CIWS DAS
Software Specification Document

Date:	25/02/2014	**Issue:**	0.7
Reference:	CIWS-OATS-TN-001		
Custodian	Marco Frailis		

Prepared by:		**Date:**	**Signature:**
List of Core authors	Marco Frailis Stefano Sartor Andrea Zacchei		
Contributors:		**Date:**	**Signature:**
List of All contributors			
Approved by:		**Date:**	**Signature:**

Contents

1 Introduction

The **Data Access System** (**DAS**) is part of the Customizable Instrument Workstation FrameWork software project, aimed at providing a framework, named **CIWS-FW**, for the storage, processing and quick-look at data acquired from space-borne and ground-based telescope observatories, to support the Assembly, Integration, Verification and Testing (AIV/AIT) activities on scientific instruments. The CIWS-FW should also facilitate reuse of the instrument workstation software components for the subsequent Commissioning and Operations phases to be carried out either in the mission Ground Segment of space-borne experiments, or in the Observatory site of ground-based telescopes.

The DAS system is a reusable software system that allows storage, retrieval and management of metadata and data acquired and processed by the instrument workstation (Level 1 data) or produced by the subsequent levels of data processing. It provides tools and application programming interfaces to:

- define the data model of a specific project, i.e. specify all the data structures to be archived and retrieved by the system, in terms of metadata attributes, relations with other data types and the data format and layout of the binary data;
- store the metadata and the association between data types in a relational database system, to allow query capabilities and transaction semantics on the data objects stored in the system;
- handle the persistence of binary data, ranging from small data objects to large binary data streams, and keeping consistency between metadata and corresponding data.

1.1 Purpose of the document

This document combines the output of the software requirements definition and the architectural design of the DAS system, according to the ESA guidelines for the application of software engineering standards to small software projects [RD1].

The software requirements in this document provide a technical specification of the user requirements specified in the CIWS User Requirement Document [AD1] and detail the higher level software requirements specified in the CIWS-FW Software Specification document [AD2].

1.2 Acronyms and abbreviations

AIV	Assembly, Integration and Verification
AIT	Assembly, Integration and Testing
API	Application Programming Interface
CIWS	Customizable Instrument Workstation Software
DAS	Data Access System
DDL	Data Definition Language
DMC	Data Management Component
DPS	Data Processing System

2 Model description

2.1 General description

The Data Access Subsystem (DAS) is a software system that will allow storage, retrieval and management of the L0 and L1 data produced by the Data Processing Subsystem (DPS). This system is mainly meant to be accessed by other software applications: the DPS pipelines, that will store the telescope or instrument raw and edited telemetry into the DAS, and the data clients, represented by data analysis tasks or data reduction pipelines (ref. FP7 proposal).

A possible approach to the architecture of the DAS system is the logical separation between two components: i) a DAS application programming Interface (DAS API), i.e. a set of interfaces available to any client application, in order to access and archive the instrument data; ii) the DAS Data Management Component (DAS DMC), that physically stores the data and performs the operations – query, read, write, delete - requested through the API. The DMC subsystem accesses a DBMS system (e.g. a relational database management system such as MySQL or Oracle). The rationale for this separation is that the interface seen by the client should be identical, no matter which physical implementation will be used (e.g. local file system or remote database). This approach has been used successfully in the Planck project, for the realization of the Data Management Component (DMC) of the LFI and HFI instruments.

Another critical abstraction needed in the DAS is the independence of the DAS API from the data model. This can be achieved by providing an abstract Data Definition Language (DDL) in XSD format (XML Schema Definition) in order to define the data structures needed for a particular project. Such DDL grammar should be able to define data structures analogous to those defined using the FITS format or the VOTable format, which are standards widely used in the astrophysics experiments. A DDL file is an XML file, using the DDL grammar, that defines all data structures (or types) used in a particular project. The DDL file defines the metadata and data format of each type.

The DDL has to provide an intuitive method for defining data structures. An object oriented approach simplifies this task and encourages the delegation of this activity to the end-users, in particular for the definition of the attributes of a specific type (keywords and columns) and the associations (relations) between different types.

The mapping from the DDL file data model to a relational database schema is part of the DASM subsystem. A possible solution consists of using existing Object-to-Relational Mapping systems (ORM). An example of an open-source ORM system, for the C++ language, is ODB (http://www.codesynthesis.com/products/odb/). The advantage of using an existing ORM system is that it already provides native interfaces to several DBMS systems, such as Oracle, MySQL, SQLite. Another solution is to explicitly define the mapping between the DDL data model and a relational data model, and then implement the interface to one or more DBMS systems using the ODBC standards or the native C API provided by each DBMS.

The DASI provides a simplified access to the data. Queries are performed on the metadata. A query returns the object identifiers satisfying the query expression. Then, the API provides functions to

open an object by its identifier, retrieve single keywords or single columns, set new keywords or append new columns, directly retrieve the associated objects.

2.2 DAS system overview

An overview of the DAS software system architecture is shown in the following diagram.

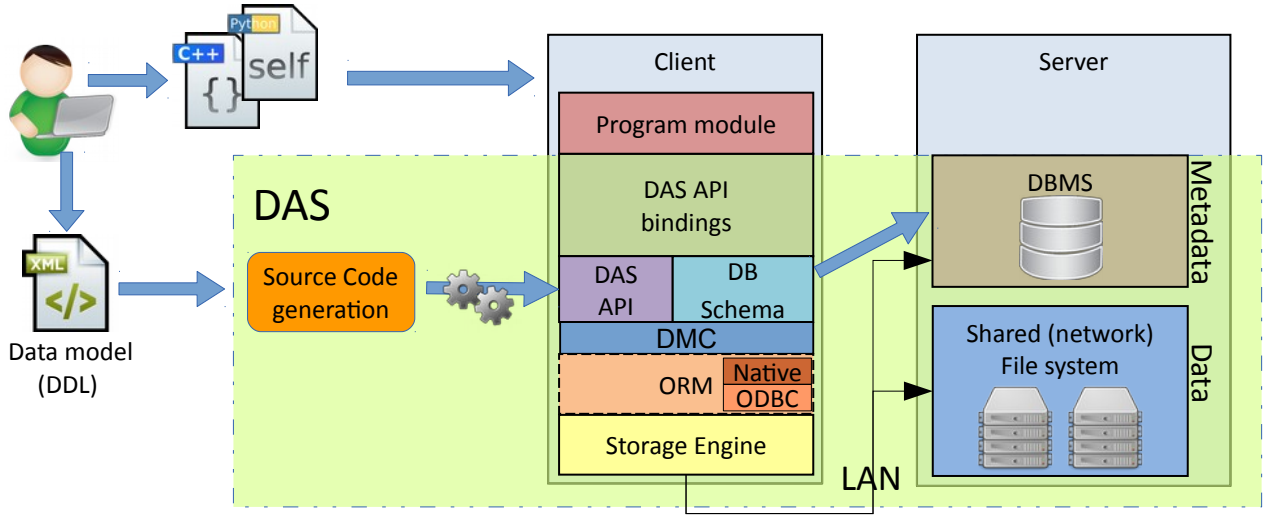


Figure 2.1: DAS system overview

On the client side, the program module can be written in different programming languages: C, C++, Fortran, Python, IDL. The DAS API can provide a common API in C/C++ language. The support to additional languages can be added by writing wrappers and adapters of the C/C++ API. This is also the solution adopted in the Planck/LFI DMC, where support to Fortran, IDL and Python has been added by wrapping the C API. The operations provided by the DAS API can be implemented by using an existing ORM system. In such architecture, the client communicates directly with the DBMSs, where the metadata are archived, i.e. no intermediate server processes, services or protocols are considered.

Data can be stored either in the DBMS, for instance as BLOBs (Binary Large OBjects), or as external files in a distributed or shared file system. For efficiency reasons, and depending on the final size of the project data, the communication between the clients and the server should be performed on a LAN. Data transfer between remote sites should be performed by using alternative synchronization mechanisms.

2.3 Data Definition Language

The DDL provides an XML grammar to define new data types that will be stored in the Data Access System.

The diagram below provides an overview of the language elements. A DDL data type definition includes two main components: metadata and data. The metadata section defines a list of keywords describing the data. The data section can define either a binary table or an image. These two

components are analogous to the header block and data block of the FITS standard. Since the data section is not mandatory, it is possible to define metadata-only data types.

Additionally, a data type can be associated to other correlated data types. A single association with a data type can also specify: i) a multiplicity, when many instances of the same type are involved in an association; ii) the relation type (shared, exclusive, extend), that affects the mapping of the association in the database schema. In order to define new data types by extending existing ones, in a data type definition it is possible to add a single parent type (ancestor).

By default, the DAS system stores the metadata in a back-end relational DBMS; the data is transparently persisted in a binary format (analogous to the VOTable binary serialization) in a shared file system. However, for small data sizes (small arrays or images), a DDL type definition can specify to store the data as BLOBs (Binary Large Objects) in the DBMS.

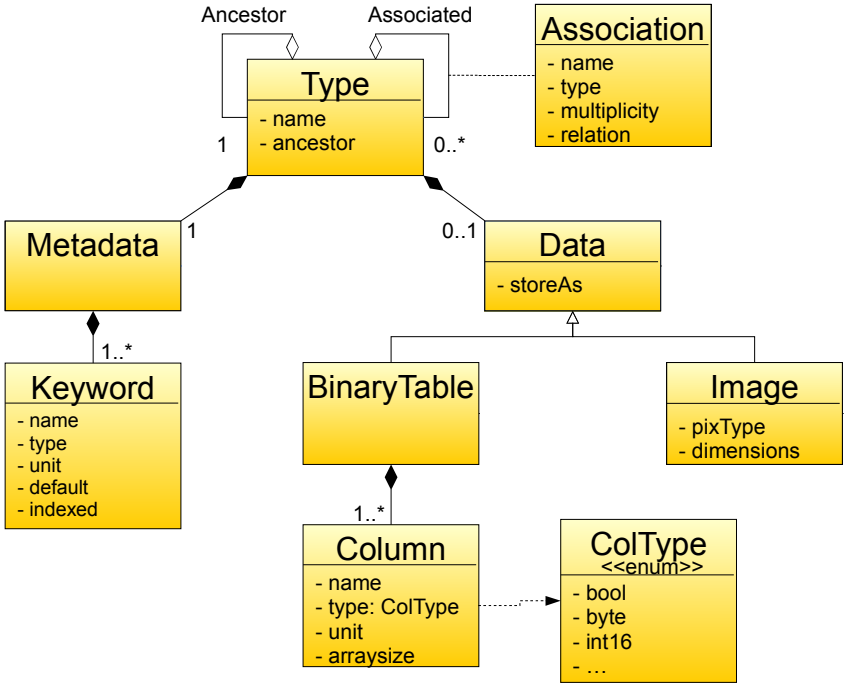


Figure 2.2: Data Definition Language elements overview

2.4 DAS Domain Model

The domain model provides a dictionary of terms used in the project and a common vocabulary to define unambiguously the use cases. According to the ICONIX process, it defines real-world (problem domain) objects and the relationships between them. The relationships are only specified as generalization (is-a) and aggregation (has-a) relationships.

We divide the DAS System domain model in four sub-domains:

- **Data type definition**: this domain defines the objects or terms related to the data type definition language of the DAS system;
- **Metadata dabatase**: it is the domain of the objects related to the storage and retrieval of the

metadata information of each data type instance, including query operations and transaction semantics.

- **Storage engine**: it groups the domain objects related to the persistence binary data of each data type instance.
- **System configuration**: all domain elements used to define the configuration of the DAS system, such as the files containing the definition of the user-defined data types, the specification of user account information, the database schema creation, etc.

Each sub-domain includes terms that are used in the definition of a behavioral requirement (a use case) or a functional requirement, in order to disambiguate the requirements definitions and verify the completeness of requirements coverage.

2.4.1 Data type definition

In the DAS system, user-defined data types are specified with a Data Definition Language (DDL). For each type, the user can define the Metadata as a list of Keywords. All data types must inherit (directly or indirectly) from the Essential Metadata data type, that includes mandatory Keywords, such as the data object name and version.

In a data type definition, the user can specify also the Associated Types, i.e. types providing correlated information (metadata and data).

The Data section of a data type definition specifies the data layout and format. The DDL of the DAS system provides data layouts and formats analogous to the FITS standard. Therefore, data layouts can be Binary Tables or Images. Binary tables are composed of simple Columns, where each element is a

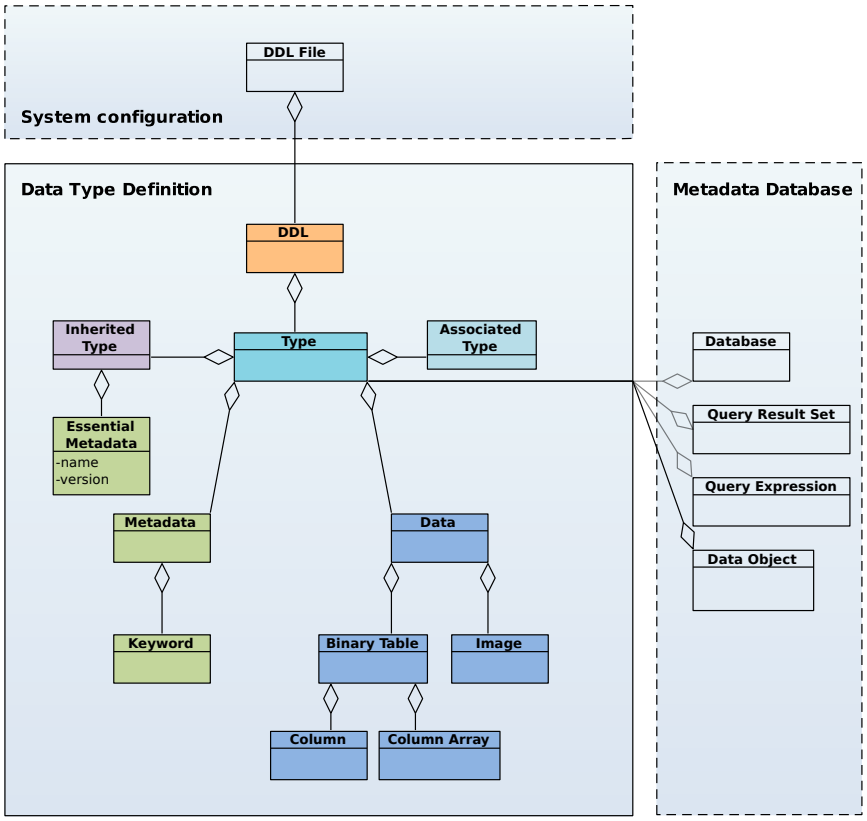


Figure 2.3: Data type definition domain

value of primitive type (integer, floating point, string), or a Column Array, i.e. a column where each element is an array.

The diagram in Fig. 2.3 shows the Data Type Definition domain objects and their relationships with the other domains. In particular, a user specifies the data types in a DDL file, i.e. an XML file in the DAS DDL format.

Database operations and queries are performed by specifying the user-defined Type (DDL type) to which the operation is applied.

2.4.2 Metadata database

The Metadata Database domain (see Fig. 2.4) defines the main elements of the DAS API. The Database object provides the primary interface to the back-end RDBMS (Relational DataBase Management System). It keeps information on the database instances (DB Info) accessible through the DAS system and the account information (Admin Account, User Account) needed to perform operations on the RDBMS. In particular, creation or modification of the database schemas can only be performed by the Administrator.

Database operations, when involving the back-end DBMS, are performed within a Session, which keeps an in-memory cache of the data type instances already retrieved from or persisted into the DBMS, and involve one or more database Transactions.

A Query is performed on a single DDL type at a time. The user has to specify a query expression on the keywords of the selected type or those of its associated types. The query returns a Query Result

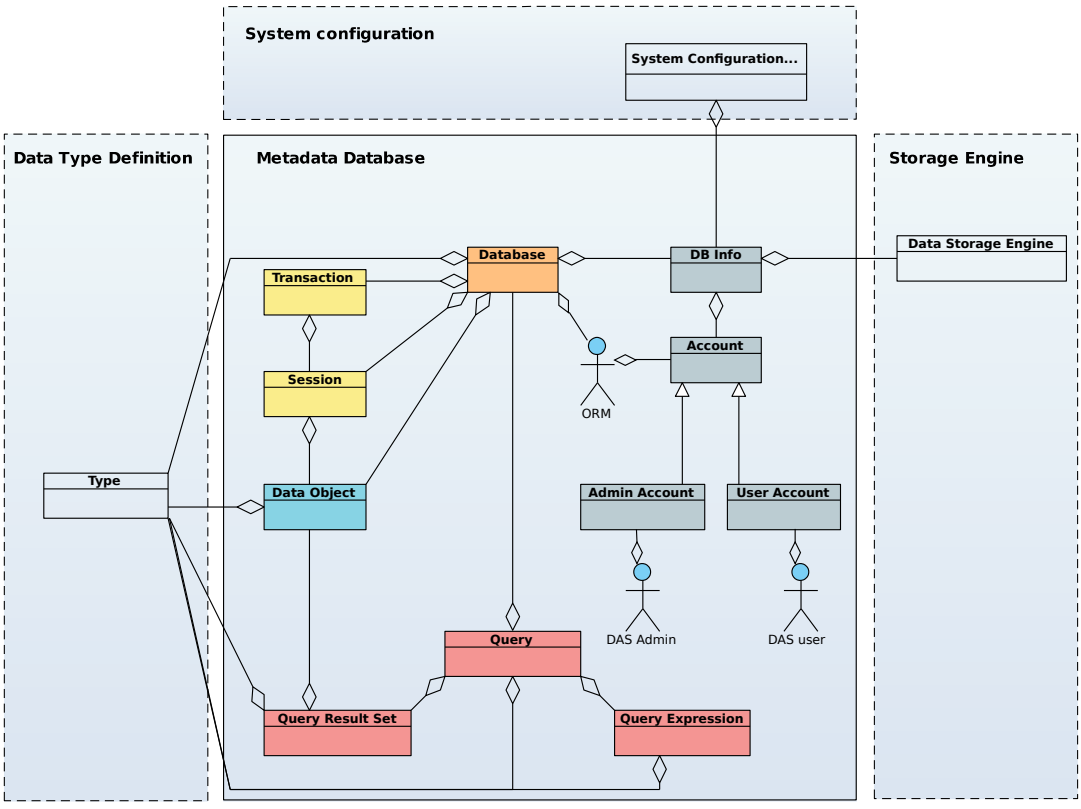


Figure 2.4: Metadata database domain

Set, with all Data Objects of the selected type matching the query expression.

2.4.3 Storage engine

The data section of user-defined types can be stored in different formats. The Data Storage Engine domain class defines the common interface to all formats (see Fig. 2.5). The main storage engine provided by the DAS system is a file based storage engine which persists raw binary data as files in a shared file system. An alternative persistence mechanism provided is a Blob Storage Engine, mainly targeted to small data types, which stores binary data as Binary large Objects (BLOBs) directly in the back-end DBMS.

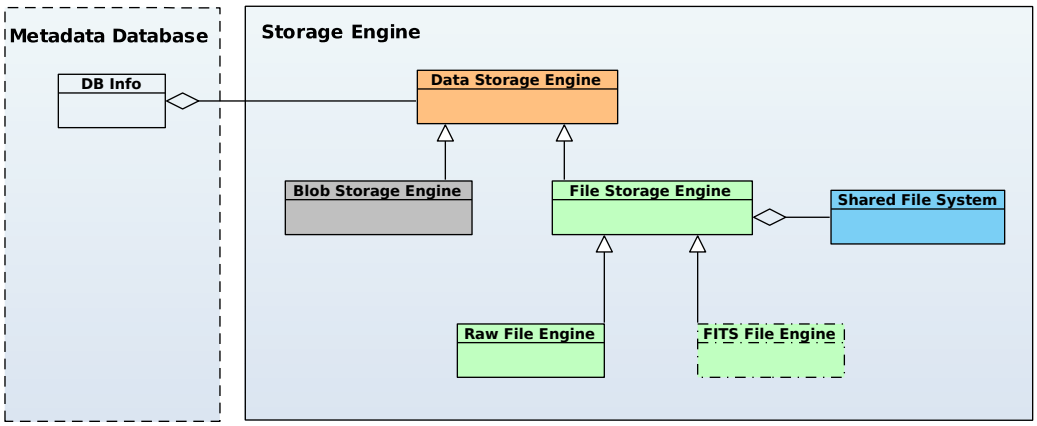


Figure 2.5: Storage engine domain

Additional storage engines can be added in the future, such as a file storage engine based on the FITS format.

2.4.4 System configuration

The DAS user must first create a DDL file containing the definition of the data types he will need to store into and retrieve from a DBMS. Another essential file is the DB Configuration file, where the user specifies the database instances where the data is stored. Each database instance definition will include a reference to a DDL file.

The System Configuration Manager of the DAS reads both configuration files to perform two tasks: i) a Source Configuration, which automatically generates the source code for each type defined in the DDL file; ii) a DataBase Configuration, which automatically creates the database schema of the DDL types for each database instance defined in the DB Configuration file. Mapping of the DDL types metadata and auxiliary data into a relational DBMS is performed with the help of an Object to Relational Mapping system (ORM). The first tasks requires the use of a generator (DDL ORM Generator) which parses the DDL files and generates the classes corresponding to the DDL types, the ORM directives necessary to persist instances of such classes and additional ORM helper classes to perform the database operations. The second tasks, implemented by the DB Schema manager, generates the database schemas.

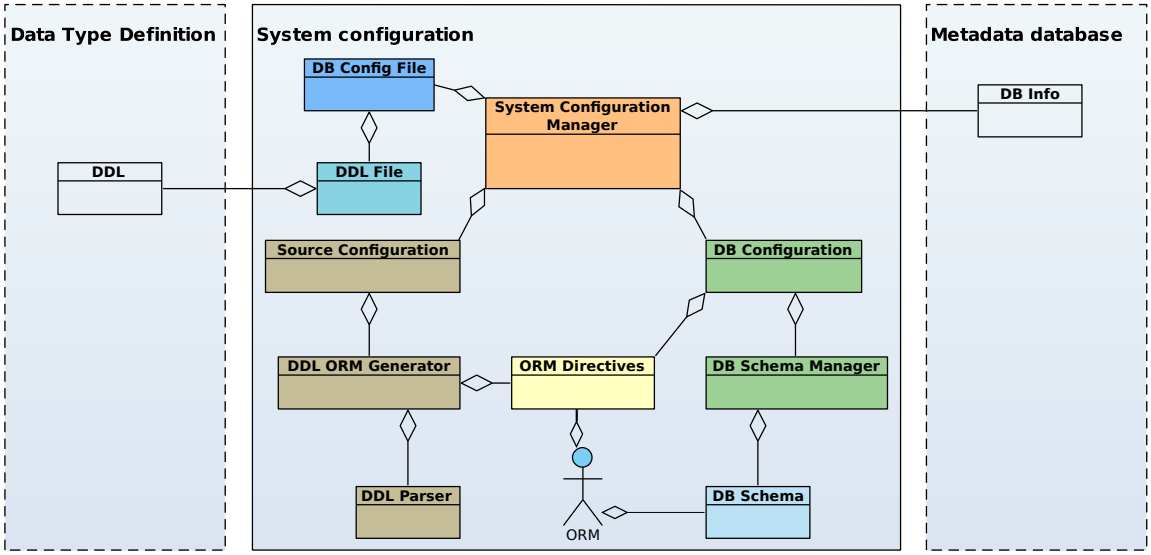


Figure 2.6: System configuration domain

3 Software requirements

3.1 Functional requirements

3.1.1 General requirements

DAS-SR-FN-GEN-10 **DAS client interface**

The client shall have access to the DAS operations only via the DAS API.

DAS-SR-FN-GEN-20 **Data definition language**

The DAS system shall provide a Data Definition Language (DDL) in XML format for the definition of the data structures to be stored by client applications. The language shall permit the definition of both metadata and data attributes.

DAS-SR-FN-GEN-30 **Data type definition**

A data type definition shall accommodate any combination of fields which is compatible with the allowed varieties of primitive data types (bytes, integers, longs, float and double precision real numbers, strings) and references to other data types instances.

DAS-SR-FN-GEN-40 **DAS API operations**

For each data type defined using the DDL, the DAS shall provide the following operations:

1. crate a new object instance conforming to the object's DDL data type definition
2. store the object (after the client has set all mandatory fields)
3. query objects of the same type that satisfy a boolean expression on one or more object metadata attributes
4. delete an object already persisted
5. track an object version, i.e. multiple versions of the same object shall be supported in order to keep an history of the object changes

DAS-SR-FN-GEN-50 **Persistence technology independence**

The DAS API shall hide the persistence technology used by the system in order to support multiple technologies (DBMSs, binary data formats, file systems).

DAS-SR-FN-GEN-60 **Database instances**

It shall be possible to connect to multiple database instances (DAS instances) simultaneously within the same process.

DAS-SR-FN-GEN-70 **Back-up**

The DAS shall provide a functionality to back-up and restore an entire DAS archive (Metadata and data).

3.1.2 Data Definition Language

DAS-SR-FN-DDL-10 **DDL mapping**

Mapping from the DDL types to the relational database tables should be stable and documented.

Maintaining a predictable mapping of the DDL to the relational database schema would support a direct, read-only access to the back-end database, using native or standard interfaces (e.g. ODBC).

DAS-SR-FN-DDL-20 **DDL changes**

Changes to DDL data type definitions shall be automatically validated for compatibility with the DDL grammar and constraints.

DAS-SR-FN-DDL-30 **DDL type information content**

Each data type definition shall have a unique name and include a metadata section and a data section. The metadata section shall contain fields of primitive data types. The data section can define either a binary table or an image.

DAS-SR-FN-DDL-40 **Data type inheritance**

The DDL shall support, in a data type definition, single inheritance from another data type definition. With the inheritance, a derived type can add new metadata attributes or new binary table columns to the base type.

DAS-SR-FN-DDL-50 **Data type associations**

The DDL shall support, in a data type definition, named references to other data types instances (1-1, 1-N, and M-N relations).

DAS-SR-FN-DDL-60 **Data type inline documentation**

The DDL shall support, in a data type definition, the possibility to add descriptions of the entire data type or of single references, attributes, table columns or images within the data type definition. Such feature can be useful for inline documentation of data types.

DAS-SR-FN-DDL-70 **Metadata attributes**

In the metadata section of a data type definition, each field definition shall include the following attributes: name, type, unit, optional default value, and if it is indexed or not (default: not indexed).

DAS-SR-FN-DDL-80 **Column attributes**

A binary table column definition should include the following attributes: name, type, unit, maximum string length in case of a string column, the array size in case the column elements elements are arrays.

DAS-SR-FN-DDL-90 **Multi-dimensional arrays**

Both column array elements and images shall support multiple dimensions.

3.1.3 Data objects

DAS-SR-FN-DOB-10 Essential metadata

For each DAS data object stored in the archive, the following information shall be available to the client: the object id (its primary key), name and version (such pair must be unique for each object), the data type of the object (its DDL type), the user (username) that as created or updated the data object, the modification date and time.

In the above requirement, an object name serves as a humanly memorable, identification string, analogous to a file name, that can be used to open a specific object by name or as a result of a query on metadata values.

DAS-SR-FN-DOB-20 Object version

The user shall be able to get the information about the highest version number of a persistent object in the data store. The user should have the possibility to store an object without specifying the version number; then the highest version number plus one is used for storing the object.

DAS-SR-FN-DOB-30 Objects hierarchy

The system shall guarantee that when a data object is stored, the whole hierarchy of objects associated to it (referenced by it) is made persistent and stored as well.

DAS-SR-FN-DOB-40 Circular references

Circular references between objects shall not be allowed (A refers to B which refers to C which refers to A).

DAS-SR-FN-DOB-50 Recursive deletion

The system should provide the possibility that when a data object is deleted, all referenced objects which are not associated with any other object are also deleted. The user shall be given the option to confirm the deletion by approving a list of all objects affected by the delete (if possible, using a graphical user interface, otherwise providing a command line program).

3.1.4 Database operations

DAS-SR-FN-RDB-10 Transaction semantics

Interaction between the DAS and the client shall apply transaction semantics, where a group of operations (query, read, write or delete operations) should either be completed in its entirety or not done at all. At least for metadata information, a transactional database should be used as a back-end.

DAS-SR-FN-RDB-20 Release memory resources

The client shall be able to perform "commit and clean", that is save the changes performed so far to persistent objects and cleaning them from its memory.

DAS-SR-FN-RDB-30 Objects locking

The DAS shall provide a data object locking mechanism of type "shared read/exclusive write or modify" (shared/exclusive locking). At least for metadata information, this mechanism should be

provided by the back-end database system.

DAS-SR-FN-RDB-40 **Query on essential metadata**

It should be possible to query the system for all data objects matching search criteria based on the following fields (which are general to all the objects): name, version, user, creation/modification date and time. Queries on user and creation date are considered useful only for maintenance purposes and do not need particular efficiency constraints or goals.

DAS-SR-FN-RDB-50 **Query on metadata**

For each data type, the specific data fields (other than the one listed in DAS-SR-RDB-40) for which a query and possible subsequent retrieval for that type of object can be performed shall be identified in its data type definition (in the DDL).

DAS-SR-FN-RDB-60 **Query on associated types**

It shall be possible to specify, in a query expression, conditions on the metadata of the associated types.

DAS-SR-FN-RDB-70 **Query with boolean and relational operators**

Query expressions shall support the use of both boolean and relational operators.

DAS-SR-FN-RDB-80 **Query with multiple databases**

If more than one DAS instance is available to the user, it shall be possible to specify the DAS instance for which the query is intended.

3.1.5 DAS API

DAS-SR-FN-API-10 **Database methods/functions**

The DAS API shall provide functions or methods to:

- connect to one or more DAS instances (and corresponding database back-ends);
- start a transaction;
- commit a transaction;
- query all objects of a selected data type that satisfy a query expression on its metadata fields;
- open a data object stored in the database using its name and version or the object ID.

DAS-SR-FN-API-20 **Data object methods/functions**

The DAS API shall provide functions or methods to:

- get or set any metadata attribute of an opened object;
- get or set the associated objects;
- for data objects containing a binary table, get any column data or append new data to a column;
- for data objects containing an image, get or set the image data, append new image tiles.

DAS-SR-FN-API-30 **Array slicing**

For objects containing a binary table or an image, it shall be possible to access part of a column data or part of the image data by specifying index ranges, without the need of loading the entire array into the client application memory.

DAS-SR-FN-API-40 **Metadata and data error handling**

The DAS API should provide mechanisms to trigger either a compile-time error or a run-time exception or a warning message in the following cases:

- Incorrect keyword name for the selected type;
- Invalid type conversion for the selected keyword. A warning in case of truncation;
- Incorrect column name for the selected type;
- Invalid type conversion for the selected column. A warning in case of truncation;
- Invalid type conversion for the selected image. A warning in case of truncation;
- Invalid range for the selected column or column element;
- Invalid range for the selected image;
- Invalid association name for the selected type;
- Invalid type for the selected association.

3.1.6 Data persistence

DAS-SR-FN-DPR-10 **Binary format**

The DAS system shall provide at least the following formats and technologies to store each binary table and image data:

- as a binary file in a shared file system;
- as Binary Large OBject (BLOB) in the back-end DBMS. For binary tables, a single BLOB for each column could be more flexible.

The DAS user shall be able to specify the data format as an attribute in the DDL data type definition.

Storing binary data as BLOBs in the DBMS should only be used for small data types. It is a DAS user task to identify which DDL types can use this persistence option.

DAS-SR-FN-DPR-20 **Persistence format extendability**

The DAS system should provide mechanisms to add new persistence formats, e.g. providing and abstract API.

New persistence formats could be required in the future, such as a FITS format.

3.1.7 Error handling and logging

DAS-SR-FN-ERR-10 **DAS API error handling**

When the DAS prevents the client from performing an illegal operation, an exception (error) code shall always be returned to the client.

DAS-SR-FN-ERR-20 **Error codes**

Each illegal operation shall be associated with a unique corresponding exception or error code (no generic exception/error codes).

DAS-SR-FN-ERR-30 **Logging levels**

The DAS shall be able to log the main operations performed through the DASI API. Several logging levels should be supported (Information, Warning, Error, Debug), that can be activated by a configuration file parameter.

DAS-SR-FN-ERR-40 **SQL logging**

It should be possible, for debugging purposes, to activate a tracing of the SQL queries performed on the back-end database system and the time requested by each query.

3.1.8 DAS configuration

DAS-SR-FN-CFG-10 **DAS configuration**

The DAS shall be configurable by means of configuration files, specific environment variables and, if possible, also by command line arguments.

DAS-SR-FN-CFG-20 **DAS configuration parameters**

Configuration parameters shall include:

- DBMS type (e.g. MySQL, Oracle);
- database instance connection parameters;
- path to the DDL file
- Logging level
- enabling of SQL tracing

3.2 Non-functional requirements

3.2.1 Interface requirements

DAS-SR-IF-10 **DAS API languages**

The DAS shall provide data access interfaces (directly or through a language binding) in the following programming languages: C++, C, Python.

DAS-SR-IF-20 **FITS/VOTable compatibility**

The DAS data type format shall be compatible (easily exportable) with the FITS and VOTable formats.

DAS-SR-IF-30 **Object/Relational Mapping**

The DAS system shall use and Object/Relational mapping system to map the DDL types and persist them in a Relational DataBase System (RDBMS).

DAS-SR-IF-40 **Query expressions**

Query expressions shall be based only on the information provided by the DDL data type definitions

and not on the database schema knowledge.

DAS-SR-IF-50 **Query using a GUI**

The DAS system should provide a GUI for browsing and querying the data according to the types defined in each DDL file.

3.2.2 Performance

DAS-SR-PF-10 **Maximum overhead**

With respect to directly reading/writing meta-data and data on files (e.g. FITS files) in a distributed/shared file system, the DAS shall introduce a maximum of 20% overhead.

DAS-SR-PF-20 **Throughput scaling**

The DAS system average throughput should scale linearly with the number of data objects in the database.

DAS-SR-PF-30 **Minimum simultaneous connection**

The DAS should be able to handle a minimum of 4 simultaneous client connections while maintaining the general performance requirements.

3.2.3 Security

DAS-SR-SE-10 **Access rights**

Access rights to a DAS instance should be based on the user management capabilities provided by the back-end database system.

DAS-SR-SE-20 **Account configuration**

The DAS should be able to read the user password from a configuration file stored in the user home directory.

3.2.4 Documentation

DAS-SR-DR-10 **API Documentation**

All DAS classes and operations shall be documented. For each operation, the minimum set of required documentation is the following:

- operation scope and purpose;
- data type returned by the operation;
- operation parameters with parameter descriptions;
- error codes (exceptions) generated by invalid cases;
- example of usage.

DAS-SR-DR-20 **In-line documentation**

It shall be possible to generate reference documentation automatically, from in-line comments and

information contained both in the DDL files and DAS source code.

4 System design

4.1 System components overview

An overview of the DAS system components is shown in Fig. 4.1 . The DAS user must first define a set of data types to be stored in the DAS system. These definitions are provided in input to the system as an XML file, named DDL file, which is associated to a back-end DBMS instance in the DAS configuration file.

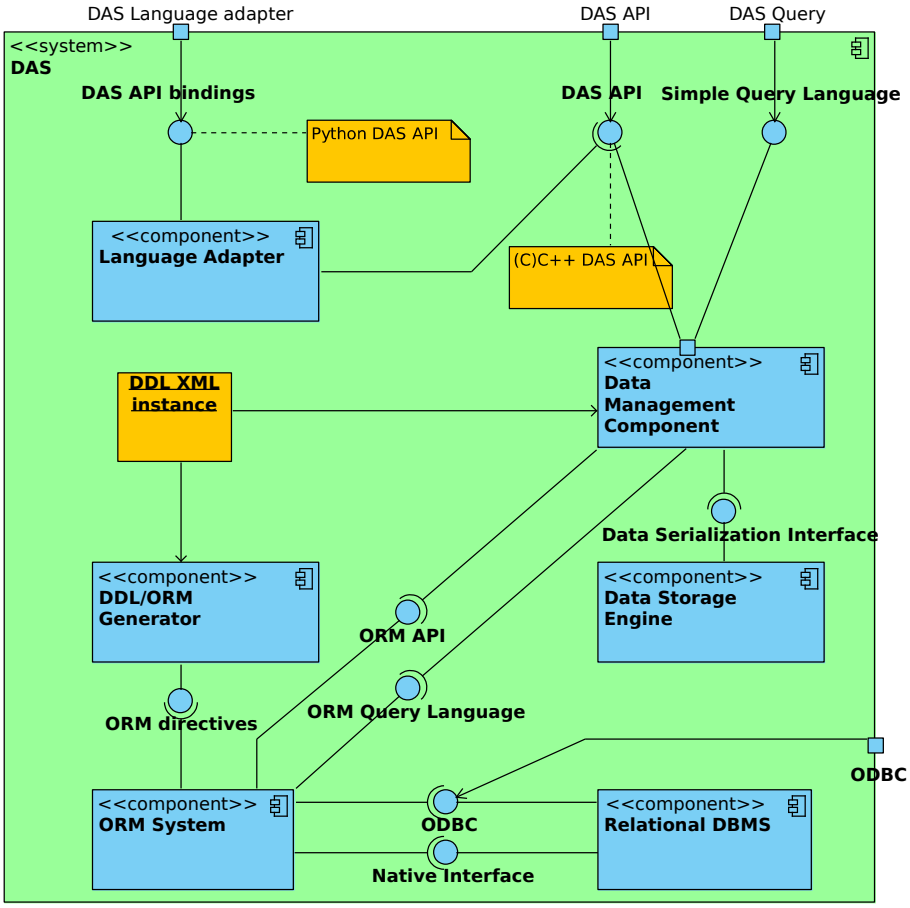


Figure 4.1: DAS system component model

The DDL/ORM Generator is the main component of the DAS Configuration Management. It parses the DDL file and automatically generates a C++ class definition for each DDL type. The C++ class definition also includes directives for the ORM system, in order to guide the mapping of the DDL type elements into the relational schema.

The ORM system parses the source files generated and, following the directives, creates auxiliary classes that transparently provide a conversion layer that maps between DDL objects in the application's memory and their relational representation in the database. Mapping of basic C++ types to suitable SQL types is automatic, even if it can be modified by the directives provided. In a separate

step, it also creates the relational database schema, to be imported in the DBMS. The ORM provides its own interfaces to the back-end DBMS. In particular, an object query language that fits the DDL classes and that requires no knowledge of the underlying relational schema.

The Data Management Component implements the DAS API, which supplies an higher level of abstraction with respect to the ORM API. In particular, the ORM deals mainly with the metadata section and the associations of the DDL types, while the data section, that can involve large binary data (binary tables and images), requires alternative persistence mechanisms (Data Storage Engines), such as external binary files stored in a shared file system and referenced by auxiliary information in the back-end DBMS. Hence, the DAS API maintains a tight coupling between the metadata and the data of a DDL type instance (a Data Object) by transparently keeping consistency between the two sections when transactions are committed and hiding the specific persistence mechanism used for each DDL type.

The Language Adapter component implements the bindings of the native DAS API (in C++) for a number of other languages or scripting languages. It contains the glue code between the different languages and adapts the DAS API constructs to those of the target languages. For instance, it exports C++ forward iterators (that can be returned by a query result) as Python iterators. Moreover, its tries to avoid as much as possible copying of memory buffers, by passing buffer references and using reference counters and custom deallocators.

4.2 Data Definition Language grammar

The DDL grammar has been introduced in section 2.3 and is also represented with some more details by the diagram in Fig. 4.2. The grammar is formalized in the XML Schema Definition language (XSD), fully detailed in Appendix A.

In the DDL, a data type **inherits** the definition of an existing type using the attribute *ancestor.* The inherited data definition elements (associations, keywords, table columns or image) cannot be modified. They can only be extended, by adding new associations, keywords or new binary table columns.

All DDL types must have as ancestor, directly or indirectly, the **essentialMetadata** type (default value), which provides few mandatory metadata keywords: *name*, *version*, *dbUserId*, *creationDate*. The name keyword is useful to have human readable identifiers for data objects (analogous to file names). However, all data objects have also an automatically assigned unique numeric identifier, the *das_id*, through which they can be retrieved more efficiently. The version keyword is a numeric value that enable keeping an history of modifications of a given data object so that a previous state con be recovered. The other two keywords, dbUserId and creationDate, are used for traceability purposes and are automatically set by the DAS system.

In a data type definition, **associated types** can be declared by specifying the association name. Additional attributes, the *multiplicity* and the *relation* type, help detailing the cardinality of both sides of the relation and select the best mapping in the relational schema.

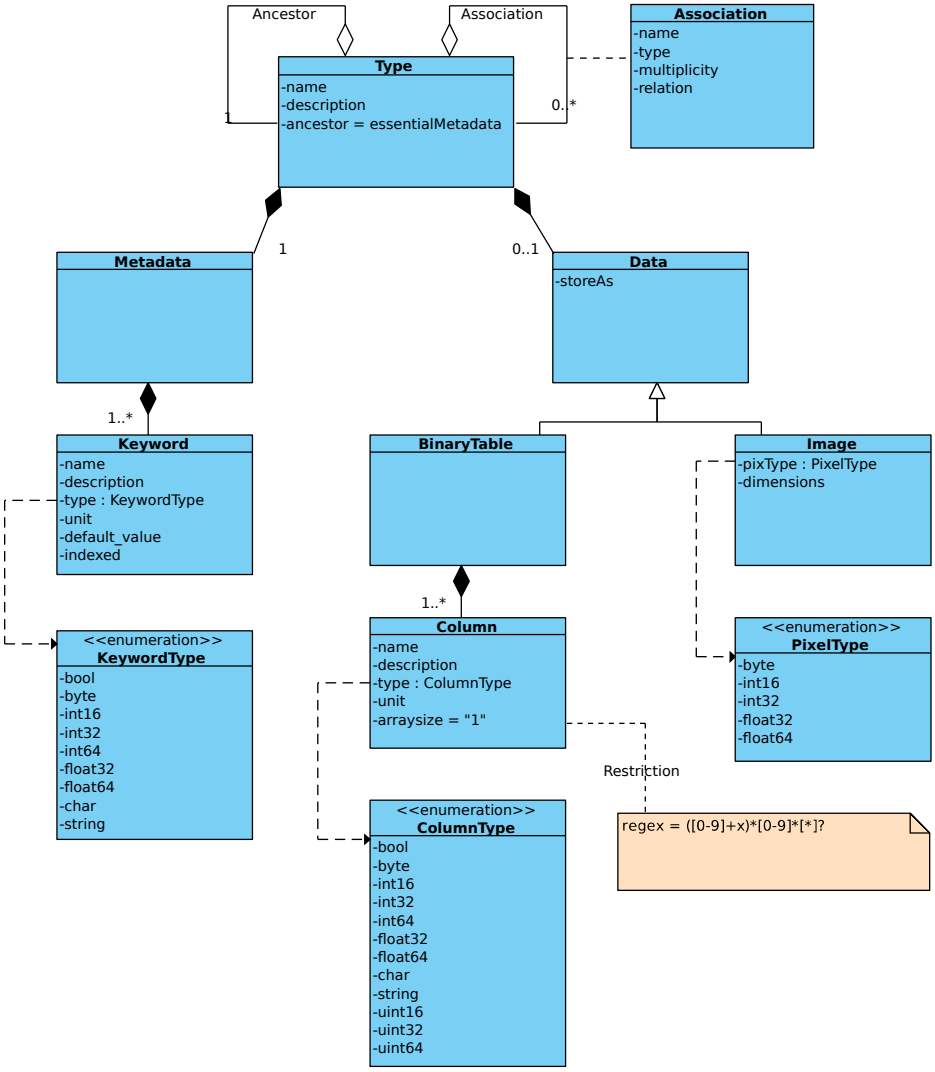


Figure 4.2: Data Definition Language grammar elements

The **metadata** section of a DDL type definition declares a list of keywords (metadata attributes) analogous to the FITS format header keywords. Usually, such keywords provide user-defined auxiliary information for the data section. In the DAS system, their purpose is also to provide coarse-grained information on the data, to reduce the amount of data indexed and stored in the RDBMS and to efficiently query and retrieve the persisted data objects.

The **data** section of a DDL type definition is not mandatory and can include either a binary table or an image. A **binary table** is defined by a set of **column definitions**. Apart from basic types, column elements can be defined also as fixed or variable length, multi-dimensional arrays, using the arraysize attribute.

Images are analogous to the FITS image data block and are defined by the pixel type and the number of dimensions.

For both binary tables and images the user can select, with the *storeAs* attribute of the *Data* element, which type of persistence the system should use. The current available options are: Binary Large

Objects (BLOBs) or external binary files.

4.3 Mapping the DDL to a relational schema

Mapping of the DDL to a relational schema is performed with the auxilium of an ORM system. In particular, ODB (www.codesynthesis.com) is an open-source ORM product with GPL and commercial license that has been selected for a number of critical features:

- It is entirely developed in C++, which is also the main language of the CIWS-FW;
- It supports several DMBSs, including MySQL, Oracle, PostgreSQL, SQLite, providing a common interface;
- It needs a reduced set of pragma-based directives to be added to a C++ class to enable its persistence in a DBMS;
- It has a clear set of rules to guide the object-relational mapping, including inheritance, relationships, table and column names, mapping of complex C++ types;
- It uses smart and weak pointers, to reduce memory load. It also provides lazy pointers for object relationships to have a finer grained control on relationships loading from the database.
- It supports mapping of core Boost and QT types.

In general, through the ORM system, each DDL type is mapped to a separate database table, where keywords are mapped to corresponding table columns.

4.3.1 Inheritance

Inheritance, in the DDL, is employed to reuse common data definitions in multiple types (*reuse inheritance*). DDL types using inheritance (e.g. from the essentialMetadata type) are mapped to completely independent entities in the database, with different object id spaces.

Another inheritance style is the so called polymorphism inheritance, where derived types are accessed through the common interface of the base type. Usually, types employing the polymorphism inheritance are mapped in the relational model to entities that share the object id space.

The DAS system will first support only the reuse inheritance, by creating completely independent entities in the database for each DDL type.

4.3.2 Metadata keywords

For each DDL type, the DAS systems maps the type to a relation (a table) of the relational schema. Each keyword is mapped to a column of the relation. The primary key, called **das_id**, is a surrogate key of type BIGINT.

The following is an example of a DDL type definition, named **rawDetectorFrame**, only showing the metadata section:

```
<type name="rawDetectorFrame" description="frame of a single detector chip">
  <metadata>
    <keyword name="detector" type="string" description="CCD detector identifier"/>
    <keyword name="gain" type="float64" unit="e-/ADU" />
    <keyword name="rdnoise" type="float64" description="amplifier read noise" />
```

```
    <keyword name="qe" type="float64" description="quantum efficiency"/>
  </metadata>
</type>
```

The corresponding table of the relational schema created by the DAS system is shown in figure 4.3:

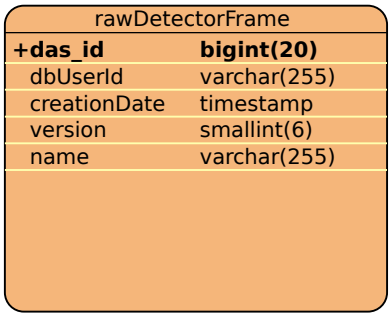


Figure 4.3: DDL type mapping example - metadata

The table name matches with the DDL type name. The attributes dbUserId, creationDate, name and version are by default inherited from the essentialMetadata type. The remaining attributes – detectotr, gain, rdnoise, qe – are mapped to suitable standard SQL types.

4.3.3 Associations

The DDL supports the definition of binary unidirectional relationships between DDL types. A DDL type definition may include several associations definitions. Each association definition specifies an association name, the associated DDL type name, the association multiplicity (one, many) and the relation type (shared, exclusive, extend). The multiplicity and the relation type define the cardinality ratio for the binary relationship.

The following example extends the rawDetectorFrame type definition by adding two associations:

```
<type name="rawDetectorFrame" description="frame of a single detector chip">
  <associated name="astrom" type="astrometryParams" multiplicity="one"
              relation="shared" />
  <associated name="stats" type="detectorFrameStatistics" multiplicity="one"
              relation="exclusive" />
  <metadata>
  ...
  </metadata>
</type>
```

According to the above definition, a rawDetectorFrame is associated to a single data object of type astrometryParams, providing the astrometric solution for the detector. Such astrometric solution is shared by many raw detector frames. Another association, called stats, provides the detector frame statistics computed for a particular raw detector frame. Figure 4.4 shows the corresponding entity-relationship diagram.

Mapping of DDL associations into the relational schema is performed according to common approaches adopted when mapping relationships of an ER model to a relational model. In particular:

1. **(one, exclusive)**: in this type of associations, the target type (e.g. the type

detectorFrameStatistics of the previous example) includes as foreign key the primary key of the source type (rawDetectorFrame);

2. **(one, shared)**: in this case, the source type includes as foreign key the primary key of the target type;
3. **(many, exclusive)**: analogously to the (one, exclusive) case, the target type of the association includes as foreign key the primary key of the source type;
4. **(many, shared)**: in this case, the relationship relation approach is adopted, i.e. a third relation (table) is created, cross-referencing the primary keys of the source and target types of the association.

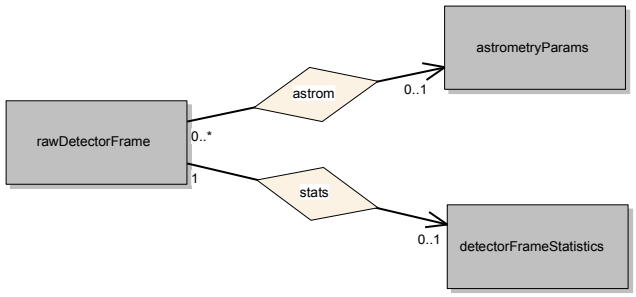


Figure 4.4: DDL associations: ER diagram for the rawDetectorFrame example

4.3.4 Binary tables

In the data section of a DDL type definition, the attribute *storeAs* specifies the type of persistence to be applied to data objects. The main types of persistence mechanisms provided by the DAS system are:

- as binary files in an shared file system, whose references are kept in the metadata database (storeAs = 'file');
- as BLOBs, in the metadata database (storeAs = 'blob').

The second mechanism is mainly meant for small data objects. The default persistence mechanism is the file-based one. Transaction semantics should maintain the consistency between the metadata and the data of a data object. This requires proper modeling, in the relational schema, of the auxiliary information associated to the data section of a DDL type definition.

In particular, when a data object column is read by a process and updated by another concurrent process, each one in its own transaction, the DAS system must guarantee the same isolation level provided for the data object metadata. One possible solution is the so called **multi-versioning**, analogous to the InnoDB multi-versioning, where a data structure (a table) keeps track of the old versions of a given binary column.

The following example shows the definition of a data type, named lfiHkDaeSlowVoltage, which includes a binary table.

```
<type name="lfiHkDaeSlowVoltage" description="DAE voltages - slow telemetry">
  <metadata>
    <keyword name="run_id" type="string" description="test run ID (e.g. FUNC_0001)"/>
    <keyword name="startTime" description="start time of the timeline in UCT" type="int64"
```

```
            unit="ms"/>
    <keyword name="endTime" description="end time of the timeline in UCT" type="int64"
            unit="ms"/>
    ...
  </metadata>
  <data>
    <binaryTable>
      <column name="sampleOBT" type="int64" unit="2^-16 s"
              description="sample ob-board time"/>
      <column name="sampleSCET" type="int64" unit="ms"
              description="sample time in UTC"/>
      <column name="LM151322Raw" type="uint16" unit="ADU"
              description="I1 LFI27 M1 - Raw value"/>
      <column name="LM151322Eng" type="float64" unit="V"
              description="I1 LFI27 M1 - Eng. value"/>
    </binaryTable>
  </data>
</type>
```

The corresponding mapping of this DDL type in the relational schema is shown in figure 4.5.

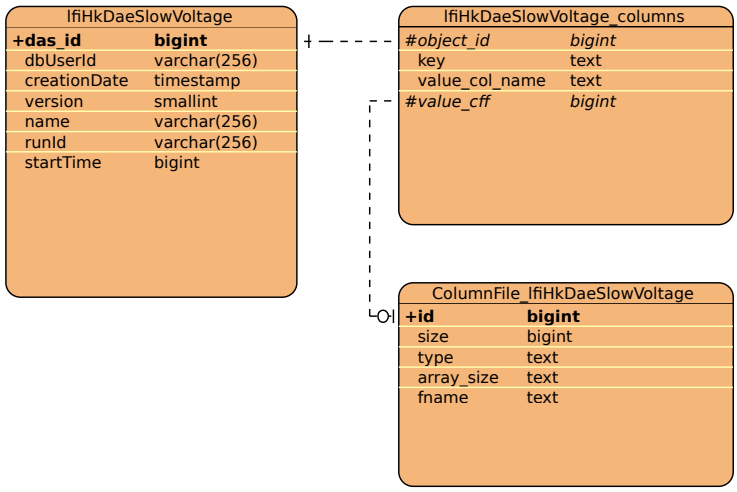


Figure 4.5: DDL type mapping: binary table example

The table lfiHkDaeSlowVolge contains the type metadata, analogous to the example show in section 4.3.2. For each specific column of an lfiHkDaeSlowVolge data object, the table ColumnFile_lfiHkDaeSlowVoltage contains information about each version of that column data file, such the id (primary key), the total size, the type of the column elements, the number and size of its dimensions (array_size), the path to the column file (fname). A third table, lfiHkDaeSlowVoltage_columns, binds a data object with the current version of each column data.

4.4 DAS System Configuration

4.4.1 Functions

The DAS System Configuration component is responsible for:

1. parsing the DDL files containing the user-defined data types;
2. for each DDL type, generating a C++ class corresponding to that type and adding the necessary ORM directives;
3. creating additional auxiliary C++ classes to obtain, at run-time, information for each DDL type
4. using the ORM compiler for generating the glue code with the ORM system library
5. using the ORM compiler to generate the database relational schema corresponding to the DDL types.

4.4.2 Main components

Figure 4.6 shows an overview of the DAS System Configuration component. The first active component is the DB Configuration Parser which runs the other components, passing the input data in agreement with the configuration file. The main components are the DDL-ORM Generator and the DDL Info Generator, which generate the C++ classes needed for the ORM compiler and the final library. The DDL Parser component reads the DDL files, validates them against the XSD schema and generates the DDL type tree instantiating each time the correct DdlNode class.

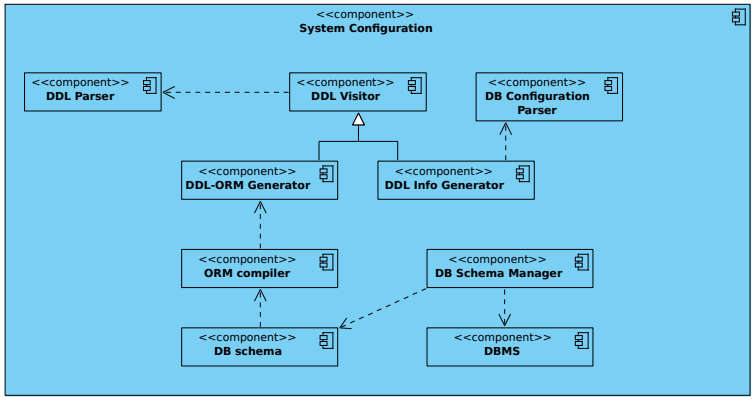


Figure 4.6: DAS System Configuration components

The Figure 4.7 exhibits the usage of the Visitor pattern. The DdlNode class plays the visitable object role. Each class derived from it represents a component of the DDL and drives the user-defined data type tree traversal invoking the proper DdlVisitor method implemented in the DDL Info Generator and in the DDL-ORM Generator.

The DLL Info Generator creates the C++ data structures to represent the DDL types in order to allow a runtime interrogation about the structure of a given DDL type. For what concern the metadata, the Generator stores all the keyword names and types. For the data instead, the Generator stores name, type and array size of each column if the DDL type has a binary table, and type and dimensions if the DDL type has an image data. The DDL-ORM Generator does most of the work generating the C++ classes representing the DDL types. For each DLL type it creates an header file and a source file. The header file contains the class declaration: members, methods and the directives needed by the ORM compiler, while the source file contains the implementation of the methods.

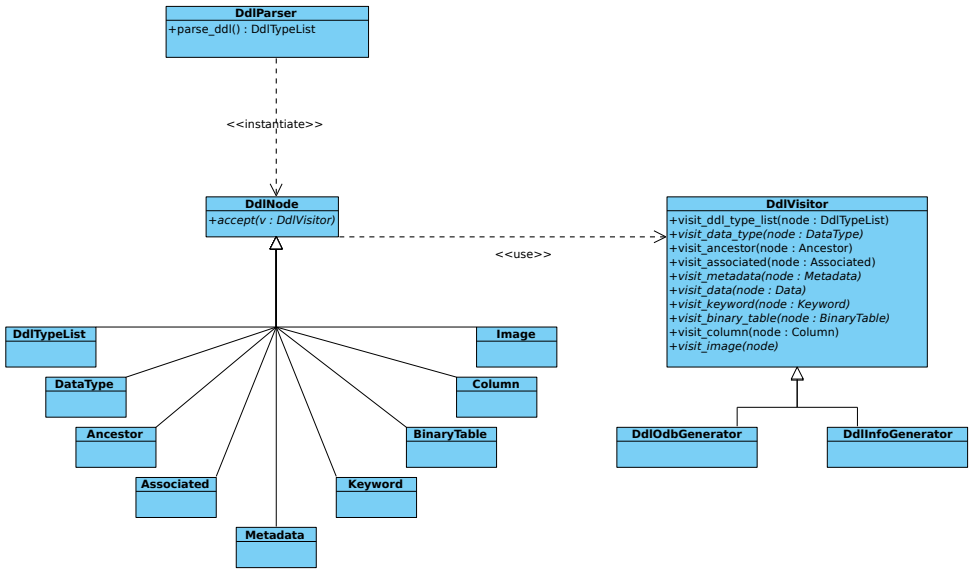


Figure 4.7: The DDL ORM

The ORM compiler extract the DDL type class structure and the directives from the header files; following those directives it generates the auxiliary C++ code needed for the interaction with the database. The code generated in this phase is for the internal use of the DAS System and no direct access of it is provided in the library. The ORM Compiler generates also the Datadase Schema files needed to store the DDL metadata. The DB Schema Manager reads those files and following the statements of the configuration file such as database address and port instatiates the schemas in the DataBase Management System.

4.4.3 Processing

The configuration system starts its processing with the DB Configuration Parser reading from the configuration file the database settings: database address and port, DBMS vendor, DDL file associated and the storage engine related settings. Then, the DDL Parser runs on the DDL files: it parses the file, validates it against the XSD schema and then it creates, for each one, a data type tree with a node of type DDLtypeList as a root. The trees are then validated against advanced constraints:

1. the keyword names of the derived types must not appear in the ancestor's keyword names;
2. a type derived from one which has a binary table data, must not have an image data and vice versa;
3. a type derived from one which has binary table data, may define new columns on the table,

but with different name;

4. the anchestor chain in a derived path must non contain loops;
5. in the types which contain associated types, the path induced by the association must non contain loops.

If one of the previous constraints ins't met, the process ends with an error message. A further constraint regarding all the trees together is checked: if two or more types with the same name are found, those must have identical structure, as say, the must be the same type. Once the previous checks finish, the the different tree roots are joined together to form an unique tree. This tree is passed to the DDL-ORM Generator and the DDL Info Generator which create the C++ source files.

At this stage, the ORM compiler can run and create the DB schema for the meta-data storage and the auxiliary C++ code needed in order to save and retrive the DDL data types meta-data to and from the database. The ORM directives located in the header files help the ORM compiler in the transaction from the C++ object model to the relational model of the traditional DBMS. Those directives can:

- identify which class member shold be used as a primary key;
- suggest which SQL type use to map a specific class member;
- choose whether a class should be mapped to its own table or as a composite object;
- declare a object member pointer as a foreign key;
- declare a class member to be transient in order to not save its state in the database;
- declare that a class member shall be indexed in the relational table.

The Figure 4.8 shows the steps performed by the DB Schema Manager. At the beginning the manager acquires the DDL representing the current DB schema and the file containing the new DDL. The new DDL must include all the previously DDL types with no changes, plus some new. Once the old and new DDL type consistency is checked (check missing types in new DDL and check type mismatches boxes in the Figure 4.8), the manager update the database schema inserting the new tables and the relavive constraints, and then updates the table containing the DDL.

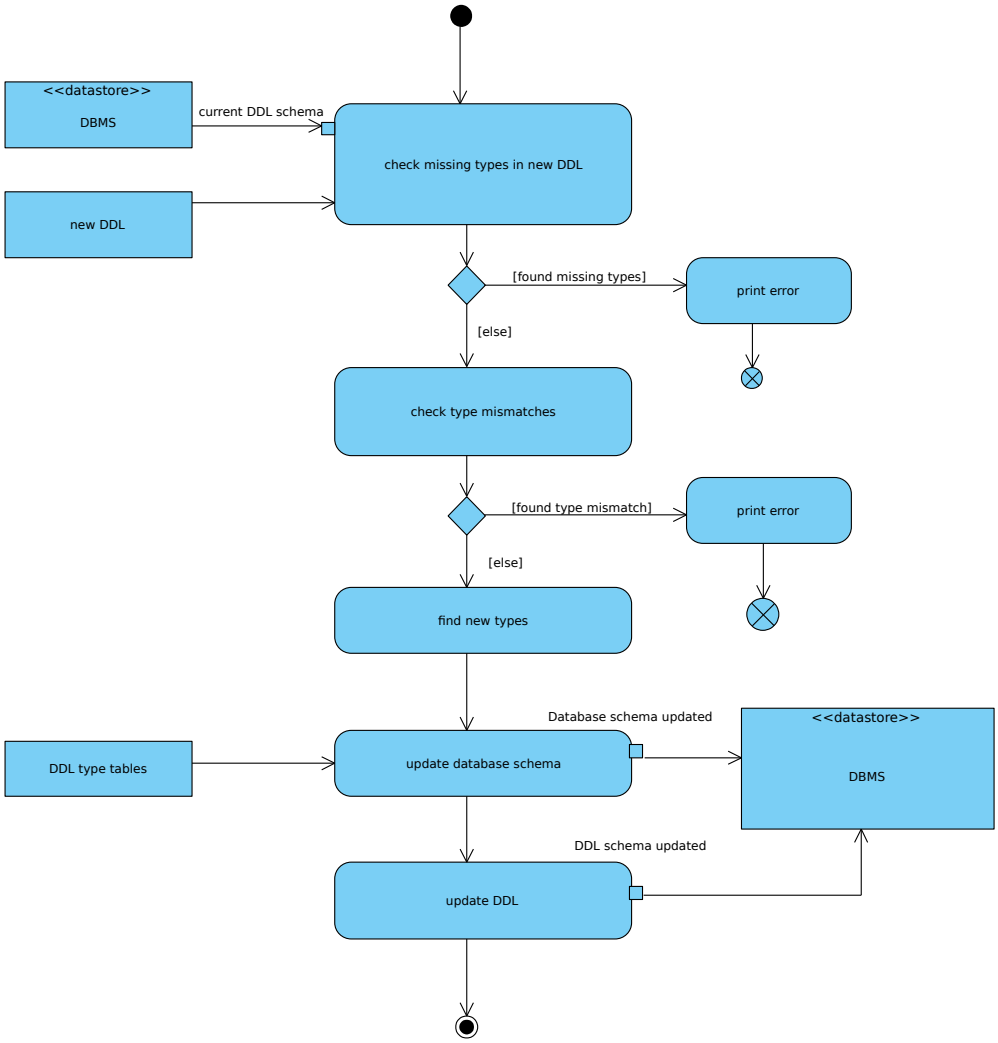


Figure 4.8: DataBase schema management

4.4.4 Data

In order to configure the DAS system the administator needs to provide 2 files: config.json, access.json.

4.4.5 config.json

The configuration file contains an array of objects. Each one contains a configuration for a different database. The file is parsed and validated during the configuration step (cmake command) through the json-schema file resources/config_schema.json. Note that each change to this file must be followed by a reconfiguration and consequet build of the DAS system in order to make them active.

```
[
  {
    "host"          : "localhost",
    "port"          : 3306,
```

```
    "db_type"       : "mysql",
    "mysql_socket"  : "/var/lib/mysql/mysql.sock",
    "alias"         : "test_level1",
    "db_name"       : "test_level1",
    "ddl"           : "ddl_level1_types.xml",
    "storage_engine": {
      "name"          : "Raw",
      "root_dir"      : "/mnt/data/test_level1/",
      "default_path"  : "%F/%t/%n_%v/",
      "custom_path"   : "%s/%B/%n/%v",
      "temp_path"     : "/mnt/data/temp/%$LOGNAME$",
      "unref_data_expiration_time" : 864000
    }
  },
  {
    "host"          : "localhost",
    "port"          : 3306,
    "db_type"       : "mysql",
    "mysql_socket"  : "/var/lib/mysql/mysql.sock",
    "alias"         : "test_level2",
    "db_name"       : "test_level2",
    "ddl"           : "ddl_test_types.xml",
    "storage_engine": {
      "name"          : "Raw",
      "root_dir"      : "/mnt/DBs/data/test_level2/",
      "default_path"  : "%F/%t/%n/%v/",
      "custom_path"   : "%s/%B/%n/%v",
      "temp_path"      : "/mnt/DBs/data/temp/%$LOGNAME$",
      "unref_data_expiration_time" : 60
    }
  }
]
```

Each of the properties shown in this example are mandatory:

- host : the hostname or ip address where the database is located
- port : the listening port of the database
- db_type : the database vendor, currently only mysql is supported
- mysql_socket : the location of the mysql pipe file to use when connecting to local MySQL server
- alias : human readable name to use in the code to refer to this database
- db_name : the name of the database to access for storing meta-data
- ddl : the relative path of the file that contains the DDL for this database
- storage_engine : this object contains the configuration of the data storage engine. Except for name, all other properties change for each engine type. Please refer to the storage engine specific documentation in the following sections. Note that currently we provide only the Raw storage engine.

4.4.6 access.json

This file must exists in the $HOME/.das/ directory of the user and contains the credentials to access the databases. This file is read run-time therefore changes on them may be performed without any further configuration.

```
[
  {
    "alias"     : "test_level1",
    "user"      : "foo",
    "password"  : "bar"
  },
  {
    "alias"     : "test_level2",
    "user"      : "foo",
    "password"  : "secret"
  }
]
```

- alias : the database alias that match the one specified in the config.json file.
- user : the username of the database.
- password : the password for that database.

4.5 Metadata Database

4.5.1 Functions

The Metadata Database component is responsible for:

1. handle database connections;
2. manage database transaction;
3. store and retrive DDL object's meta-data;

4.5.2 Main components

The Figure 4.9 shows the components involved. The top four components: Database, DdlInfo, Transaction, DasObject; are those exposed by the API and therefore accessible to the user.

The Database component provides an high level API to access to the DBMS. It uses the DdlInfo component which provides at run-time the configuration information about database vendor, database name, address, port and user credentials. The database component, odb subcomponent, provides a low level interface to the DBMS. This, is used internally by the Database which hides most of the complexity.

In a similar way, the Transaction component, exploits the odb transaction interface, to provide a simple API that handles a DBMS transactions. Both the Database and the Transaction components use the Resource Acquisition Is Initialization (RAII) programming idiom which aids a correct use of the database resources such as connections, transactions and locks.

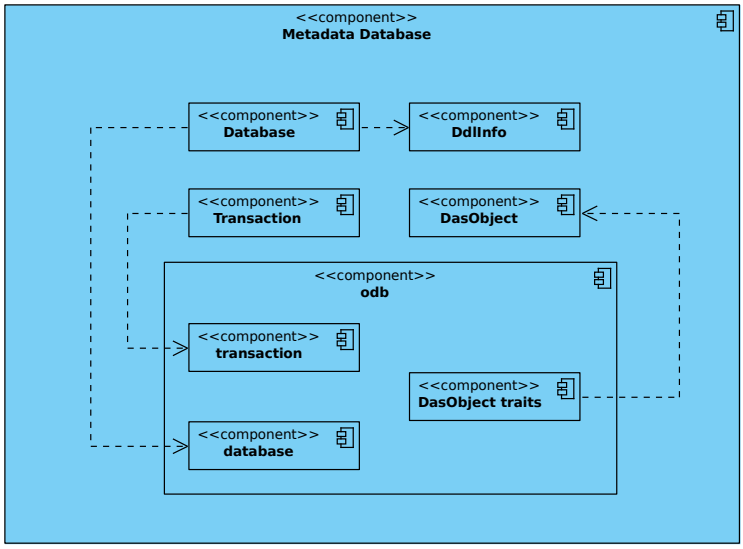


Figure 4.9: Metadata Database

The DasObject traits component prepares, using the prepared queries mechanism, the statements representing the DasObjects to be submitted to the database.

4.6 Storage Engine

4.6.1 Functions

The Storage Engine component is responsible for:

1. write object's data
2. read object's data
3. append new object's data
4. maintain consistency between data and meta-data of the same object
5. allow concurrency access of the system

4.6.2 Components

The Figure 4.10 shows the components and the interfaces which compose the Storage Engine. The StorageTransaction and StorageAccess represent the common interface which different kinds of storage engine implement. The Das system provides two storage engines: the blob storage engine which uses a DBMS as a back-end for storing the data and a so called raw storage engine which uses a shared file system to store the data.

The Transaction component takes advantage of the public methods provided by the StorageTransaction class (Figure 4.11), to involve the storage engine of that object in the current database transaction.

In the Figure 4.10 illustrates the information exchange between the DasObject and the StorageAccess. The first provides details about the kind of data which can be stored in that objects: column names and relative types in a binary table or pixel type and dimensions in an image. The second instead, provides actual data access which the DasObject present, hiding the low level details, in the interface showed in Figure 4.11.

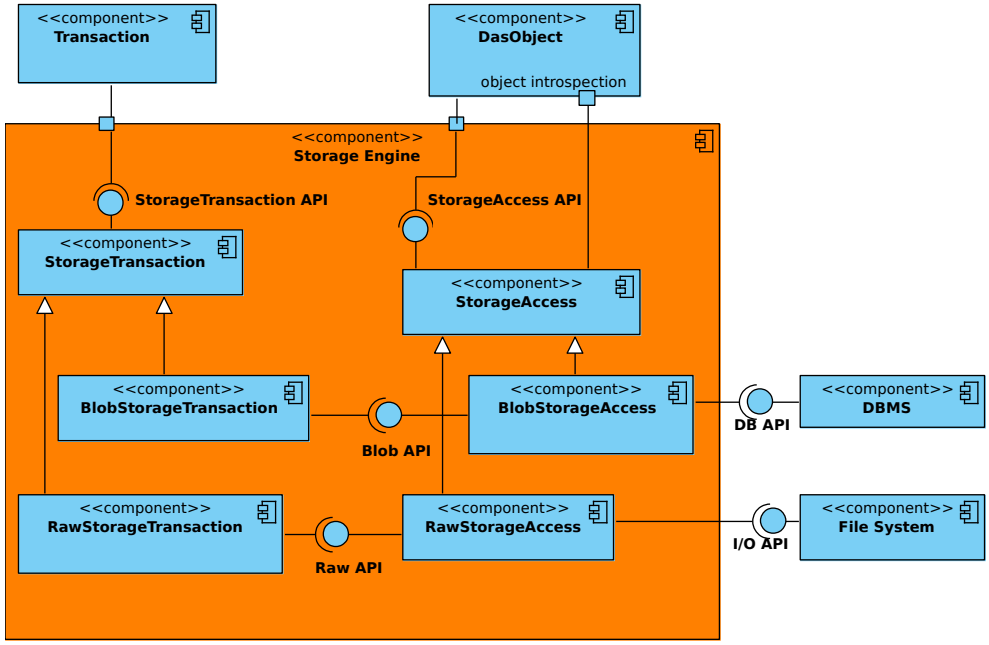


Figure 4.10: Storage Engine components

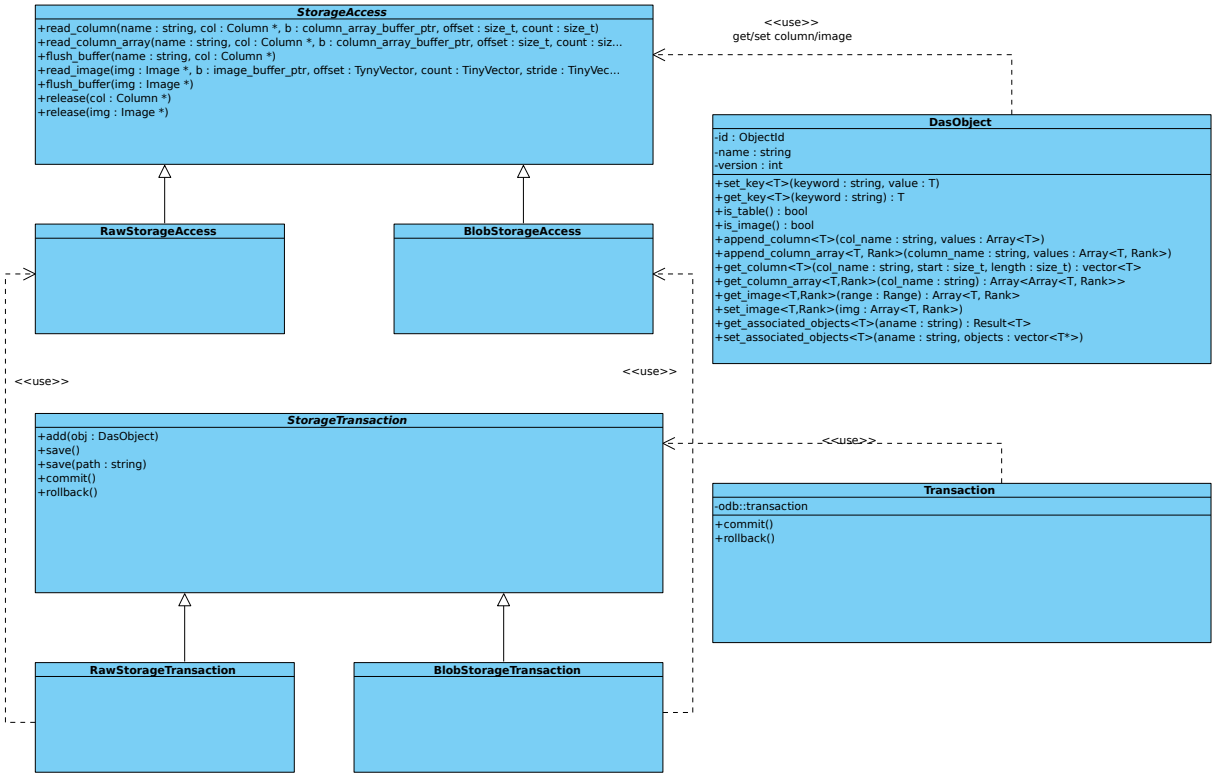
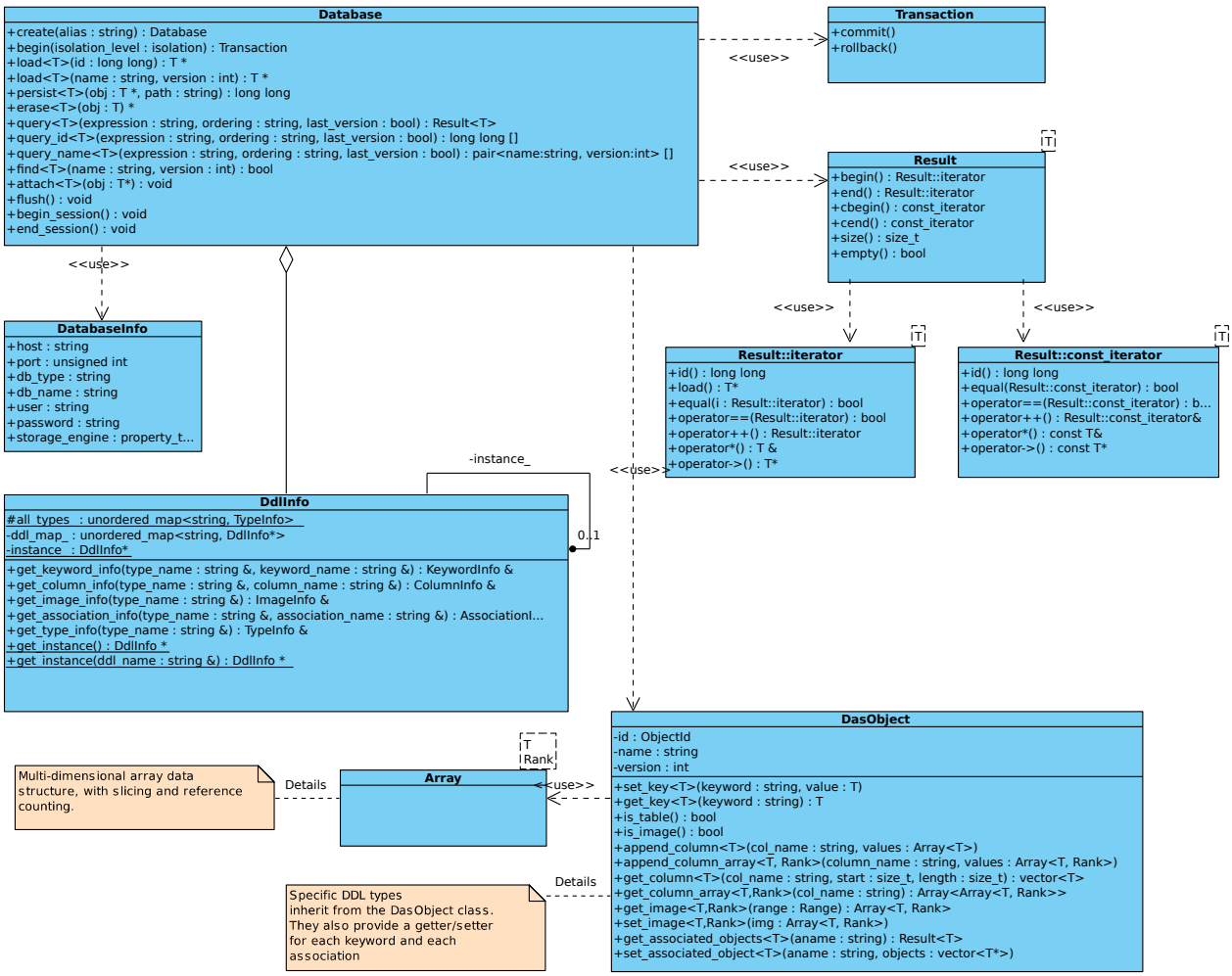


Figure 4.11: Storage Engine class diagram

4.7 System API



5 Appendix A

Data Definition Language XSD

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- DAS Data Definition Layer  -->

<!-- The DAS DDL is a list of one or more data types. -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns="http://oats.inaf.it/das"
           targetNamespace="http://oats.inaf.it/das" elementFormDefault="qualified">
  <xs:element name="ddl">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" ref="type"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <!--
      A data type has zero or one ancestor, an arbitrary number of associated
      data types, a mandatory metadata section (if not inherited from an ancestor)
      and an optional data section.
  -->
  <xs:element name="type">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="associated"/>
        <xs:element minOccurs="0" maxOccurs="1" ref="metadata" />
        <xs:element minOccurs="0" maxOccurs="1" ref="data" />
      </xs:sequence>
      <xs:attribute name="name" use="required" type="xs:ID"/>
      <xs:attribute name="ancestor" type="xs:IDREF"  default="essentialMetadata"/>
      <xs:attribute name="description"/>
    </xs:complexType>
  </xs:element>


  <!--
      A data type can have associated data types. Each association can be a 1 to 1 or
      a 1 to many relation
  -->
  <xs:element name="associated">
    <xs:complexType>
      <xs:attribute name="name" use="required"  type="xs:Name"/>
      <xs:attribute name="type" use="required" type="xs:IDREF"/>
      <xs:attribute name="multiplicity" default="many">
        <xs:simpleType>
          <xs:restriction base="xs:token">
            <xs:enumeration value="one"/>
            <xs:enumeration value="many"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="relation" default="shared">
        <xs:simpleType>
          <xs:restriction base="xs:token">
            <xs:enumeration value="shared"/>
            <xs:enumeration value="exclusive"/>
            <xs:enumeration value="extend"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="description"/>
    </xs:complexType>
```

```
</xs:element>

<!--
    The metadata section defines a list of keywords
-->
<xs:element name="metadata">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="unbounded" ref="keyword"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>


<!--
    A keyword has a name, a type, a unit  and a description.  If a default value
    is set for a meta data, it will not be necessary to set explicitely a value
    for this meta data during runtime. A (database) index will be generated for a
    keyword if "index" is set to "yes", default "no".
-->
<xs:element name="keyword">
  <xs:complexType>
    <xs:attribute name="name" use="required" type="xs:Name"/>
    <xs:attribute name="description"/>
    <xs:attribute name="type" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:token">
          <xs:enumeration value="byte"/>
          <xs:enumeration value="int16"/>
          <xs:enumeration value="int32"/>
          <xs:enumeration value="int64"/>
          <xs:enumeration value="float32"/>
          <xs:enumeration value="float64"/>
          <xs:enumeration value="boolean"/>
          <xs:enumeration value="char"/>
          <xs:enumeration value="string"/>
          <xs:enumeration value="text"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="unit" default="none"/>
    <xs:attribute name="default"/>
    <xs:attribute name="index" default="no">
      <xs:simpleType>
        <xs:restriction base="xs:token">
          <xs:enumeration value="no"/>
          <xs:enumeration value="yes"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:element>


<!--
    The data section defines either a binary table or an image. Both the image or
    the binary table can be stored as external files or as blobs within the database.
    The default is storing the data as external files.
-->
<xs:element name="data">
  <xs:complexType>
    <xs:choice minOccurs="1" maxOccurs="1">
      <xs:element ref="binaryTable" />
      <xs:element ref="image" />
    </xs:choice>
    <xs:attribute name="storeAs" default="file">
```

```
      <xs:simpleType>
        <xs:restriction base="xs:token">
          <xs:enumeration value="blob"/>
          <xs:enumeration value="file"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:element>

<!--
    The binary table defines a list of columns
-->
<xs:element name="binaryTable">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="unbounded" ref="column" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<!--
    A column has a name, a type, a unit  and a description. If the type of the column is
    string, then the maximum length of the string should be specified. The default value
    is 256.
-->
<xs:element name="column">
  <xs:complexType>
    <xs:attribute name="name" use="required" type="xs:token"/>
    <xs:attribute name="type" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:token">
          <xs:enumeration value="byte"/>
          <xs:enumeration value="int16"/>
          <xs:enumeration value="int32"/>
          <xs:enumeration value="int64"/>
          <xs:enumeration value="float32"/>
          <xs:enumeration value="float64"/>
          <xs:enumeration value="boolean"/>
          <xs:enumeration value="char"/>
          <xs:enumeration value="uint8"/>
          <xs:enumeration value="uint16"/>
          <xs:enumeration value="uint32"/>
          <xs:enumeration value="string"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="unit" default="none"/>
    <xs:attribute name="maxStringLength" type="xs:short" default="256"/>
    <xs:attribute name="description"/>
    <xs:attribute name="arraysize" default="1">
      <xs:simpleType>
        <xs:restriction base="xs:token">
          <xs:pattern  value="([0-9]+x)*[0-9]*[*]?"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:element>

<!--
    The image provides a two dimensional array of pixels (data cubes are currently
    not considered). A pixel type must be specified.
-->
<xs:element name="image">
  <xs:complexType>
```

```
      <xs:attribute name="dimensions" default="2">
        <xs:simpleType>
          <xs:restriction base="xs:unsignedByte">
           <xs:minInclusive value="2" />
           <xs:maxInclusive value="11" />
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>

      <xs:attribute name="pixType" use="required">
        <xs:simpleType>
          <xs:restriction base="xs:token">
            <xs:enumeration value="byte"/>
            <xs:enumeration value="int16"/>
            <xs:enumeration value="int32"/>
            <xs:enumeration value="float32"/>
            <xs:enumeration value="float64"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>

      <xs:attribute name="description"/>

    </xs:complexType>
  </xs:element>


</xs:schema>
```

END OF DOCUMENT