Software Programming Standards

	Name	Date	Signature
Prepared by	K. Kosack (CTAO, CEA Saclay)		KOSACK Karl Digitally signed by KOSACK Karl Date: 2020.01.23 13:48:26 +01'00'
Approved by	S. Schlenstedt (CTAO)		Stefan Schlenstedt Digitally signed by Stefan Schlenstedt Date: 2020.01.23 13:54:17 +01'00'
Released by	W. Wild (CTAO		

This Version:					
Issue	Rev.	Created	Comment	Ownership	Distribution
1	a	2020-01-23	Release version. Replaces document *SYS-STAND/161012*.	CTAO Computing	CTA

Change Log:				
Issue	Rev.	Created	Reason / Remarks / Initiation	Part Affected
draft	0.3	2018-02-01	Alignment with the system simplification planning and minor updates.	all
draft	0.2	2017-03-29	Incorporated first round of comments from PC. Incorporated additional comments received from A. Okumura, D. Hoffmann, J. Schwarz, M. Punch, I. Sadeh, D. Melkumyan, T. Le Flour, E. Lyard.	all
draft	0.1	2016-09-06	Preliminary draft originally incorporated into the Software Development Plan & Standards. Released as *SYS-STAND/161012*.	all

List of Contributors:		
Contributor	Affiliation	Contribution Subject / Chapter
Francesco Dazzi	CTAO	All chapters
Matthias Füßling	CTAO	All chapters
Igor Oya	CTAO	All chapters
Gino Tosti	CTAO, Univ. di Perugia	All chapters

Table of Contents

1 Introduction

The Cherenkov Telescope Array (CTA) construction project will encompass a wide range of software sub-projects covering, but not limited to, integrated control systems, common array-control software, data processing software, Graphical User Interface (GUI) software, science tools and computing support. There will be also several operating environments and frameworks such as embedded systems, ALMA Common Software (ACS) components, grid / batch jobs, web front-ends, and end-user laptop executables. It is challenging to choose standards that work in all cases, therefore this document provides the minimal programming standards for software developed for the CTA construction project.

1.1 Purpose

The purpose of this document is to guide the software developers in the production of products (hereinafter referred to as CTA software) that can be easily understood, operated and maintained by the Cherenkov Telescope Array Observatory (CTAO) personnel over a long period of time. The standards presented in this document can evolve in time to better suit the needs of both software development teams (suppliers) and CTAO (customer), responsible of the CTA construction project. Therefore, this shall be considered as a living document. Each software development team may produce more stringent standards that are applied internally to the teams. These internal standards may be used to improve this document in a subsequent revision.

1.2 Scope

The software programming standards presented in this document apply solely to new software explicitly written for the CTA construction project. They do not apply to existing frameworks or legacy software for which standards already exist (e.g. ACS, *sim_telarray*, etc). Moreover, this document does not cover the programming standards for embedded software and firmware written for controlling devices like Field Programmable Gate Array (FPGA), Programmable Logic Device (PLD), Programmable Logic Controller (PLC), microcontrollers, etc. These domain-specific programming standards will be treated in dedicated documents.

The applicable standards are indicated in this document with the verb “shall”, whereas the others are guidelines and recommendations that the software development teams are encouraged to consider. The CTA software that is being developed for the Critical Design Review (CDR) shall be compliant with these applicable standards.

This document follows the CTA definitions [1] and is an update of and supersedes the document previously released with document number *SYS-STAND/161012*.

2 Computing Environments

The CTA computing environments, namely the combination of operating systems and computing hardware, can be generally divided into three categories, hereafter described.

Production Environment: the computing environment where the software is executed at a data centre or grid site, including all necessary dependencies.

Development Environment: the computing environment where the software is developed (e.g. developer's local machines), which includes an Integrated Development Environment (IDE), compiler or interpreter, all necessary dependencies, common libraries, and the ability to run the unit test suites.

Integration Environment: the computing environment where the software is continuously integrated: compiled, tested, and packaged for distribution, including the building of all documentation. This is generally an identical system to the production environment, but may include e.g. cross-compilation and packaging for developer machines as well. This includes supported compilers / interpreters, all dependencies, as well as any software and data necessary for running the unit and integration test suites.

The production and integration environments will be based on Linux Red Hat Enterprise (RHEL) re-compiled distributions (e.g. *Scientific Linux* and *CentOS*), and are tightly linked to the software available at the computing centres. These are the minimal systems on which CTA software shall compile and run. The development environment is however more flexible to accommodate the needs and comfort of the people writing software. Developer's machines may run for instance more user-oriented *Linux* systems as well as *macOS* machines. Nevertheless, it shall be verified that all software committed to the repositories compile (for compiled languages), pass all code-quality and functionality tests, and run on the production and integration environments.[1]

The CTA standard computing environments are as follow.

- **Production Environment:** RHEL re-compiled distribution (e.g. *Scientific Linux* and *CentOS*)[2]
- **Development Environment:** any operating system (e.g. *CentOS*, *macOS Catalina*, *Windows 10*) and IDEs supported by the respective provider and common libraries chosen by CTAO (e.g. see section 5.2).
- **Integration Environment:** RHEL re-compiled distribution (e.g. *Scientific Linux* and *CentOS*), with optional additional support for testing and packaging on developer systems (e.g. *macOS* or *Windows 10*).

The software developers contributing to the CTA construction project shall ensure that the delivered software is tested and works on the final production and integration environments chosen by CTAO. Lightweight virtualization (e.g. *Docker Containers*) may be used to achieve some independence from these standard environments, however software shall still build correctly in the final production and integration environments, and it shall also be possible to build and execute the container in that environment.

[1] Further details on these environments are planned for a future version of the document.
[2] The specific version of the production environment will be defined in a future release of this document.

For the integration of the CTA software, the CTAO will implement a central Continuous Integration (CI) system that will be specified in a subsequent version of this document.

3 Languages

The general principle is: *keep code simple, clear, clean*[1]*, and well-documented*. Over the life of CTAO (30 years[2]), many software developers / maintainers will need to read and modify what others have written, and it therefore shall be easy to follow and understand.

The programming languages and minimum versions accepted by CTAO are hereafter listed.

- **C++**: version 11 (*International Organization for Standardization (ISO) / International Electrotechnical Commission (IEC) 14882:2011*) or a later ISO standard.
- **Java**: *Oracle Java SE JDK* version $\geq$ 11. The use of a later major version of *Java* will depend on when it will be adopted by ACS.
- **Python**: version $\geq$ 3.6. Compiled *Cython* extensions are allowed.
- **JavaScript**: *ECMAScript* version $\geq$ 5 is allowed, but only for web-based user-interfaces and visualizations.

Exceptions are authorized (after justification to the CTAO) only in specific cases where the chosen standard language prevents compilation due to the requirements of underlying libraries (e.g. ACS). The addition of other languages is discouraged to minimize the complexity of long-term maintenance, but might be not forbidden if a strong technical justification is made to the CTAO.

The scripting languages (short glue code to connect a series of executables) accepted by CTAO are hereafter listed.

- **Python**: version $\geq$ 3.6 shall be used for any complex scripting.
- **BASH**: version $\geq$ 4.1 is allowed solely for simple cases and its use shall be justified in the code documentation.

In general, piping together several executables and redirecting output is more appropriate in *BASH*, whereas a more complex chain with if-statements, complex loops, machine-independence, sub-routines or command-line parsing is better implemented and maintained with *Python*, which has more complete debugging and testing support.

[1] A recommended book is [2].
[2] and beyond for higher-level data management software.

4 Code

4.1 Code Style

Coding standards help to keep the code readable and maintainable by a large group of developers who may be all working on the same file simultaneously. Common code style rules are particularly important when using revision control systems (e.g. *Git*) since the format of code is used to track changes and merge updates from multiple developers. The style of code will be checked and evaluated by automated systems (e.g. *SonarQube* or *TravisCI*) connected to the CI system, allowing managers to monitor and enforce rules.

The coding standards depend on the languages used and they refer to the accepted languages listed in section 3. The following rules are considered for general code formatting / style:

- **C++**: use *Google* style (`https://google.github.io/styleguide/cppguide.html`) and follow the recommendations in the C++ Core Guidelines.
- **Python**: use *PEP8* style (`https://www.python.org/dev/peps/pep-0008/`).
- **Java**: use *Google* style (`https://google.github.io/styleguide/javaguide.html`).

The previous rules are valid unless superseded by the following CTA specific style rules:

Human Language
All variable names, comments, and documentation shall be in English. The choice of spelling (American vs. British) should be consistent within a Work-package.

Indentation
Four spaces (not tabs) shall be used for indentation in all languages. In *C++* this can be achieved for instance using in clang-format the *IndentWidth*:4 option, and in *Emacs* setting c-basic-offset option.

Line Length
The line length shall by at maximum 90 characters so that the code can be displayed in two to three columns on a normal display (useful when merging). Most code re-formatting tools do this by default (e.g. *black* for python or *clang-format* for C++)

Variable Naming
The variable names shall:

- be obvious and un-abbreviated, using full English words with consistent spelling, with the exception of:
  - integer loop indices, where the recommendation is to use a descriptive name starting with *i, j, k*, such as `i_tel`, `j_pix`;
  - very compact loops (maximum of a few lines of code), where the full body of the loop can be seen without scrolling - smaller indices like `i` or `j` are acceptable, though two-character indices are recommended[1];
  - *lambda*-expression place-holders;
  - list / set / generator comprehension place-holders;
- be longer than a single character, except in the cases listed above;
- be lower-case, or at least start with a lower-case letter;

[1] Variables longer than one character are useful for search / replace operations (replacing all uses of `i` with `i_tel` is not easy, but `ii` is), as well as for visual identification (is the index a 1 or an i?).

- use underscores to separate words or `camelCase` if used consistently within a Work-package, but avoid mixing various conventions;
- have the unit name appended if the variable may have ambiguous units. This is not needed if a common unit-tracking library (e.g. `astropy.units`[2] or `Boost::Units`[3]) is used. In the case a common unit library cannot be used, it is good practice to choose a common set of units and enforce them (e.g. require that all angles within a Work-package are in radians and are never expressed in degrees), and continue append the unit to the variable name to avoid ambiguity.

Good examples are:

- `max_energy_tev`;
- `pixel_rotation_angle_deg`;
- `likelihood`.

Bad examples are:

- `n`;
- `rdval`;
- `prot`;
- `p_rot`.

It is recommended to use an editor with the auto-completion function in order to facilitate the typing of long variable names.

Function / Method Naming

The function names shall:

- start with a verb and be clear as to what they do (avoid abbreviations);
- use a consistent capitalization style[4] depending on the language (e.g. `camelCase` for *Java* and *C++*, `underscore_separated` for *Python*);
- use properties instead of setters / getters for access to simple internal variables if the language supports them (e.g. *Python*);
- use common wording whenever possible, at least within the Work-package (e.g. do not have a function `make_parameters()` and then another `generate_other_parameters()`, keep the wording consistent);
- have a correct and consistent spelling (avoid `get_first_color(); get_2nd_colour();`);
- use named parameters if the language supports them, (e.g. *Python*) when an argument list is long or ambiguous. I.e. prefer calling multi-parameter functions using names to location-based parameters.

Good examples are:

- `readCTAOptions();`
- `read_cta_options();`
- `make_rectangle(center=(10, 10), width=10, length=50))` if using *Python*, which supports named parameters;
- or if C++ use clear types like `make_rectangle(Point(10,10), WidthLength(10, 50);`
- `number_predicted(), predicted_counts()`
- `apply_energy_correction(energy_tev)`.

Bad examples are:

- `readCTAoptions()` (inconsistent capitalization)

[2] `http://docs.astropy.org/en/stable/units/`
[3] `http://www.boost.org/doc/libs/1_61_0/doc/html/boost_units.html`
[4] Note abbreviations like CTA shall be all caps when using `camelCase`, and the following word shall still be capitalized.

- `doIt()` (unclear purpose)
- `make_rectangle(10,10,10,50)` (unclear arguments)
- `npred()` (non-obvious abbreviation)

Classes

Classes shall:

- use `CapWords` style (start with capital letter, no underscores);
- have subclass names that end with the parent class name when possible, to make the relationship clear (e.g. `FileWriter` ⇒ `FITSFileWriter`, `TextFileWriter`);
- use namespaces (*C++*) or packages (*Python*, *Java*) that group common classes and related functions.

4.2 Code Design

The following rules apply for the CTA software code design:

Object Orientation

For code that is object oriented the developer shall:

- identify and follow common object-oriented design patterns (see e.g. `https://en.wikipedia.org/wiki/Software_design_pattern`);
- generate class hierarchies that are consistent and make sense - class B shall only be a subclass of class A if the phrase "*B is an A*" makes grammatical sense, if not, the developer might consider encapsulation ("*A has a B*") instead of inheritance, or use a different design pattern;
- not overuse classes and class-hierarchies, because not everything must be a class and in many cases plain functions outside of classes can be more flexible.

Public API Design

For any code that exposes a public Application Programming Interface (API) intended to be used by other developers, the developer shall:

- separate the functions / classes / methods / variables that are part of the API from functions that are meant to be internal / private;
- shall mark as "deprecated" in the documentation (and code if the language supports it) the API functionality that is to be removed in the next release(s), to signal to developers using it that they need to consider changing their dependent code.

Error Reporting

The error reporting shall have the following characteristics.

- Descriptive exceptions (if the language supports them) shall be used for error handling (but not for flow-control) and shall be caught and handled as soon as possible[5].
- The specific exception types shall be defined for failure modes that extend built-in generic exception classes like `RuntimeError` (`std::runtime_error`), `ValueError` (`range_error`), etc. The generic exception classes shall not be used directly, but sub-classed / extended with descriptive names.
- The exceptions shall include a descriptive message that can help a user to understand what lead to the exception and what value caused it. For instance, if the exception is of type `FileNotFoundError`, the message shall contain the file name with full path that was expected.

[5] Note that this rule does not agree with the *Google* coding guidelines, which recommend avoiding exceptions due to legacy code that does not support them. Because this document pertains to new code only, there is no strong issue against the use of exceptions.

- The exception description shall be written assuming the reader is not an expert in the software. Note that in the case of a problem, exception names and descriptions are what will be seen by the software end-user or operator, who are not experts in the software's internal design. Therefore, the message shall help them to resolve the problem or to provide a useful bug report.
- Status return codes may also be used for cases where the error is not critical, is used for control flow, or is likely to occur at a very high rate. However, these codes shall be well-documented and used consistently (e.g. if `0` means *fail* in one case, it may not mean *succeed* in another). The strong recommendation is that if error status codes are used, the conventions and *enum* definitions specified in the *C* standard library in `errno.h` shall be used (0 = success, positive numbers are standard error codes), and negative error codes may be defined for anything that is not covered (user-defined).
- If calling a library function that throws exceptions, they shall be handled in your module if possible, and re-thrown as a custom exception if necessary.

Console Logging

Any console output from CTA software shall:

- use the same logging packages (or functions) (e.g. the logging module in *Python*) within a Work-package, avoiding bare `print` or `std::cout` statements, ideally with at least 4 levels message intensity: *debug*, *info*, *warning*; *error* / *alert* so that the user can determine the severity or usefulness of the output;
- format log messages in a common (and ideally globally modifiable) way - for example when executing a piece of CTA software code, one user / system may want the level + program name + timestamp + log message, while another needs the IP of the execution machine or the line number and file-name of the code block that emitted the message in addition, nevertheless with a small amount of foresight and a flexible logging system[6], it is possible to accommodate these cases without having to change each line of code where text is written to the console;
- have the ability to change the log verbosity level (e.g. to filter out debug output from important warnings);
- use a high-level logging system if it is provided in the underlying software library (e.g. ACS).

For small or low-level executables, it may be possible to archive common logging by using an external wrapper that formats messages in a common format rather than implementing the logging internally.

Configuration

For any CTA software configuration process the developer shall:

- not store in the code parameters that may need updating over the life of CTAO, but rather in an external configuration file, table, or database that the code reads at run-time, along with documentation on how / when to update that value;
- consider using a common configuration system or at least common file format for text-based user-level configuration parameters when a database is not used, rather than inventing a custom format for each use (recommended examples include *YAML*, *INI*, *XML*, *JSON*).

Miscellaneous

Other good practices are hereafter presented.

- Avoid undocumented bare constants in an expressions or formulae. Ideally, define them as constants in a common file for re-use. A good example is `energy_tev = J_TO_TEV * energy_j` and a bad one is `energy=6241509.343260179 * energy0`. In the case a formula used has some specialized constant factors in it that are not common elsewhere, make sure they are documented in the comments immediately before their use or in the function documentation, with references if applicable.
- Use a common package for physical constants, rather than redefining them in multiple source files (example `astropy.constants`, `GSL constants`, or a project-defined common local package). This includes definitions of π and common conversion factors.

[6] This can even be achieved with simple *C++* preprocessor directives wrapping standard console output functions.

- Keep the length of functions or methods to something that can be read in approximately one screen-full at a typical font size. Very long functions generally indicate bad coding and over-complexity. They should be broken into simpler, more readable components.
- Avoid deep nesting of *if / else* statements and other flow-control statements. Instead re-factor to avoid bugs and difficulty in understanding the flow.

4.3 Code Documentation and Comments

Developer (API) Documentation
The API documents shall contain:

- a description for all functions (except obvious things like setters / getters), including any side effects of the function being called, a list of the inputs / outputs, and return values that shall be accompanied by simple descriptions of each including their expected data types;
- a description of the purpose and usage of all classes and modules;
- class diagrams (both hierarchies and collaboration diagrams may be useful).

API documentation shall be written in-code using a tool that can auto-generate API documents. Recommended tools are:

- **C++**: *Doxygen*.
- **Python**: *Sphinx* using *AstroPy / NumPy* documentation conventions inside *docstrings*[7].
- **Java**: *Javadoc* and *Doxygen*.

For small projects, the software developers may also write such documentation by hand. It is recommended to avoid a heavy use of *HTML* tags.

Commenting
Comments in the code shall help a reader (typically another developer) to understand quickly what a piece of code does or intends to do, when it is not obviously clear from the code itself, and also to give the rationale for why the code was written or designed the way it was (i.e. why one algorithm was used, when another was available). They need not be extremely detailed, but shall be written assuming a new developer reads them along with the code. Trivial comments for functions that simply repeat the name of that function shall be avoided.

User Documentation
In addition to API documentation, end-user documentation shall be provided for all CTA software. This may include:

- high-level explanation of the software as a whole;
- guides to running any executable tools, GUIs, etc;
- technical descriptions of algorithms used, with references;
- *FAQ*s, troubleshooting, etc.

The user documentation may be written using the same tool adopted for the developer documentation or another tool (even a plain *Word*, *ReStructuredText*, *Markdown* or *LaTeX* document), as long as it is updated regularly with the code. In a future release of this document, CTAO reserve the right to release templates or documentation styles for the user documentation.

4.4 Functionality and Quality Tests

Unit Tests All Code developed for CTA shall include Unit Tests covering at least 60% of lines of code (80% for Python and non-compiled code) at the level of functions, methods, and classes.

[7] http://docs.astropy.org/en/stable/development/docguide.html

Other Functionality Tests Standards for other tests, for example regression, integration, and performance tests, will be covered in the *Software Quality Assurance Plan* [3]

Quality Tests All code developed for CTA shall pass a predefined set of code quality tests, to be defined in the *Software Quality Assurance Plan* [3].

5 Development Environments

5.1 Compilers, Interpreters, and Runtime Environments

As the production environment of the CTA software will change over time during the operation of CTA, it is essential for the long-term maintainability that CTAO is able to recompile and install the CTA software from source in a reproducible and easy way. Thus, the CTA software shall be delivered as a software package with full installation instructions and documented dependencies on external libraries, compiler versions, interpreters, compile and runtime environments. To this regard, CTA software code shall compile using any compiler / interpreter that provides full support for the language used (accepted languages are listed in section 3). Reliance on a specific compiler or compiler version or compiler-specific features (non-standard pragma commands, etc) shall be avoided to allow future upgrades. The code shall be tested via a CI system against multiple compilers. Recommended compilers are[1]:

- **C++:** *g++* (GCC, `http://gcc.gnu.org`), *Clang++* (Clang or *Apple LLVM*, `http://clang.llvm.org`);
- **Python**: *CPython* + *Cython* from the *Continuum Analytics Anaconda* distribution (`https://www.continuum.io`);
- **Java**: *Oracle Java SE* version 8 runtime environment.

Compiler options to enable standard warnings (e.g. `-wall` for *g++*) shall be used at all times, and code that is intended for a stable release shall not produce any warnings when compiled in the integration or production environment.

5.2 External Libraries and Dependencies

Common libraries across software products shall be used to minimize maintenance and allow for bug-fixes to propagate globally. Table 5.1 shows a list of common libraries for the accepted languages that are convenient for various situations. This list also includes the libraries for coordinate transformations that shall be used in the array and telescope control software.

The list of dependencies and their working versions for a particular software package shall be maintained by the Work-package managers. All external dependencies shall be chosen with maintenance and longevity in mind. All dependencies are considered part of the deliverable software package, even if they are not maintained by CTA developers. Therefore, external libraries shall only be chosen that are both well-maintained and are expected to be maintained over a long period of time. If this cannot be ensured, the software managers shall have an explicit contingency plan for what to do if an essential library is no longer maintained by the external community. Some recommendations to this point are hereafter provided:

- Use some form of abstraction to separate the API of external libraries that may be replaced or changed in the future with the API of internal CTA software code. Provide a common wrapper for

[1] Recommendations will adapt to the evolving support for the different compiler versions.

Recommended Common Libraries			
Purpose	**C++**	**Python**	**Java**
Celestial coordinate transformations and ephemerides	ERFA (SOFA)	AstroPy (ERFA)	JSOFA
Temporal coordinates	ERFA (SOFA)	AstroPy (ERFA)	JSOFA
Numerics / statistics	GSL	SciPy / NumPy	Apache Commons Math
Fitting / optimization	GSL, Minuit	SciPy, Scikit-Learn, iMinuit (Minuit)	Apache Commons Math
Linear algebra	GSL (BLAS) , Eigen	SciPy (BLAS)	Apache Commons Math
Unit Tracking	*TBD*	Astropy.units	*TBD*
Custom socket I/O	ZeroMQ	ZeroMQ	ZeroMQ
FITS file access	cfitsio, gammalib	AstroPy, fitsio (cfitsio), gammalib	nom-tam-fits
Array & telescope middleware	ACS=2017.06	ACS=2017.06	ACS=2017.06

Table 5.1 – Common libraries per accepted language. Underlying libraries are shown in parenthesis, when applicable. This table does not represent all acceptable libraries, only those which are recommended for cross-compatibility between Work-packages. The full list of library dependencies for a Work-package is left to each Work-package development team, and collected centrally. If more commonalities are found, this list will be updated in a subsequent version of this document. Options marked with *TBD* are not yet standardized.

a particular functionality that hides the details of which library implements it, thus allowing a new implementation to be swapped in without affecting existing code.

- Estimate the amount of effort and time needed to re-factor existing code to use a new library.
- In the worst case, consider taking over maintenance of this library internally, and calculate the expected manpower needed to do this over the lifetime of CTAO.

6 Version Numbering and Releases

CTA software packages shall have a consistent version numbers assigned to them. The following rules shall be followed:

- Version numbers shall follow the *semantic versioning* standards[1], where versions have three integer parts: `MAJOR.MINOR.PATCH`.
  1. MAJOR version number, which is incremented when there are incompatible API or functionality changes;
  2. MINOR version number, which is incremented when functionality is added in a backward-compatible manner;
  3. PATCH number, which is incremented when there is a backward-compatible bug fix applied.
- For non-release versions, the version string shall contain more information such that the exact commit can be identified in the revision control system. For example: `mypackage-1.0.dev728`, or `mypackage-1.0.1-post7+git1fecb48` (the number of commits posterior to the version 1.0.1 tag, plus an optional git hash).
- Released versions shall be *tagged* in the revision control system, with stable releases as branches with at least the MAJOR and MINOR version numbers in the branch name, so that older versions can continue to be maintained via patches without interfering with the development of newer versions.
- It is recommended that for each software release, a unique identifier (e.g. a DOI) is generated to make the software release reproducible. Further instructions will be added in a future version of this document.

[1] Semantic versioning is described in detail at `http//semver.org`.

7 Licensing

All CTA software shall include a standard license under which it is released. At the time of writing, the definition of the standard license for CTA software is in progress. The license will be described in a dedicated SW license policy document [4], and a summary added here in a subsequent version of this document.

References

[1] CTAO. *Glossary*. Available in Jama: jama.cta-observatory.org

[2] Martin R.C. (2009). *Clean Code: A Handbook of Agile Software Craftsmanship*. Robert C. Martin Series. Prentice Hall

[3] CTAO Computing. *Software Quality Assurance Plan*. in preparation

[4] —. *CTA Software Licensing Policy*. in preparation

[5] Hamill P. (2004). *Unit Test Frameworks: Tools for High-Quality Software Development*. O'Reilly Series. O'Reilly Media, Inc.

Glossary

Unit Test	Automated tests written and run by software developers to ensure that a section of an application (known as the ”unit”) meets its design and behaves as intended (see e.g. [5]).
Work-package	The term Work-Package is used to refer to the work required for the development of each Product, for example the MST work-package. Products are to be delivered to the CTA Observatory. To its participants a work-package is a project in its own right with interfaces to CTA and other work-packages and products. CTA-GLOS-474.

Acronyms

ACS	ALMA Common Software
ALMA	Atacama Large Millimeter Array
API	Application Programming Interface
CDR	Critical Design Review
CI	Continuous Integration
CTA	Cherenkov Telescope Array
CTAO	Cherenkov Telescope Array Observatory
FPGA	Field Programmable Gate Array
GUI	Graphical User Interface
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
PC	Project Committee
PLC	Programmable Logic Controller
PLD	Programmable Logic Device
PO	Project Office
RHEL	Linux Red Hat Enterprise