University of Freiburg

API definitions for the FMF libs

Rolf Würdemann, Julia Meier

September 18, 2015

Introduction

Riede *et al.* invented the “Full-Metadata Format” (FMF) in [1]. The FMF is a self documenting, human readable data storage and interchange format for the so called “small science”. One of the outstanding features of the FMF is the definition of a structure which glues together data and metadata within one file while preserving the freedom of choice which metadata to store and how to arrange it. By this means it is optimized for the storage and interchange of tabular research data. There are implementations of the FMF as part of the data analysis software “Pyphant” and in basic libraries[1]. To provide easy access to the FMF, the idea of developing separate libraries providing full compliance to [1] was born also in 2010. This report gives a textual overview of the objects and their application programming interface (API).

[1] See Appendix

By the philosophy of the FMF, acquired data belongs to one of two types of objects: On one side the actual data, either measured, derived or calculated and on the other side the metadata, describing the data and circumstances of their acquisition and providing information for reference. This view on data is represented in the definition of the FMF. It contains sections holding the metadata, the corresponding object is further called **FMFMetaSection**. Afterwards the (usually) numerical data is stored in tables, the corresponding object is further called **FMFTable**. A third object which doesn't correspond directly to the data is the **FMFHeader** which holds information for data interchange and file representation such as encoding, field separator and comment character. The main object **FMF** can hold several **FMFMetaSection** and **FMFTable** objects, as well as it holds information on the FMF version for which it is generated and the compliance level as described below. As the compliance level is an inherent trait of the libraries internal handling of objects, that does not effect writing, it will not be written to a file.

Compliance levels

The FMF is a very powerful format for keeping data and metadata together. However, not every situation needs full functionality, especially consistency checks, and thus strict compliance to the definitions of data structures in [1]. For an easier implementation, which also reflects the available amount of CPU power and memory, we have defined the following three compliance levels (CL):

- **CL 1:** Metadata and data definitions are stored as key-value pairs. Keys and values are stored as strings. Data are stored in their respective types and might be written to file using a specified formatter. There are no special fields for quantity, unit or uncernity. The only constraints checked are:

  - Does the number of rows and columns match with the number stored in the *no_columns* and *no_rows* attributes of the corresponding **FMFTable** object
  - The uniqueness of names of the **FMFMetaSection** objects belonging to an **FMF** object and the uniqueness of keys within individual **FMFMetaSection** objects
  - The existence of names and symbols of **FMFTable** objects as well as their uniqueness, if they are mandatory, and the uniqueness of keys in the data definitions within one **FMFTable** object

  CL 1 is mainly meant for writing.

- **CL 2:** In opposite to CL 1, values in **FMFMetaSections** are stored in their respective types. Dependencies, units, quantities and uncertainties are stored in separate attributes, but still stored as strings. In addition to the constraints checked by CL 1, the following constraints are checked:

  - Matching of the dependencies in the definitions of **FMFTables**,
  - Compliance of the units with the unit definitions according to the specified FMF version

  CL 2 is meant to give applications the possibility for easy processing.

- **CL 3:** In addition to CL 2, quantities and units are stored in their respective types and in separate attributes which enable easy processing. In addition to CL 2, the following constraints are checked:

- Matching of the units (if possible)
- Matching of the quantities (if possible)

CL 3 ensures full compliance to the definition of FMF in [1].

Application Programming Interface (API)

In the following the application programming interface (API) including attributes and public methods is defined. The first object defined is the **FMF**, which glues toghether the different sections and tables. Afterwards the API for the central objects is described in the order of **FMFMetaSection** and **FMFTable**.

Note that any FMF object is an ordered structure. **FMFMetasections** and **FMFTables** are added to an **FMF** in order of their initialization. Analogously global comments can be added before and after an **FMFMetasection** or **FMFTable**. Key-value pairs and comments appear in an **FMFMetasection** in the order they were added. This idea holds true also for **FMFTables**. Although, due to the more complex structure, it is realized differently: For example, a comment added after a column appears in the data definitions after the definition of the column and not in the data part of the table. However, it still appears after the object after which it was added.

On errors, all methods should raise an exception. If this is impossible, a value corresponding to NULL in C should be returned for instantiators or a zero-element/false otherwise. The appropriate error-variables should be set with the error-code and message defined by the list of errors in the appendix.

FMF

The **FMF** is the main object that corresponds to the entire FMF file. It is composed of **FMFMetaSection** objects holding the meta-data and **FMFTable** objects holding the data, the **FMFHeader** and several additional attributes.

Public Attributes:

- `FMFHeader` **header**: The basic parameters for the file representation of FMF object
- `FMFMetaSection[]` **meta_sections**: A list containing the meta-data sections of the FMF object
- `FMFTable[]` **tables**: A list containing the tables with the actual data
- `FMFComment[]` **global_comments**: A list of global comments
- `int` **compliance_level**: Compliance level of FMF object
- `int` **max_compliance_level**: Maximum compliance level supported by the library
- `float` **version**: Version of FMF for which the current object was created
- `float` **max_version**: Maximum version supported by the library

Public methods:

- `FMF` **initialize**(): Initializes the object. Returns the object itself on successfull initialization. **initialize** can be called either without any argument or with arguments *title*, *creator*, *place*, *created* and *contact*. In the first case an empty **FMF** object is returned, in the latter a minimal valid **FMF** object with reference section defined by the input variables is returned. Because it is not defined in [1], *contact* is optional in the latter case. This method might be called also by the language specific object initialization routines.

- `FMFMetaSection` **set_reference**(*title, creator, place, created, contact*): Add the mandatory "*reference" section to an **FMF**. *title*, *creator*, *place* and *created* are mandatory, *contact* is optional. If the section "*reference" already exists its values will be changed according to the submitted parameters.

- `FMFTable` **get_table**(*symbol*): Get the **FMFTable** object which matches *symbol*. If no symbol is submitted, get the next **FMFTable** iteratively. An error is raised if there is no **FMFTable** with this symbol or no further **FMFTable**. Mixing calls by symbol and iterative calls also raises an error.

- `FMFTable` **add_table**(*name, symbol*): Add a single **FMFTable** object to the **FMF** object. The **FMFTable** object is returned and should be filled. *name* and *symbol* are optional if there is only one table, but their existence is verified otherwise. If a table with an already existing name and symbol is added, an error is raised. **FMFTables** are added in order of calls.

- `FMFMetaSection` **get_meta_section**(*name*): Get the **FMFMetaSection** object which matches *name*. If no name is submitted, get the next **FMFMetaSection** iteratively. An error is raised if there is no **FMFMetaSection** with this name or no further **FMFMetaSection**. Mixing calls by symbol and iterative calls also raises an error.

- `FMFMetaSection` **add_meta_section**(*name*): Add a single **FMFMetaSection** object to the the **FMF** object. The **FMFMetaSection** object is returned and should be filled. The *name* is mandatory and must not contain '*' as first character. If the **FMFMetaSection** can not be created an error is raised. Section names have to be unique. The mandatory "*reference" section can not be created by this method. See **set_reference**() for creating and altering the "*reference" section. **FMFMetasections** are added in order of calls.

- `bool` **set_header**(*encoding, comment_char, seperator, misc_params*): Set the **FMFHeader** object holding the parameters for file representation. *encoding*, *comment_char* and *seperator* will be set to default values if not given. The parameter *misc_params* is optional. The latter expects a list of arbitrary key-value-pairs with names and parameters when called. Default values are ";" for comment, tab ("\t") as separator and system-encoding. If the latter is not available, "utf-8" or, if simple encoding is necessary, "us-ascii" should be used.

- `FMFHeader` **get_header**(): get the **FMFHeader** object holding the parameters for file representation.

- `FMFComment` **add_comment**(*comment_string*): add a global comment with the text in the *comment_string* to the **FMF** object. A global comment is written to the FMF directly after the header line if no object is created via **add_meta_section**() or **add_table**(), or directly after the last key-value-pair of a section if it was added after this section. If a global comment is added after adding a table, it is placed after the line according to the table in the table definitions. If there is only one table without name and symbol and thus no table definitions, the global comment is placed after the last line of the tables data section.

- `bool` **verify**(): verify if the **FMF** is a valid **FMF** object according to CL and version set in **FMFHeader**. CL has precedence over version.

- `bool` **write**(*filepointer*): Write to filepointer. Return true on success and false else.

- `bool` **read**(*filepointer*): Read FMF from filepointer into **FMF** object. Return true on success and false else.

FMFMetaSection

This object holds the name and entries for the metadata sections. The reference section is of type `FMFMetaSection` with name “*reference” and mandatory entries as specified in the section on the **FMF** object. The reference section usually is created on initialization of an **FMF** object. As stated above we also like to add the *contact* to the “*reference” section to give people an easy way to contact the creator[2].

Public Attributes:

- `string` **name**: The name of the section. The name of the reference section is “*reference”. The use of “*” is restricted to mandatory keywords. Thus names of other sections must not begin with “*”.

- `FMFMetaSectionEntry[]` **entries**: A list containing the metadata entries for this section.

[2] preferable not her/his institute address as these might get canceled

- `FMFComment[]` **comments**: A list containing the comments in this section

Public methods:

- `FMFMetaSection` **initialize**(*name*): Generates new **FMFMetaSection** object with name *name*. *name* is mandatory and must be unique. The use of "*" as first character of *name* is restricted to mandatory keywords. Its use is restricted to indirect methods, e.g. direct creation of objects with names starting with "*" is forbidden. This method might be called also by the language specific object initialization routines.

- `FMFMetaSectionEntry` **get_entry**(*key*): Get the entry referenced by *key*, or the next one if no *key* is given. If the given key does not exists or there is no further entry, an error is raised. Mixing calls by key and iterative calls also raises an error.

- `bool` **add_entry**(*key, value*): Generates an **FMFMetaSectionEntry** with key *key* and value *value*. The supported type of value is determined by the CL. Returns true on success, false else. If the key *key* already exists or *value* is of the wrong type, an error is raised.

- `FMFComment` **add_comment**(*comment_string*): Add a comment with the text in the *comment_string* to the **FMFMetaSection**. The comment will be written directly after the section header if no entry was added, and directly after the entry added before otherwise. It is not recommended to add comments as a last entry to the section. This may cause ambiguities on reading, for such a comment can be mistaken for a global comment added after the section.

- `bool` **verify**(*CL, version*): Verify if **FMFMetaSectionEntry** is correct according to given CL and version. CL has precedence over version.

FMFTable

This object is a container for the actual tables and their data. As long as there is only **one** table, name and symbol can be omitted, otherwise they are stored in an **FMFTableDefinition** object.

Public Attributes:

- `string` **name**: Name of the **FMFTable** object. Can be empty if there is only one table with the according **FMF** object.
- `FMFSymbol` **symbol**: Symbol of the **FMFTable** object. Can be empty if there is only one table with the according **FMF** object.
- `FMFDataDefinition`[] **data_definitions**: A list containing one data definition per data column.
- `int` **no_columns**: Number of columns of data (this corresponds to the number of `FMFDataDefinition` objects in **data_definitions**)
- `int` **no_rows**: Number of rows of data.
- `FMFUnion[][]` **data**: Two dimensional array of data, odered column, row.
- `FMFComment[]` **comments**: A list of comments in **FMFTable** object.

Public methods:

- `FMFTable` **initialize**(*name, symbol*): Generates new **FMFTable** object with name *name* and symbol *symbol*. *name* and *symbol* are mandatory if there is more than one table. If submitted, they must be unique. This method might be called also by the language specific object initialization routines.

- `FMFDataDefinition` **add_column**(*name, key, formatter, dependency, unit, uncernity*): Add the corresponding column to the **FMFTable** object. Only *name*, *key* and *formatter* are mandatory. For CLs higher than CL1, *dependency* and *unit* must be given if they exist. Formatter are usually specified using printf convention. Due to the fact that other conventions are possible, the used ones should be documented.

- `bool` **add_data_column**(*column_of_data*): Add a column of data to the table. The first added column determines the length of the individual columns and thus the value of the attribute **no_rows**. Return true on success, false on error.

- `bool` **add_data_row**(*row_of_data*): Add a row of data to the table. The number of elements within the row has to match the number of columns defined by **add_column**.

- `FMFUnion[]` **get_data_column_by_symbol**(*symbol*): Get the values from data column *symbol*. Should raise an exception if no column with symbol *symbol* exists.

- `FMFComment` **add_comment**(*comment_string*): Add a comment with text in *comment_string* to **FMFTable** object. If a comment is added before any column is defined, the comment is placed at the top of the **data** section of the table. Comments added after a column is defined go after the according row in the data definitions section. Comments added between two rows of data, are placed between these two rows.

- `bool` **verify**(*CL, version*): Verify that data definitions and data do match the requirements according to *CL* and *version* given. CL has precedence over version.

APPENDIX I: List of Exceptions and error codes

This section will be split into three parts. At first a list is given which introduces the used exceptions, warnings and error codes. The second part describes the suggested use of the exceptions at CL 1 per method for each class. The third part will contain the suggested use of exceptions on additional issues for CL 2 and CL 3. It is not yet included in this document.

Exceptions and error codes

All errors to be raised by the methods can be boiled down to one of the following exceptions.

Exception	Error-Number	Description
MissingSubmission	0x01	At least one mandatory keyword or parameter not submitted.
MultipleKey	0x02	Submitted key does already exists.
ForbiddenSubmission	0x04	Submitted keyword or parameter contains forbidden character(s).
TableConsistencyViolation	0x08	An **FMFTable** object must not have columns of different length nor inconsistencies between the attributes **no_rows**, **no_columns** and **data**.
UndefinedObject	0x10	Object could not be retrieved.
AmbigousObject	0x20	Object not properly specified.
SpecificationViolation	0x40	Object does not comply with compliance level or version specifications.
IOError	0x80	Input/Output Error.

Warnings

Warnings are not mandatory to be implemented, but they may be helpful in preventing evitable ambiguities.

Warning	Number	Description
AmbigousComment	0x100	A comment added here may cause ambiguities on reading.

Exceptions and according error messages per method at CL 1

In the following we describe the exceptions and error messages for each method within each class. Tables 1 to 5 hold the exceptions and error messages for the class **FMF**. Tables 6 and 7 hold the according information for class **FMFMetaSection** and tables 8 to 11 for class **FMFTable**.

Table 1: Suggested exceptions per method for the class **FMF**, methods **initialize** to **get_table** in order as given in subsection FMF of section Application Programming Interface (API)

Method	Exception	Message	Description
initialize	**UndefinedObject**	AllocationError	General error related to memory allocation
	MissingSubmission	MissingArgument	Only some of the arguments are submitted
set_reference	**MissingSubmission**	MissingArgument	Only some of the mandatory arguments are submitted
get_table	**UndefinedObject**	TableNotFound	The specified **FMFTable** does not exists
	UndefinedObject	NoFurtherTable	No further **FMFTable** exists
	AmbigousObject	MixedCalls	A mix between querying table by **FMFTable** and iterative calls occurred

Table 2: Suggested exceptions per method for the class **FMF**, methods **add_table** to **get_meta_section** in order as given in subsection FMF of section Application Programming Interface (API)

Method	Exception	Message	Description
add_table	**UndefinedObject**	AllocationError	General error related to memory allocation
	MissingSubmission	MissingTableName	No **FMFTable** name was submitted
	MissingSubmission	MissingTableSymbol	No **FMFTable** symbol was submitted
	MultipleKey	TableNameExists	The submitted **FMFTable** name already exists
	MultipleKey	TableSymbolExists	The submitted **FMFTable** symbol already exists
get_meta_section	**UndefinedObject**	SectionNotFound	The specified **FMFMetaSection** does not exists
	UndefinedObject	NoFurtherSection	No further **FMFMetaSection** exists
	AmbigousObject	MixedCalls	A mix between querying **FMFMetaSection** by name and iterative calls occurred

Table 3: Suggested exceptions per method for the class **FMF**, methods **add_meta_section** to **add_comment** in order as given in subsection FMF of section Application Programming Interface (API)

Method	Exception	Message	Description
add_meta_section	**UndefinedObject**	AllocationError	General error related to memory allocation
	MissingSubmission	MissingName	No **FMFMetaSection** name was submitted
	MultipleKey	SectionNameExists	The submitted **FMFMetaSection** name already exists
	ForbiddenSubmission	ForbiddenName	The submitted **FMFMetaSection** name does contain forbidden character(s)n
set_header	**ForbiddenSubmission**	WrongEncoding	The supplied encoding does not follow the encodingscheme used by emacs
get_header	None	None	None
add_comment	**UndefinedObject**	AllocationError	General error related to memory allocation

Table 4: Suggested exceptions for method **verify** of the class **FMF** as given in subsection FMF of section Application Programming Interface (API), part one

Method	Exception	Message	Description
verify	**SpecificationViolation**	InvalidFMF	The **FMF** is invalid – unspecific error, use if nothing else fits
	SpecificationViolation	InvalidVersion	The **FMF** violates the definition for the specified version
	MissingSubmission	MissingTableName	At least there is one **FMFTable** object without name in a multi **FMFTable** **FMF** object
	MissingSubmission	MissingTableSymbol	At least there is one **FMFTable** object without name in a multi **FMFTable** **FMF** object
	MultipleKey	NonUniqueTableName	There are at least two **FMFTable** objects with the same name

Table 5: Suggested exceptions for method **verify** of the class **FMF** as given in subsection FMF of section Application Programming Interface (API), part two

Method	Exception	Message	Description
verify	**MultipleKey**	NonUniqueTableSymbol	There are at least two **FMFTable** objects with the same symbol
	MultipleKey	NonUniqueMetaSectionName	There are at least two **FMFMetaSection** objects with the same name
	SpecificationViolation	InvalidFMFTable	One of the **FMFTable** objects is invalid
	SpecificationViolation	InvalidFMFMetaSection	One of the **FMFMetaSection** objects is invalid

Table 6: Suggested exceptions per method for the class **FMFMetaSection**, methods **intialize** and **get_entry** as given in subsection FMFMetaSection of section Application Programming Interface (API)

Method	Exception	Message	Description
initialize	**UndefinedObject**	AllocationError	General error related to memory allocation
	MissingSubmission	MissingName	No **FMFMetaSection** name was submitted
	MultipleKey	SectionNameExists	The submitted **FMFMetaSection** name already exists
	ForbiddenSubmission	ForbiddenName	The submitted **FMFMetaSection** name does contain forbidden character(s)
get_entry	**UndefindedObject**	EntryNotFound	The specified **FMFMetaSection** entry does not exist
	UndefindedObject	NoFurtherEntry	No further **FMFMetaSection** entry exists
	AmbigousObject	MixedCalls	A mix between querying **FMFMetaSection** entries by name and iterative calls occurred

Table 7: Suggested exceptions per method for the class **FMFMetaSection**, methods **add_entry** to **verify** in order as given in subsection FMFMetaSection of section Application Programming Interface (API)

Method	Exception	Message	Description
add_entry	**UndefinedObject**	AllocationError	General error related to memory allocation
	MissingSubmission	MissingKey	No key submitted for **FMFMetaSection** entry
	MultipleKey	EntryKeyExists	The submitted **FMFMetaSection** entry key already exists
	ForbiddenSubmission	BadValueType	The Type of the submitted **FMFMetaSection** entry value is not supported at this Compliance Level.
add_comment	**UndefinedObject**	AllocationError	General error related to memory allocation
verify	**SpecificationViolation**	InvalidFMFMetaSection	The **FMFMetaSection** object is invalid – unspecific error, use if nothing else fits
	SpecificationViolation	InvalidVersion	The **FMFMetasection** violated the definition for the specified version
	MultipleKey	NonUniqueEntryKey	There are at least two **FMFMetaSection** entries with the same key

Table 8: Suggested exceptions per method for the class **FMFTable**, methods **initilize** and **add_column** as given in subsection FMFTable of section Application Programming Interface (API)

Method	Exception	Message	Description
initialize	**UndefinedObject**	AllocationError	General error related to memory allocation
	MissingSubmission	MissingTableName	No **FMFTable** name was submitted
	MissingSubmission	MissingTableSymbol	No **FMFTable** symbol was submitted
	MulitpleKey	TableNameExists	The submitted **FMFTable** name already exists
	MultipleKey	TableSymbolExists	The submitted **FMFTable** symbol already exists
add_column	**UndefinedObject**	AllocationError	General error related to memory allocation
	MissingSubmission	MissingColumnName	No column name was submitted
	MissingSubmission	MissingColumnSymbol	No column symbol was submitted
	MissingSubmission	MissingColumnFormatter	No column formatter was submitted
	MultipleKey	ColumnNameExists	The submitted column name already exists
	MultipleKey	ColumnKeyExists	The submitted column key already exists
	ForbiddenSubmission	InvalidFormatter	The submitted column formatter was not recognized

Table 9: Suggested exceptions per method for the class **FMFTable**, methods **add_data_column** to **add_data_row** in order as given in subsection FMFTable of section Application Programming Interface (API)

Method	Exception	Message	Description
add_data_column	**UndefinedObject**	AllocationError	General error related to memory allocation
	ForbiddenSubmission	InvalidNumberOfRows	The submitted column does contain a number of rows different from **no_rows**
	SpecificationViolation	InvalidDataColumn	The submitted data column does not match given formatter
add_data_row	**UndefinedObject**	AllocationError	General error related to memory allocation
	ForbiddenSubmission	InvalidNumberOfColumns	The submitted row does contain a number of columns different from **no_columns**
	SpecificationViolation	InvalidDataRow	The submitted data row does not match given formatter

Table 10: Suggested exceptions per method for the class **FMFTable**, methods **get_data_column_by_symbol** and **add_comment** in order as given in subsection FMFTable of section Application Programming Interface (API)

Method	Exception	Message	Description
get_data_column_by_symbol	**MissingSubmission**	MissingSymbol	No column symbol submitted
	Undefined Object	InvalidSymbol	Submitted column symbol does not exist
add_comment	**UndefinedObject**	AllocationError	General error related to memory allocation

Table 11: Suggested exceptions per method for the class **FMFTable**, method **verify** in order as given in subsection FMFTable of section Application Programming Interface (API)

Method	Exception	Message	Description
verify	**SpecificationViolation**	InvalidFMFTable	The **FMFTable** object is invalid – unspecific error, use if nothing else fits
	AmbigousObject	InconsistentNumberOfColumns	Number of columns in data does not match **no_columns**
	AmbigousObject	InconsistentNumberOfRows	Number of rows in data does not match **no_rows**
	AmbigousObject	InconsistentNumberOfColumns	Inconsistent number of columns in data
	AmbigousObject	InconsistentNumberOfRows	Inconsistent number of rows in data
	SpecificationViolation	InvalidVersion	The **FMFTable** violated the definition for the specified version

APPENDIX II: FMF data structures in Pyphant

An implementation of the FMF is included in the data analysis software Pyphant. For it is only used for writing, all data fields are converted to unicode strings. Thus, this implementation resembles what we define as CL 1 in section Compliance levels, but without any verification. For the class structure see Figure 1. Pyphant can also read data from FMF files into “Field Containers”, a data structure within Pyphant. Therefore no special FMF datastructure is needed for this task.

The functionality of Pyphant roots in data structures that are derived from the base class “DataContainer”. This includes the above mentioned “FieldContainer” as well as an object called “SampleContainer” that will not be further discussed here. These data structures embody a philosophy similar to the FMF: To keep data, units and errors together and enable simultaneous processing as well as keeping metadata in the same data structure with data.

Thus, a “Field Container” (FC) holds data similar to a typical FMF of a measurement: The object called “data” in an FC holds one dimensional data that can depend on an arbitrary number of “axes”. The FC aditionally may hold the physical unit of the data and an error as well as a dictionary of parameters. The axes themselves are one dimensional FCs. Every FC has a longname and a shortname. As the axes are FC, this holds also true for them. For information about the “DataContainer” and “FieldContainer” classes including attributes, see Figure 2.

During the generation of an FMF instance from a FC in Pyphant, the axes, the data and the error are interpreted as columns of

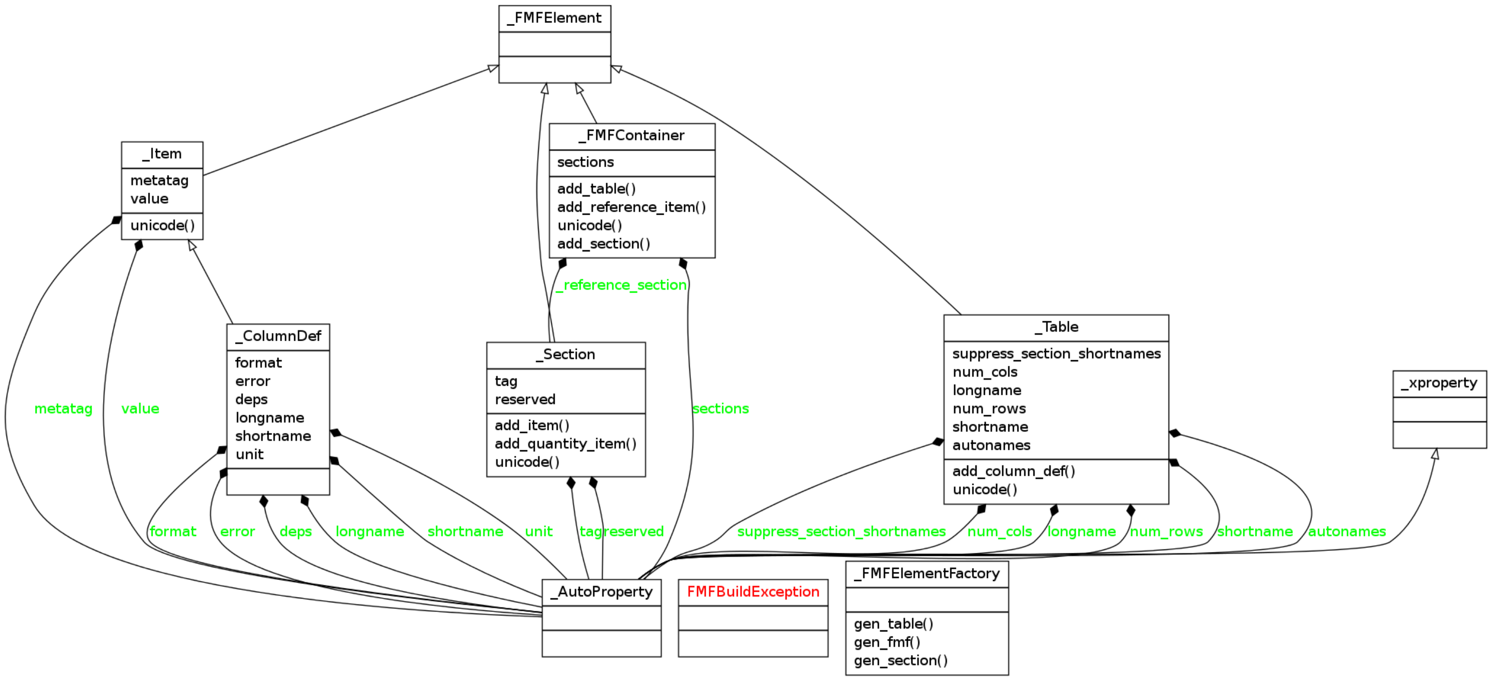

Figure 1: The class diagram of the FMF implementation that is part of the Pyphant data analysis software. Class diagram generated by “pyrevert”.

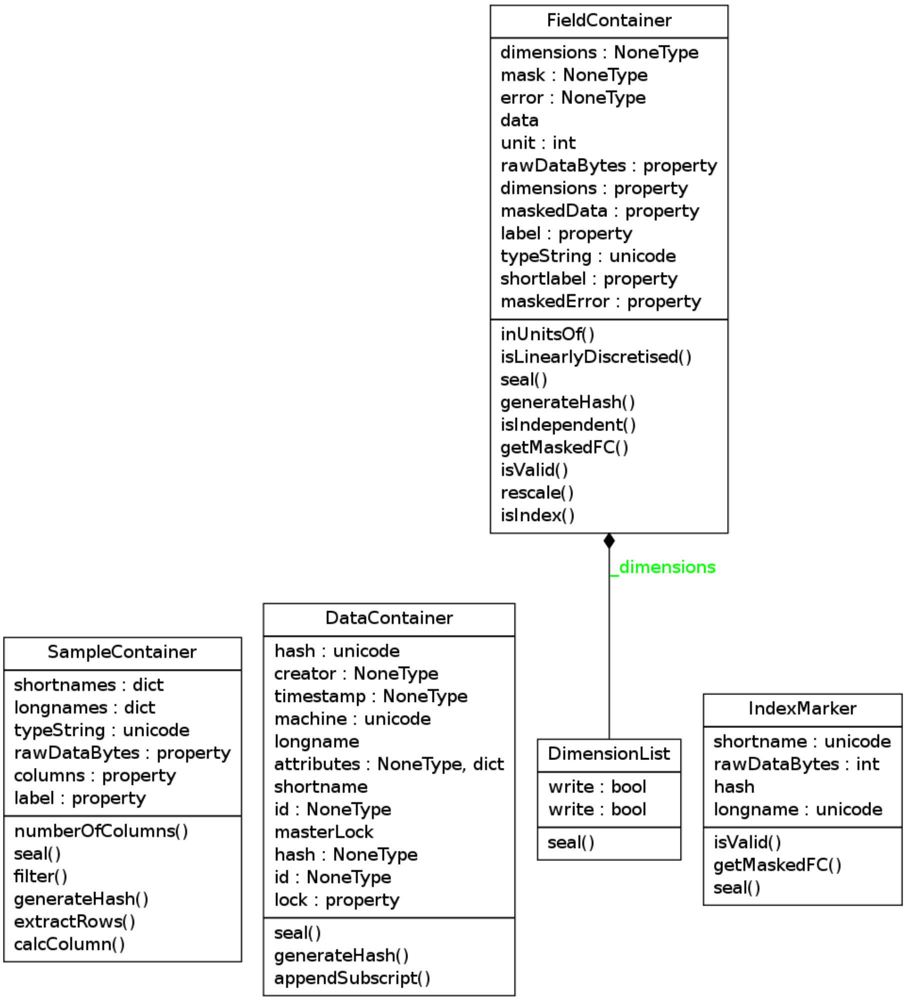

Figure 2: The “FieldContainer” class in Pyphant. The “dimensions” attribute holds the axes, the “error” holds the error data, “data” holds the data and “unit” the unit of data and error. “longname” and “shortname” are attributes of the “DataContainer” class from which “FieldContainer” inherits. Note that the “DataContainer” class has a “creator”, “timestamp” and “machine” attribute, too. These attributes hold information which can be used to generate a “*reference” section. Further parameters are held in the “attributes” attribute of the “DataContainer” class.

a table. The longnames of the axes are used as names of the according columns in the data definitions. The shortnames of the axes constitute the according symbols, together with units if there are any. The name of the column holding the one dimensional “data” from the FC in the data definitions is the longname of the FC. The according symbol is composed of the FCs shortname, the known dependencies on the axes, the unit and the symbol of the error column if given. Name and symbol of the column holding the error are also derived from the original FCs long- and shortname.

APPENDIX III: Rudimentary implementations in python: readfmf and simplefmf

There are already two rudimentary implementations of FMF as stand-alone libraries in phyton: readfmf and simplefmf. While readfmf is a implementation for reading FMF files, simplefmf is an implementation for writing. Both are available on GitHub.

readfmf

readfmf is derived from pythons “ConfigObj”. Calling the function **stream2data**(*filepointer*) returns two objects, representing the FMF data from the stream indicated by *filepointer*. The first of the two returned objects is of type “ConfigObj” and contains a dict holding the data from the meta sections. The second returned object is of type “ndarray” and contains the data of the tables, indexed by the table symbol and ordered in column, row as string values. There are **no** CL or version checks.

simplefmf

simplefmf comprises three individual classes, “FMFDataDefinition”, “FMFTable” and “SimpleFMF”, the former two aggregated to the latter that represents the FMF. “FMFDataDefinition” handles the data definitions for the “FMFTable” objects. The meta sections are stored and handled within the “SimpleFMF” object. The entries of the meta sections, as well as the table definitions are stored as key-value pairs of strings. Table data are stored in their respective types and a formatter can be choosen for output. Thus “simplefmf” resembles CL 1 without reflecting the API defined above.

References

[1] Moritz Riede, Rico Schueppel, Kristian O. Sylvester-Hvid, Michael C. Röttger Martin Kühne, Klaus Zimmermann, and Andreas W. Liehr. On the communication of scientific data: The full-metadata format. **Computer Physics Communications**, 181:651 – 662, 2010.