Large Synoptic Survey Telescope Design Standard	Document No. **LCA-10099-A**	Status ? LSST Camera APPROVED Effective Date 17 July 2018
	Author(s) J. Chiang R. Dubois	
	Subsystem ? Camera Control System, DAQ, Camera Data Management	
Document Title **LSST Camera Software Standards**		

1 Change History Log

Revision	Effective Date	Description of Changes
A	July 17, 2018	Initial draft release. Release per LCN-1368.

2 Contents

1 Change History Log 1
2 Contents 1
3 List of Tables 2
4 Introduction 2
5 Applicable Documents and Reference Documents 2
6 Definitions 2
6.1 Acronyms 2
6.2 Definitions 3
7 Design Practices 3
7.1 Agile Software Development 3
7.2 Code Reviews 3
8 Implementation Practices 4
8.1 Coding Standards 4
8.1.1 C/C++ 4
8.1.2 Python 4
8.1.3 Java 4
8.2 External Dependencies 4
8.2.1 Test Data Handling 4
8.2.2 CCS 4
8.2.3 DAQ 4
9 Supported Platforms 4
9.1 Data Formats 5
9.2 Code Management 5
9.2.1 Source control 5
9.2.2 Code release 5
9.2.3 Code distribution 5

9.3 Development environment 5
9.4 Build tools 5
9.5 Automated builds 5
10 Verification and Testing 5
10.1 Unit tests 5
10.2 System tests 6
10.3 Issue tracking 7
11 Documentation 7

3 List of Tables

Table 1: Acronyms List 2
Table 2: Definitions 3

4 Introduction

This document describes the policies, procedures, and criteria that the Camera project follows in the development, configuration management, verification and validation, and quality assurance of the Camera software.

5 Applicable Documents and Reference Documents

The following documents are cited for reference.

Ref #	Document Number and Title
[1]	LSE-13, LSST Software Quality Assurannce Guidelines
[2]	LSE-14, LSST Sofware Configuration Managament Guidelines
[3]	LSE-15, LSST Software Verification and Validation Guidelines
[4]	LSE-16, LSST Software Development Plan

6 Definitions

6.1 Acronyms

Table 1: Acronyms List

Acronym	Definition
BNL	Brookhaven National Laboratory
CCS	Camera Control Ssytem
DAQ	Data Acquisition
FRS	Functional Requirements Specification
ICD	Interface Control Document
IN2P3	Institut National de Physique Nucleaire et de Physique des Particules
SLAC	SLAC National Accelerator Laboratory

Acronym	Definition
TBD	To Be Determined
TBR	To Be Resolved
TBS	To Be Specified
TDH	Test Data Handling
VST	Vertical Slice Test

6.2 Definitions

Table 2: Definitions

Term	Definition
DM Stack	LSST Data Management software

7 Design Practices

7.1 Agile Software Development

Camera software design follows Agile software development practices (http://en.wikipedia.org/wiki/Agile_software_development), especially the iterative and incremental methodologies (http://en.wikipedia.org/wiki/Iterative_and_incremental_development) that call for complex systems to be developed incrementally through several cycles wherein portions of the functionality are added at each iteration and which go through a process of testing and evaluation and re-adjustment of the requirements and design. For smaller components, where requirements and use case analysis allows for functionality and interfaces to specified with sufficient detail, test driven development methodologies (http://en.wikipedia.org/wiki/Test_driven_development) are employed to drive the implementation.

7.2 Code Reviews

Software packages will undergo code reviews to ensure the overall package design and architecture adheres to our software standards. Code reviews will occur when updates to packages are made that are to be merged into the main branch for release.

The primary goals of a code review are to address the following questions:

- Does the code satisfy the design requirements?
  - Explicit requirements for functionality
  - Non-functional requirements like memory usage, performance, etc..
- Is the code maintainable?
  - Understandable by a reasonable programmer
  - Adequately documented

8 Implementation Practices

8.1 Coding Standards

8.1.1 C/C++

We have adopted CERN coding standards for C/C++: http://pst.cern.ch/HandBookWorkBook/Handbook/Programming/CodingStandard/c++standard.pdf. These and similar standards have been used successfully for large high energy physics experiments at CERN, SLAC, Fermilab, and other DOE labs. They are familiar to Camera software developers, almost all of whom have developed software for past HEP experiments.

8.1.2 Python

We use the Python coding standards by Guido van Rossum and Barry Warsaw, http://www.python.org/dev/peps/pep-0008/. These are the industry standards for Python development and are used by the developers of the Python language itself. LSST Data Management has adopted these standards with minor revisions.

8.1.3 Java

We use the standard Java coding conventions: http://www.oracle.com/technetwork/java/codeconventions-150003.pdf.

8.2 External Dependencies

8.2.1 Test Data Handling

External libraries will comprise industry-standard third-party packages that supply the required functionality. For producing data products, the cfitsio library (http://heasarc.gsfc.nasa.gov/fitsio/) will be used. The following third-party Python modules will be used to provide plotting, numerical algorithms, and FITS-file and database access: matplotlib, numpy, scipy, pyfits, MySQL-Python. Finally, the LSST Data Management software will be used for analysis of the pixel data acquired at the sensor-level through the full focal plane. The DM stack also contains or uses many of the aforementioned libraries.

8.2.2 CCS

We use maven to manage/download external Java dependencies. The design philosophy is to try to have service layers to isolate subsystem code from specific external dependencies (e.g. JMS, JGroups) to allow migration to new external dependencies as appropriate in future. We have some native code dependencies (for example the DAQ interface libraries) which are currently handled in an ad-hoc way.

8.2.3 DAQ

TBS.

9 Supported Platforms

The platforms for development and production work will be Redhat 5/Scientific Linux 5 (64 bit) and Redhat 6/Scientific Linux (64 bit). These are the posix-based operating systems supported at SLAC for interactive and batch use.

9.1 Data Formats

FITS (Flexible Image and Transport System) will be used for image (and tabular) data (http://fits.gsfc.nasa.gov/fits_standard.html). This is the standard data format for all astronomical data.

The Sensor group has developed a set of filename, header, and directory structure standards (https://docushare.lsstcorp.org/docushare/dsweb/Get/LCA-10140/EOTest_File_Spec_v1.doc).

9.2 Code Management

9.2.1 Source control

The git repository at git@git.lsstcorp.org will be the repository for all Camera code. git is an open source tool which manages files and directories and the changes made to them over time allowing the recovery of older versions of an object or the examination of how the object changed.

9.2.2 Code release

TBD.

9.2.3 Code distribution

TBD.

9.3 Development environment

In accord with the open-source requirement for all LSST production code, we will use the Gnu Compiler Collection for C/C++ code. Specific editors or IDEs are not mandated.

9.4 Build tools

We use the industry standard build tools, SCons for C/C++ and Python and Maven for Java.

9.5 Automated builds

The Camera software will be use the Jenkins Continuous Integration tool (http://jenkins-ci.org/) for ensuring that the various Camera software components inter-operate correctly with one another. Jenkins is an application that enables building and testing of software projects continuously, as new development is added to the project.

10 Verification and Testing

10.1 Unit tests

The scope of unit testing is to verify the design and implementation of all components from the lowest level defined in the detailed design up to and including the lowest level in the architectural design. The inputs to the unit testing process are the successfully compiled modules. These are iteratively assembled and tested during unit testing until the unit test validates components in the architectural design. The successfully unit tested architectural design components are the outputs of unit testing process. LSST Camera software generally uses a bottom-up assembly sequence to iteratively compose each architectural component.

The unit test description should provide the sequence for assembling the architectural design units and the types of tests necessary for individual modules.

The Unit Test Plan's test cases should be developed from the detailed design of the baseline. The type of tests performed during unit testing may include:

- White-box Tests: designed by examining the internal logic of each module and defining the input data sets that force the execution of different paths through the logic. Each input data set is a test case.
- Black-box Tests: designed by examining the specification of each module and defining input data sets that will result in different behavior (e.g. outputs). Black-box tests should be designed to exercise the software for its whole range of inputs. Each input data set is a test case.
- Performance Tests: if the detailed design placed resource constraints on the performance of a module, compliance with these constraints should be tested.

xUnit compliant (or near-compliant) test frameworks will be used: boost.test, unittest, JUnit.

10.2 System tests

The scope of system testing is to verify compliance with the system objectives as stated in the FRS. A test should be defined for every essential software requirement, and for every desirable requirement that has been implemented. The input to system testing is the successfully integrated system.

The System Test Plan's test cases will be developed from the FRS requirements selected for the baseline. Black-box and other types of test should be used wherever possible. When a test of a requirement is not possible, an alternative method of verification should be used. The type of tests performed may include:

- **Function Tests** should be designed using techniques such as decision tables, state-transition tables and error guessing to verify the functional requirements.
- **Performance Tests** should be designed to verify:
  - that all worst case performance targets have been met;
  - that nominal performance targets are usually achieved;
  - whether any best-case performance targets have been met; and
  - should be designed to measure the absolute limits of performance.
- **Interface Tests** should be designed to verify conformance to external interface requirements. Interface Control Documents (ICDs) form the baseline for testing external interfaces. Simulators will be necessary if the software cannot be tested in the operational environment.
- **Usability Tests** should verify the user interface, man machine interface, or human computer interaction requirements,and logistical and organizational requirements.
- **Load Tests** should be designed to verify requirements for the usage of resources such as CPU time, storage space and memory. The best way to test for compliance is to allocate these resources and no more, so that a failure occurs if a resource is exhausted.
- **Security Tests** should check that the system is protected against threats to integrity and availability. Tests should be designed to verify that basic security mechanisms specified in the System Engineering Requirements have been provided.

- **Compatibility Tests** should attempt to verify portability by running a representative selection of system tests in all the required environments.
- **Stress tests** evaluate a system at or beyond the limits of its specified requirements. Testers should look for inputs that have no constraint on capacity and design tests to check whether undocumented constraints do exist.

10.3 Issue tracking

Reporters of anomalous behavior of Camera software should use the JIRA (https://jira.slac.stanford.edu/secure/Dashboard.jspa) issue tracking system. The workflow of the issue life cycle from inception to resolution is recorded within the JIRA system.

11 Documentation

The industry standard Doxygen tool (http://www.stack.nl/~dimitri/doxygen/) will be used for code-level documentation of C++ and Python code. This documentation will be generated for each software release. Following standard conventions, Python doc-strings will be used to provide a minimal amount of user-level documentation.