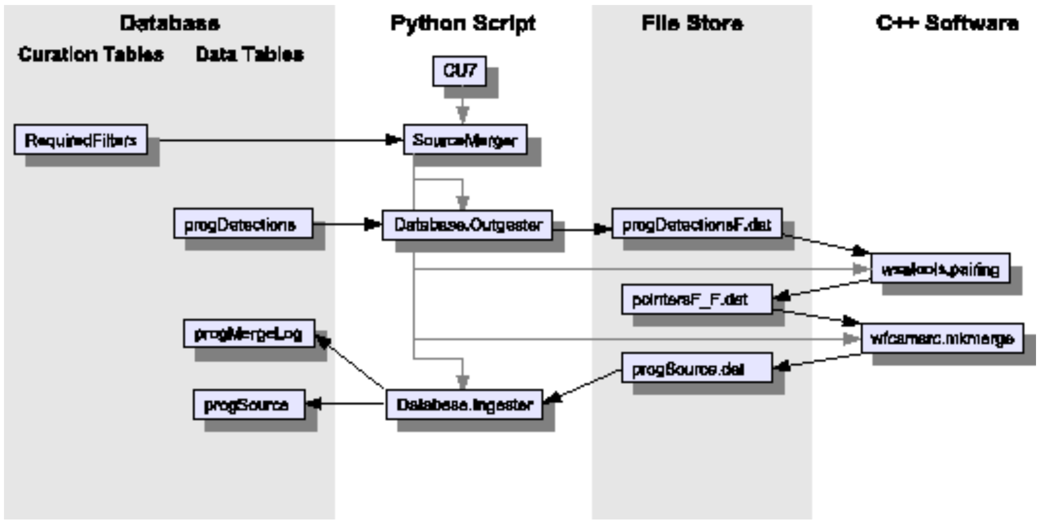

Figure 10: A flow-chart for CU7. Procedural flow is denoted by grey arrows, and data flow between the database and Python/C++ software components by black arrows. Here the source table for a given programme `prog' is created, from the detections of each passband, 'F', a unique combination of filter and pass.

Table 6: Schemas that CU7 depends upon.

Schema Name	Schema File	Schema Tables
Curation Tables Schema	`VSA_InitiateArchive.sql`	RequiredFilters
Programme Schema	`VSA_progSchema.sql`	progSource, progMergeLog

CU13

The CU13 script uses the CASU FITS image production tools, to create both the new deep stacked images and mosaiced images, for each programme that list entries in the database tables RequiredStacks and RequiredMosaics respectively (see figure 11). The software components implemented by this Python script are listed in table 3.

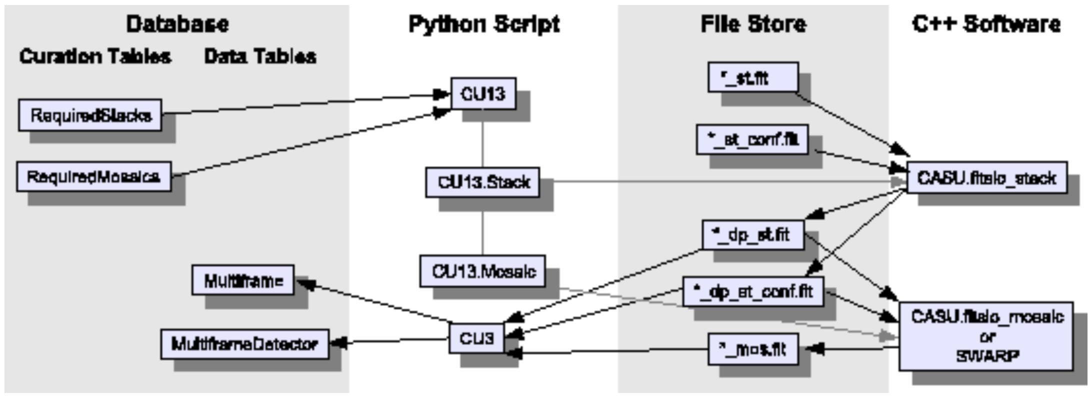

Figure 11: A flow-chart for CU13. Procedural flow is denoted by grey arrows, and data flow between the database and Python/C++ software components by black arrows.

The deep stacks are formed from a list of intermediate stacks of the same filter that meet the data quality criteria given in the RequiredStacks table. Furthermore, the provenance history of each intermedia stack is used to ensure that deep stacks are not formed from any duplicated frames. Both the new deep stacks and their confidence images are then produced using the `CASU.fitsio_stack` software tool, from the list of intermediate stacks and corresponding confidence images. The FITS header keywords of the deep stack image files are then prepared prior to database ingest by CU3, in the usual manner (described in section 5.3).

Mosaiced images from the same filter are produced from a list of the most recent adjacent deep stacked images. Unless specified otherwise by the RequiredMosaics table, the `CASU.fitsio_mosaic` software tool is used to construct the mosaiced images, which alters the background of each image to the same level based on the individual frame background measurements calculated by `imstatistics` an iraf software tool. If specified by the RequiredMosaics table, the external software tool, `SWARP`, may be used to produce the mosaiced image as an alternative to `CASU.fitsio_mosaic`. In our current design, if the mosaic is larger than 2 GB it is split into parts, the number of parts being an integer square such that each part has a size of approximately 1 GB. This is to overcome possible problems caused by the 2 GB file size limitation on 32-bit PCs. Finally, the FITS header keywords of the mosaiced image files are then prepared prior to database ingest by CU3, in the usual manner (described in section 5.3).

CU14

Source detection catalogues for the new image products produced by CU13 are created by the CU14 script, using the same source extractor tool as was used to produce the standard image catalogues from CASU. For ultra-deep stacks the SExtractor software tool is used an alternative. A set of source extraction parameters are retrieved from the RequiredStacks and RequiredMosiacs tables for the deep stack and mosaic images respectively (see figure 12, and table 3 for the list of software components used by CU14). Ingest of these new detection catalogues proceeds in the usual manner using CU4 as described in section 5.4.

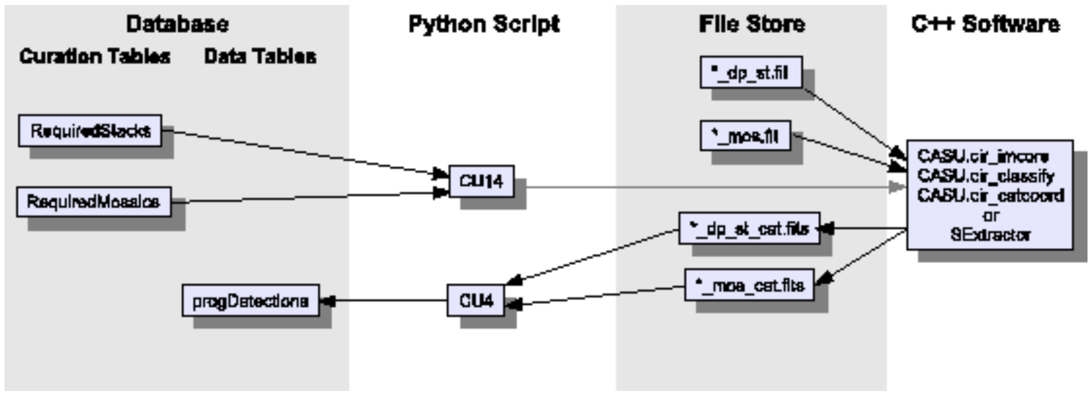

Figure 12: A flow-chart for CU14. Procedural flow is denoted by grey arrows, and data flow between the database and Python/C++ software components by black arrows. Here source detection catalogues for the new deep stack and mosaic image products for a given programme `prog' are created.

CU16

The CU16 script creates joins between the survey source tables and the external catalogue source tables, such as the SDSS, SSA, and 2MASS. The software components implemented by this Python script are listed in table 3, and the procedural flow diagram for this CU is illustrated in figure 13. The script is completely driven by the database table `RequiredJoins', creating neighbour tables with schemas defined in (`VSA_NeighboursSchema.sql`). The full list of schema dependencies is provided in table 7.

Table 7: Upon the addition of a new external catalogue neighbour join, the above schemas would require updating for CU16.

Schema Name	Schema File	Schema Tables
Curation Tables Schema	`VSA_InitiateArchive.sql`	RequiredNeighbours, ExternalSurvey, ExternalSurveyTable
Neighbour Tables Schema	`VSA_NeighboursSchema.sql`	Neighbour table definition

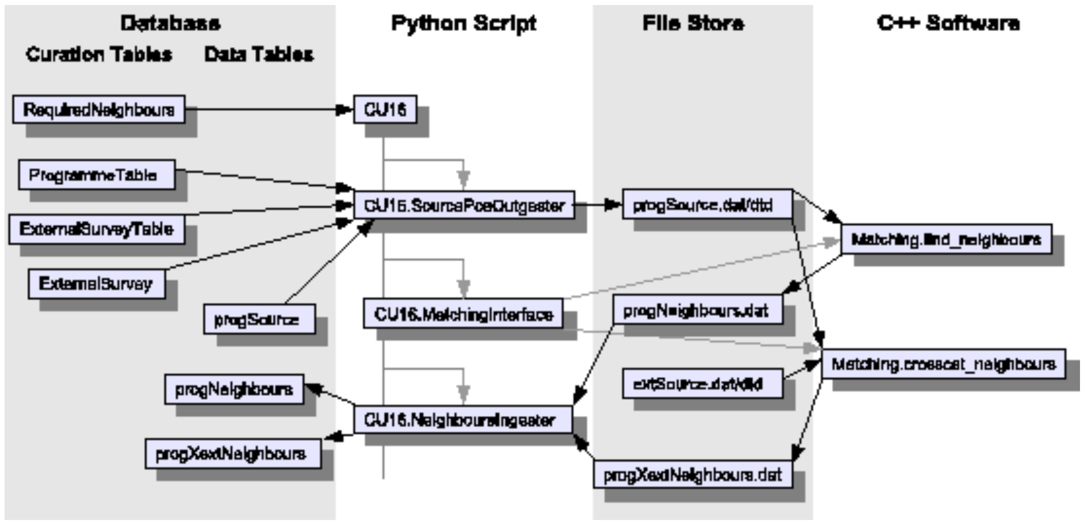

Figure 13: A flow-chart for CU16. Procedural flow is denoted by grey arrows, and data flow between the database and Python/C++ software components by black arrows. Here the neighbour tables for a given programme `prog' are created, consisting of a self-neighbour table, and various cross-matched neighbour tables with external catalogues `ext' that are stored as source position flat-file tables in the file system following an earlier outgest. To outgest external surveys the ExternalSurvey, ExternalSurveyTable tables are also queried prior to the outgest of the external catalogue's source table. In the case of producing a cross-match neighbour table with the SDSS survey, additional attributes are queried from the original source table and appended to the ingested neighbour table.

To produce these *neighbour tables* we utilise a customised version of a CSIRO-developed catalogue matching code, the C++ `Matching' code [1]. The algorithm used by this code, scales as O(N log M), where N and M are the respective sizes (i.e. number of rows) of the two catalogues, making it the most efficient method for comparing two extremely large catalogues. This method is particularly well suited when two catalogues are of comparable size as the largest catalogue dictates the overall performance of the code, as it always scales as N . On our system this software is I/O limited meaning the performance is completely dominated by the read / write speed of the data, with the CPU not being used to full capacity. [1]

The source matching code is external C++ software and so cannot access the database. Therefore,source matching is performed on outgest binary files containing just the unique source IDs and position co-ordinates. Binary files provide the best I/O access performance, and are much more efficient than database operations. These source position tables are permanently stored on the mass storage raid array for each of the external catalogues. Outgest of the VSA source postion tables for the current programme occurs upon each invocation of the CU16 script. All source position table files are sorted by declination on outgest - a requirement of the source matching algorithm. Also each binary file (extension `.dat') is accompanied by a corresponding data-type definition (extension `.dtd') ASCII file, that contains the field sizes for the source ID and co-ordinates fields. These can vary between external catalogues and so the code needs to know the field sizes.

CU18

The purpose of this curation use case is to ensure that every database table has the full set of required indices defined prior to public release. This list of indices for every table is maintained in an SQL script called `VSA_Indices.sql`. The CU18 Python script simply parses this SQL script, creating every missing index on each table.

CU19

A Python script invoking a series of database execution statements implements this CU; the public release of database products. Following a verification that the last entry in the ProgrammeCurationHistory table for each survey programme corresponds to the final required curation task, a release database is produced from a copy of the curation database on the catalogue load server. The copy is created by backing up the curation database, and then restoring its contents to the new release database, thus ensuring a backup of the curation database is made prior to release. This release database is then trimmed of any superfluous tables and data that are only required for curation purposes, before being copied to the public catalogue server. In addition to updating the both archive and programme curation histories, CU19 also updates a release history table, Release.

CU21

The software implementation of CU21, the preparation of non-survey, proprietary databases, consists of a series of Python scripts. These scripts take the registration details submitted through the archive web interface by the principal investigators of each new non-survey and produce SQL schema files for the creation of the relevant database tables for this non-survey. Entries are placed into the database to denote the new programmes (in tables Survey and Programme), and the SQL schema files for the curation tables that drive CU3 and CU4 are modified. Following successful completion of the CU21 scripts the ingest of the non-survey programme data by CUs 3 & 4 may occur in the usual way.

DATA FLOW

This section provides a collection of useful reference notes concerning the flow of data from its original format in the FITS file supplied by CASU to the final released format in the VSA public database.

Table 8: Comment tags used to annotate the SQL schema script files.

Table metadata	
`--/H`	Table header for schema browser.
`--/T`	Table text for schema browser.
Attribute format metadata	
`--/D`	Attribute description.
`--/C`	Attribute Unified Content Descriptor (UCD).
`--/U`	Attribute units.
`--/N`	Default value.
`--/V`	List of allowed specific values provided as a sanity check.
`--/R`	Range of values provided for range checking (deprecated).
`--/X`	Maximum value sanity check (deprecated).
`--/M`	Minimum value sanity check (deprecated).
Data source	
`--/K`	FITS header keyword from which to extract attribute value.
`--/F`	FITS binary table TTYPE label from which to extract attribute value.
`--/Q`	Name(s) of attributes used to compute this attribute (comma separated list of tablename.attributename, a self-referencing `--/Q` tag requires some additional special parsing)
`--/L`	HTM level to compute HTM index to.
Schema browser specifics	
`--/I`	Tooltip text for schema browser.
`--/G`	Glossary information is provided (tablename::attributename format)
`--/S`	Flag to tick a checkbox as a default; no parameter given
`--/O`	Omit the given FITS Keyword (`--/K`) in the browser description; no parameter given

```
CREATE TABLE MultiframeDetector(
-------------------------------------------------------------------------
--/H Details for individual detector frames that are part of a multiframe.
--/T Required constraints: primary key is (multiframeID,extNum)
--/T (multiframeID) references Multiframe(multiframeID)
-------------------------------------------------------------------------
multiframeID   bigint not null default -99999999,
               --/D the UID of the relevant multiframe
               --/C ID_FRAME
extNum         tinyint not null default 0,
               --/D the extension number of this frame
               --/C NUMBER --/Q fitsfile
cuEventID      int not null default -99999999,
               --/D UID of curation event giving rise to this record
               --/C REFER_CODE
compFile       varchar(256) not null default 'NONE',
               --/D Filename of the compressed image of this image frame, eg.
                   server:/path/filename.jpg
               --/C ID_FILE --/Q deviceID
julianDayNum   int not null default -99999999,
               --/D the Julian Day number of the UKIRT night
               --/U Julian days --/C TIME_DATE --/K UTDATE --/Q julianDayNum
               --/R 2453280,None
camNum         int not null default -99999999,
               --/D Number of WFCAM camera (1, 2, 3 or 4)
               --/C ID_PLATE --/K IMAGE.CAMNUM
filterID       tinyint not null default 0,
               --/D UID of combined filter (assigned in WSA: 1=Z, 2=Y, 3=J,
                   4=H, 5=K, 6=H2, 7=Br, 8=blank)
               --/C INST_FILTER_CODE --/K FILTER --/Q filterID
               --/V Z,Y,J,H,K,H2,1-0S1,Br,BGamma,Blank
runID          varchar(256) not null default 'NONE',
               --/D Name of CASU raw data file
               --/C ID_FILE --/K IMAGE.RUNID --/N NONE
creationDate   datetime not null default '31-Dec-9999',
               --/D Creation date/time of file
               --/C ?? --/U MM-DD-YYYY:hh:mm:ss.sss --/K IMAGE.DATE
               --/N 12-31-9999
imageExtName   varchar(16) not null default 'NONE',
               --/D Extension name in source FITS image file
               --/C ?? --/K IMAGE.XTENSION --/N NONE
skySubScale    float not null default -0.9999995e9,
               --/D Scale factor applied to sky subtraction image
               --/C ?? --/K IMAGE.SKYSUB  --/N -0.9999995e9 --/Q skySubScale
morphClassFlag tinyint not null default 0,
               --/D Image morphological classifier flag, set if the classifier
```

Schema-driven code features

The database schema is described by a set of SQL script files. From these files the database tables are created, updated, and documented on the public web server. They may either be directly ``played'' on the database, or parsed by the Python/C++ code. The database schema describes the data, its format and its purpose. Much of the code developed for the science archive uses these files to determine the correct format for the data that is to be inserted or selected from the database, and to create new database tables as required. We therefore describe our science archive curation code as being ``schema driven''. This way the schema is free to evolve without requiring additional code maintenance.

Presented in figure 14 is an example SQL script file, showing part of the `CREATE TABLE` procedure for the table, MultiframeDetector, which stores pertinent metadata for individual detector images (or frames) of the observed paw print (as opposed to metadata that is applicable to the entire pointing). In addition to the standard SQL directives that describe the data (attribute name, attribute type, allow null values, and a default value), we annotate each attribute with a series of comment tags (the ``--'' sequence denotes a comment in SQL). The full list of tags is given in table 8. These tags tell our software how to handle the data, most importantly where to obtain the data. For example the data in the database may just be a straight copy of the values given in the FITS header keywords. For some attributes these FITS keyword values are parsed and presented in a slightly different format. Some data is taken from binary catalogue data, other attributes are calculated from pre-existing attribute values. Specific attribute examples are provided in the following sections.

Programmes

Every observation should be assigned to a programme, and as such every FITS file contains a keyword, value pair giving the programme name. For the UKIDSS surveys these were in form `u/ukidss/las`, for the LAS (Large Area Survey), for example. However, the values provided by the FITS files would often contain additional character appended to the end of the expected programme name. In addition, every survey programme had a corresponding science verification programme with a name such as `u/ukidss/las_sv`. Non-survey programme names are given as `u/05a/3`, `u/05a/37` etc. Finally the calibration programme would present its name as either `CAL` or `UKIRTCAL`.

Due to the addition of random characters following the main survey names, we cannot identify programme names (and subsequently determine the correct unique programme ID for the ingest into the database) with just a simple algorithm that performs a direct comparison. Furthermore, a simple ``character string starts with'' or ``character string contains'' type comparison would fail to correctly assign the programme IDs for either the calibration programmes or the non-survey programmes. Therefore, in every case where we wish to identify a programme name from the original value supplied by the FITS header keyword, we use the following algorithm:

```
if programme name starts with "u/ukidss":

  if "_sv" in programme name:
      programme name is programme name with all characters after _sv removed

    else:

      if any of the survey names is in programme name then that survey name is
the programme name

    else if programme name starts with "u/ec/":

      if any of the programmes with names that start with "u/ec/" is in
programme name then that name is the programme name

    else if programme name is "UKIRTCAL":

      programme name is "CAL"

    else:

      programme name is programme name
```

This is applied to both the Python software and the C++ software.

THE CURATION DISCOVERY TOOL

One of the outcomes of the Critical Design Review identified the need for a use case discovery tool. This tool is in development and will eventually carry out a number of tasks including:

- view the state of the system at a given time
- list the tasks need to be carried out a given time
- determine whether or not a given use case needs to be carried out
- notify the archive scientist at any time a particular use case needs to be carried out.

A graphical user interface (GUI) has been developed, and screenshots of the GUI tool in use are presented in figures 15, 16, and 17.

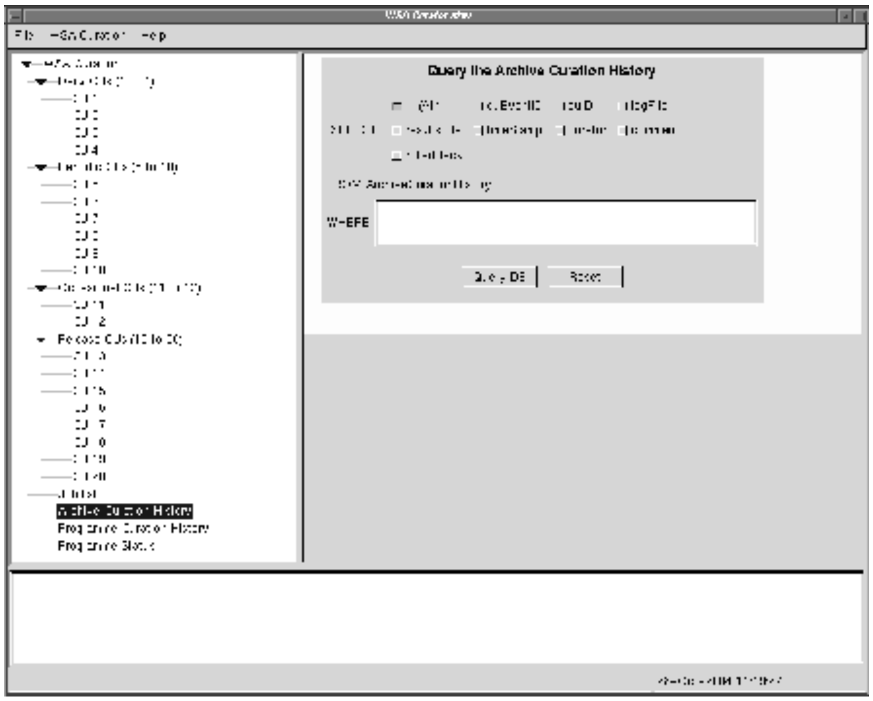
Query the Archive Curation History
WHERE
Query DB
Reset
Archive Curation History
Programme Curation History
Programme Status

Figure 15: The main window providing immediate access to the status of curation use case tasks, the present job list, individual programme status, and a query the archive curation logs.

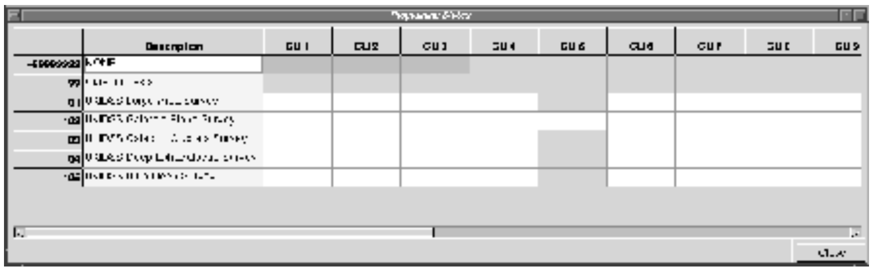

Figure 16: The Programme Status window listing the current status of every programme, detailing which curation tasks have been completed.

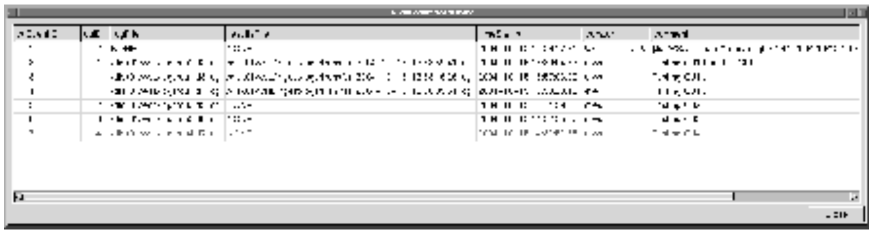

Figure 17: The curation log displayed by the query tool, presenting a list of the most recent completed curation tasks.

Bibliography

1
Abel, D. J., Devereux, D., Power, R. A., & Lamb, P. R. 2004, An O(N log M) algorithm for catalogue matching. Technical Report TR- 04/1846, CSIRO ICT

Centre, Canberra, Australia.

2
On-line VSA code documentation, http://www.roe.ac.uk/~rsc/wsa/

3
CVS repository for VSA code, http://cvs.roe.ac.uk/cgi-bin/viewcvs.cgi/?cvsroot=WFAU

ACRONYMS & ABBREVIATIONS

2MASS : Two-Micron All-Sky Survey
ADnn : Applicable Document No nn
API : Application Programming Inferface
ASCII : American Standard Code for Information Interchange
CASU : Cambridge Astronomical Survey Unit
CSIRO : Commonwealth Scientific and Industrial Research Organisation
CU : Curation Use Case
CVS : Concurrent Versions System
DBMS : Database Management System
FITS : Flexible Image Transport System
GB : Gigabyte
GUI : Graphical User Interface
HTM : Hierarchical Triangular Mesh
HTTP : Hypertext Transfer Protocol
ID : Identifier
I/O : Input/Output
JPEG : Joint Photographic Experts Group
NFS : Network File System
ODBC : Open Database Connectivity
OS : Operating System
PC : Personal Computer
SDSS : Sloan Digital Sky Survey
SQL : Structured Query Language
SSA : SuperCOSMOS Science Archive
SSH : Secure Shell
UCD : Unified Content Descriptor
UK : United Kingdom
UKIRT : UK Infrared Telescope
UML : Unified Modelling Language
VISTA : Visible and Infrared Survey Telescope for Astronomy
VSA : VISTA Science Archive
WFAU : Wide Field Astronomy Unit (Edinburgh)
WFCAM : Wide Field Camera for UKIRT surveys
WSA : WFCAM Science Archive

APPLICABLE DOCUMENTS

AD01	Hardware/OS/DBMS	VDF-WFA-VSA-006 Issue: 1.0 09/06
AD02	Database Design	VDF-WFA-VSA-007 Issue: 1.0 09/06
AD03	User Interface	VDF-WFA-VSA-008 Issue: 1.0 09/06

CHANGE RECORD

Issue	**Date**	**Section(s) Affected**	**Description of Change/Change Request Reference/Remarks**
1.0 Draft	01/09/06	All	New document
1.0	29/09/06	All	Released document

NOTIFICATION LIST

The following people should be notified by email whenever a new version of this document has been issued:

WFAU:	P Williams, N Hambly
CASU:	M Irwin, J Lewis
QMUL:	J Emerson
ATC:	M. Stewart
JAC:	A. Adamson
UKIDSS:	S. Warren, A. Lawrence

__oOo__

About this document ...

Footnotes

... capacity.[1]

As a benchmark an internal neighbour search of 10^9 sources within a radius of 10 arcseconds took ~ 5 hours on our 2.4 Ghz Pentium 4 PC.

Nigel Hambly 2006-09-30