European Organisation for Astronomical Research in the Southern Hemisphere

Programme: E-ELT

Project/WP: E-ELT Telescope Control

Guide to Developing Software for the EELT

Document Number: ESO-288431

Document Version: 3

Document Type: Manual (MAN)

Released On: 2019-05-22

Document Classification: ESO Internal [Confidential for Non-ESO Staff]

Owner:	Pellegrin, Federico
Validated by PA/QA:	Kurlandczyk, Hervé
Validated by WPM:	Kornweibel, Nick
Approved by PM:	Kornweibel, Nick
	Name

Authors

Name	Affiliation
F. Pellegrin	ESO
S. Feyrin	ESO

Change Record from previous Version

Affected Section(s)	Changes / Reason / Remarks
All	Updates for new release of ELT DEV
3.1	New directory structure based on RootAreas.docx 03/04/2018
3.8	Added wtools documentation
3.2-3.3	Added lmod documentation

Contents

1. Introduction

This document contains a brief description, a series of typical usage cases and usage practices of the ELT development tools. The aim of the document is to introduce and simplify the usage of the ELT development tools and introduce practices considered good by ESO.

This document should be used in conjunction with the various tools official documentation and usage examples, as this document does not pretend to be a complete documentation of the mentioned tools but just an additional ELT specific support.

1.1 Scope

The target audience of this document are software developers and software integrators working on ELT software. Software project managers may also receive insights on how the development tools are structured.

1.2 Definitions and Conventions

1.2.1 Abbreviations and Acronyms

The following abbreviations and acronyms are used in this document:

TBC	To Be Confirmed
TBD	To Be Defined
VM	Virtual Machine
CS	Control System
EULA	End User License Agreement
DevEnv	(ELT CS) Software Development Environment

1.2.2 Stylistics Conventions

Courier font is used to indicate source of code or scripts while courier bold font is used to indicate execution on the system, either input or outputs.

1.2.2.1 Example code

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
       printf("Hello world!\n");
       return 0;
}
```

1.2.2.2 Example execution

```
fede@esopc ~/waf/example/pkg1/exProgC $ ps
  PID TTY          TIME CMD
 9331 pts/8    00:00:00 bash
23663 pts/8    00:00:00 ps
```

2. Related Documents

2.1 Applicable Documents

The following documents, of the exact version shown, form part of this document to the extent specified herein. In the event of conflict between the documents referenced herein and the content of this document, the content of this document shall be considered as superseding.

2.1.1 ESO Documents

AD1 ELT Linux Installation guide;
ESO-287339 Version 2

AD2 ELT SVN Usage Guidelines;
ESO-283837 Version 1

AD3 Control GUI Developers Guidelines;
ESO-288608 Version 1

2.1.2 Standards

AD4 ELT Programming Language Coding Standards;
ESO-254539 Version 1

2.2 Reference Documents

The following documents, of the exact version shown herein, are listed as background references only. They are not to be construed as a binding complement to the present document.

RD1 VLT Software Development Platform Lessons Learned
ESO-289257

RD2 *<Document name>;*
<Drawing Number Revision *X>*

3. Linux Development Tools

3.1 Generic structure indications

Based on experiences from VLT software there are a series of recommendations given. For a full comparison and rationale refer to RD1.

Definition: A software module is a piece of software (code and documentation) able to perform functions and having an interface available to an external user to access the functions provided. Technically a module is a way to organize functions in homogeneous groups. The interface hides the implementation and system dependencies from the user. Managerially the module is the basic unit for planning, project control, and configuration control.

Definition: A software package is a logical grouping of software modules. The only reason of the grouping is to simplify the management of the modules in a hierarchic way. Multiple level of package hierarchy is permitted (<pkg1>/<pkg1-1>/<pkg1-1-1>/<module>)

Naming: Each module is identified by a name and is therefore unique in the project. The module_name can be made of a minimum of two up to a maximum of sixteen, suggested six, characters (a-z, 0-9) and shall be unique in the project. Names equal or too similar to UNIX names shall be avoided. The case cannot be used to build different names: i.e., the following are referring to the same module: xyz, XYZ, xYz. The module_name is used in the naming of all elements that belong to the software module. The module_name should start with an alphabetic character (a-z). for the ELT it has been decided to use namespaces (as it is the case for the majority of Linux SW applications) instead of using unique module names. Still it is required to have unique binary names. The following convention is suggested: the package name is prefix to the module name. This has to be done using lower-camel-case, for example packageModule or packageModuleFeature.

Each software module should produce only one artefact (one <module> per produced artefact) where “artefact” is like a program, a library, etc.

The module should include the build script (wscript for waf) which includes the specification of the processor, (cross-)compiler, compiler options, etc., so the build systems knows for which HW to build the artefact.

The binaries and libraries are produced by the build system in a local temporary directory (build) so that is does not pollute the source code. The local temporary directory repeats the structure of the source tree, so there are no names collisions.

In case of C/C++, public include files are located in a dedicated directory within src (to facilitate the identification of the include files to install, alternatively this information would have to be added to the build or installation script): src/include/<a>. <a> can either replicate the directory structure of the package and map to the namespace or use a free-form structure. It is evident that when using a free-form structure a name collision could happen and this has to be managed by the developer.

The recommended directory structure is to use a one level directory named after the module, which name is guaranteed to be unique by module definition.

For example:

- the most descriptive <a>=package/directory/structure/module/ or

- the suggested <a>=module/ or
- a free form structure <a>=freeform/structure/.

Generated code to be archived could also go in a dedicated gen/ directory within src/ while the test/ directory contains unit tests.

Integration tests should have their own artefact or even a separate module/artefact depending on the type of interaction required: integrating several artifacts in a module or integrating several modules.

The resource/config/ directory contains runtime configuration data which are “read only”, not subject to modification during execution like the CDT in the VLT. Configuration information may be provided by the configuration service of the SW platform; however, it should be possible to have a local representation.

The resource/data/ directory contains runtime configuration data which may need to be modified during/after execution (like calibration data).

The suggested structure, able to support different programming languages and different target types, is the following:

```
<package>/<module>/src          # Any source files (incl, headers, mocs etc.)
                  /src/include/<a>   # public include files in case of C/C++
                  /src/gen/     # generated code to be archived
                  /resource     # Resources (e.g. GUI glyphs, sounds, configs etc.)
                  /interface    # Interface specifications (IDL, mockups?)
                  /doc          # non-generated documentation
                  /test         # unit tests
                  wscript       # build script
```

Inside the *resource/* directory a tree of different subdirectories will be present to differentiate between type of resources. The types we see now are:

<module>/resource/<directory path to module>/

- *config/* - contains default configuration files
- *audio/* - contains sounds, music and other audible files
- *image/* - contains images and other visual artefacts
- *model/* - contains models
- *dictionary/* - contains dictionaries
- *data/* - contains runtime configuration

Inside each resource subdirectory, as now, the structure is free form but by convention the rule is to create a subdirectory structure that clearly identifies the path to the module. In case of shared resources the subdirectory with the module path may not needed.

The build system is supposed to just recursively copy the structure to the destination directory.

Additionally, a set of directories, which are pointed by same named environment variables, are defined where the various result of an installation or execution can be stored:

- System Root (SYSROOT): directory delivered by the ELT project that contains basic software and resources widely shared between everybody in the project. This includes for example source templates, basic libraries, basic utilities, build system support and so on. This is installed on the system and is read-only to the user and instrument manager.
- Integration Root (INTROOT): directory where the user, instrument for example, builds and installs specific software and support files. Default configuration files and resources are also part of the Integration Root. This is installed read-only to the user and is populated by the instrument manager.
- Data Root (DATAROOT): directory where all generated output data is stored. This is read-write for the user.
- Configuration Root (CFGROOT): directory where all instance configuration is stored. This is read-write for the user.

All the described Root areas are physical areas on the filesystem. Nevertheless, especially for the Integration Root that may be assembled from multiple projects output, the usage of *filesystem overlaying* may be introduced in the future.

3.2 Environmental Modules System (Lmod)

Environment Modules provide a convenient way to dynamically change the users' environment through modulefiles. This includes easily adding or removing directories to the PATH environment variable.

The software Lmod, a Lua based environment module system, is used in the ELT Development environment and replace the former VLT pecs.

A modulefile contains the necessary information to allow a user to run a particular application or provide access to a particular library. All of this can be done dynamically without logging out and back in. Modulefiles for applications modify the user's path to make access easy. Modulefiles for Library packages provide environment variables that specify where the library and header files can be found.

Packages can be loaded and unloaded cleanly through the module system.

It is also very easy to switch between different versions of a package or remove it.

The latest online user guide of Lmod can be found under:

https://lmod.readthedocs.io/en/latest/010_user.html

The modulefile contains commands to add to the PATH or set environment variables. When loading the modulefile the commands are followed and when unloading the modulefile the actions are reversed.

Example of commands which can be used in a modulefile:

```
prepend_path("PATH", value)
setenv("NAME", value)
set_alias("name","value")
family("name")
load("pkgA", "pkgB", "pkgC")
```

The standard ELT development environment provides some lua files to set default variables (PATH, LD_LIBRARY_PATH …). There are located under:

```
/eelt/System/modulefiles
```

```
> ll /eelt/System/modulefiles
total 12
-rwxr-xr-x  1 eeltmgr eelt 1813 Jul  6 10:25 eeltdev.lua
-rwxr-xr-x  1 eeltmgr eelt  531 Jun 28 14:36 introot.lua
drwxr-xr-x. 2 eeltmgr eelt   46 Jul  4 09:00 jdk
-rwxr-xr-x  1 eeltmgr eelt  149 Apr 10 12:44 python.lua
```

The eeltdev.lua file is loaded by default at login and contains all the default setting (including the load of package python and jdk)

The introot.lua can be loaded by the user to set up the PATH, LD_LIBRARY_PATH.. when he defines an INTROOT.

In addition, the user can define private lua files under the directory :

```
$HOME/modulefiles
~/modulefiles 1062 > ll
total 12
-rw-r--r-- 1 eeltmgr eelt 129 Jul  5 07:37 private-eltint20.lua
-rw-r--r-- 1 eeltmgr eelt  61 Jul  4 09:47 private-eltint21.lua
-rw-r--r-- 1 eeltmgr eelt 130 Jul  6 10:03 private.lua
```

And from this directory, Lmod will make available all the lua file and load by default, the files `private.lua` and `private-<hostname>.lua` if exist.

3.2.1 Lmod basic commands

```
$ module help         # display Lmod help message
$ module list         # list of modules loaded
$ module show package # Display what is executed by the module

$ module avail        # list of modules available to be loaded
```

Lmod uses the directories listed in `$`MODULEPATH to find the modulefiles to load, /eelt/System/modulefiles and $HOME/modulefiles are added by default.

With the sub-command `avail` Lmod reports only the modules that are in the current MODULEPATH. Those are the only modules that the user can load.

User can add/remove a directory to the MODULEPATH with;

```
$ module use /path/to/modulefiles    # Add the directory to $MODULEPATH search path
$ module unuse /path/to/modulefiles # Remove directory from $MODULEPATH
```

A user logs in with the standard modules loaded. Then the user modifies the default setup through the standard module commands:

```
$ module load package1 package2 …        # load modules
$ module unload package1 package2 …      # unload modules
```

Once users have the desired modules load then they can issue:

```
$ module save
```

This creates a file called `~/.lmod.d/default` which has the list of desired modules (collection). This default collection will be the user's initial set of modules (loaded at login).

Users can have as many collections as they like. They can save to a named collection with:

```
$ module save <collection_name>
```

And, at any time, it is possible to restore the set of modules saved in that named collection with:

```
$ module restore <collection_name>
```

A user can print the contents of a collection with:

```
$ module describe <collection_name>
```

Examples:

```
eltint20 eeltmgr:> module list
Currently Loaded Modules:
  1) jdk/java-openjdk   2) python   3) eeltdev   4) private-eltint20   5) private
eltint20 eeltmgr:> module avail

------------------ /home/eeltmgr/modulefiles -------------------------------------
   private (L)    private-eltint20 (L)    private-eltint21

------------------ /eelt/System/modulefiles --------------------------------------
   eeltdev (L)    introot    jdk/java-openjdk (L)    python (L)

------------------ /usr/share/lmod/lmod/modulefiles/Core -------------------------
   lmod/6.5.1    settarg/6.5.1

  Where:
   L:  Module is loaded
eltint20 eeltmgr:~ 1002 > module load introot
eltint20 eeltmgr:~ 1003 > module list

Currently Loaded Modules:
  1) jdk/java-openjdk   2) python   3) eeltdev   4) private   5) private-eltint20
6) introot
eltint20 eeltmgr:~ 1004 > module save
Saved current collection of modules to: default
```

3.3 ELT Common Basic Software

The System Root (SYSROOT) delivered by the ELT project is by default installed under: the directory `/eelt/`.

It is distributed with the RPM `eelt-common- X.Y.Z -n`

The default location for the SYSROOT is defined in the default lua file by the variable SYSROOT:

```
SYSROOT=/eelt/X.Y.Z
```

where X.Y.Z is the version of the rpm eelt-common- X.Y.Z -n

The SYSROOT includes in particular the following:

- build system support wtools
- getTemplate utility, used to generate module, wscript… from template
- elt-devenv utility, to highlight modifications introduced on the default ELT installation

It is installed by user eeltmgr and is read-only to the user and instrument manager.

3.4 Inline documentation

As indicated in AD4 the inline documentation in source code files should be managed using the Doxygen format. This gives the possibility to generate at the end a unique documentation even if the project consists of different languages source code files.

Nevertheless, language specific extensions to Doxygen can be managed using additional custom filters. The specific language filters present in the ELT Linux Development environment are discussed in the language section.

Configuration options to Doxygen should be passed via the configuration file and not by command line or otherwise. A template Doxygen configuration file can be generated with:

```
fede@esopc ~/waf/example $ doxygen -g mytemplate.config

Configuration file `mytemplate.config' created.

Now edit the configuration file and enter

  doxygen mytemplate.config

to generate the documentation for your project
```

The configuration should be instructed to generate at least HTML documentation which is the most used in the ELT software development.

To improve readability and organization of the documentation it is highly suggested to make good use of Doxygen *groups*, entities that permit to group together similar topics into a common documentation section. A simple grouping can be done by reflecting the directory structure: therefore, defining for each package a group, for each module a group that is part of the package group and then adding every artifact in the module to the module group.

This would create a documentation group structure that totally reflects the filesystem structure, making it easy to find and maintain the information needed.

To define a group the Doxygen directive *\defgroup* can be used, for example:

```
\defgroup groupName Long Description of the group
```

And once defined the group can be referenced as:

```
\ingroup groupName
```

Additional details can be found in the *Grouping* section of the Doxygen manual.

3.5 C++ Tools

3.5.1 Compiler suite

The ELT Programming Languages Coding Standards (AD4) specify the usage of the standard C++11 which requires the GNU Compiler Suite of version 5.x or higher for an appropriate support.

At present the ELT Linux Development environment ships both 4.x (specifically 4.8.5) and 7.x (specifically 7.2.1) since some commercial packages used by ELT software still do not support the 7.x family of compilers and therefore it is impossible, or better very unpractical due to ABI incompatibilities, to use that libraries with the newer compilers. It is foreseen that in the near future the 4.x family will be dropped and the 7.x (or possibly higher) will be the only supported and maintained.

The 4.x family is currently installed as system packages and is seen by default from the command line. The 7.x family is shipped using with the devtoolset-7 software collection and is located under /opt/rh/devtoolset-7/.

3.5.2 Unit testing framework

The framework to write C++ unit tests is the Google Test framework. It is installed on the system in /opt/gtest or can be alternatively used as an external and built directly in the project source tree.

3.5.3 Static checking tools

The ELT Linux development environment currently ships *cpplint* (in /opt/cpplint) for source code style checking and *cppcheck* (installed system wide in default path) for common programming errors checking.

Example command line execution for the tools, considering the necessity not to include in the output temporary build files, INTROOT artefacts or supporting wtools directories in the module structure:

```
/opt/cpplint/cpplint.py --output=vs7 --linelength=100 `find . -name *.cpp -o -name *.h -o -name *.hpp |  grep -v "./build/\\|./INTROOT/\\|./wtools/" ` 2> cpplint.out
```

```
cppcheck -f -j 8 --enable=warning,style,performance,portability --xml --xml-version=2 `find . -name *.cpp -o -name *.h -o -name *.hpp -o -name *.c | grep -v "./build/\\|./INTROOT/\\|./wtools/\\|/config/"` 2> $WORKSPACE/cppcheck.xml
```

3.5.4 Dynamic checking tools

Several dynamic checking tools are included in the ELT Linux development environment and their usage is highly encouraged:

- *gdb*, the GNU debugger
- *valgrind*, a suite of tools for debugging and profiling
- *strace*, the system calls and signal tracer
- *gcov*, the GNU coverage library and tools to analyse the code coverage amount. The programs and libraries have to be compiled with coverage options enabled to use the feature provided, namely:
  - compiler flags to be added: -O0 -fprofile-arcs -ftest-coverage
  - linker flags to be added: -lgcov
- To produce easily readable HTML or XML output from the *gcov* binary data the tool *gcovr* is supplied in the */opt/gcovr* directory.

3.6 Python Tools

3.6.1 Python interpreter

The ELT Linux development environment uses the Anaconda Python distribution as the main Python environment based on Python 3.5.x version. The distribution is installed in */opt/anaconda3* and can be easily added to the user's environment executing:

```
source /opt/anaconda3/bin/activate
```

Or alternatively by manually setting up the binary and library paths in the personal environment. By default the local configuration of LMOD will already set up the Anaconda Python distribution as the default one.

The ELT Linux development ships also a 2.7.x version of Python as this is used by the underlying distribution the system is based. The usage of this version is not possible as the ELT Programming Languages Coding Standards (AD4) specify that Python 3.x language must be used.

The Anaconda distribution is shipped with all the default Python modules updated at the date of creation with the addition of some others deemed of use for the ELT. A list of the installed modules and their versions can be retrieved with the command

```
conda list
```

Private Python environment can be created and managed using the conda tool. A good starting point is executing the conda environment help request:

```
conda env --help
usage: conda-env [-h] {attach,create,export,list,remove,upload,update} ...

positional arguments:
  {attach,create,export,list,remove,upload,update}
    attach              Embeds information describing your conda environment
                        into the notebook metadata
    create              Create an environment based on an environment file
    export              Export a given environment
    list                List the Conda environments
    remove              Remove an environment
    upload              Upload an environment to anaconda.org
    update              Update the current environment based on environment
                        file

optional arguments:
  -h, --help            Show this help message and exit.
```

3.6.2 Python test framework

The Python code unit tests can be performed in two ways:

- By writing in-lined *doctest* tests inside the source code.
- By writing totally separate tests under the *test/* subdirectory that use the standard Python *unittest* library.

In both cases the tests can be run by calling directly the Python interpreter and eventually executing the test starter. To uniform the test execution the usage of the test runner *nosetest* that is present in the environment is highly encouraged.

The test runner *nosetest* supports also code coverage calculation and report generation when executed with the *–with-coverage* option. Relevant command line options for Python code coverage calculation and reporting follow:

```
  --with-coverage       Enable plugin Coverage:  Activate a coverage report
                        using Ned Batchelder's coverage module.
                        [NOSE_WITH_COVERAGE]
  --cover-package=PACKAGE
                        Restrict coverage output to selected packages
                        [NOSE_COVER_PACKAGE]
  --cover-erase         Erase previously collected coverage statistics before
                        run
  --cover-tests         Include test modules in coverage report
                        [NOSE_COVER_TESTS]
  --cover-min-percentage=COVER_MIN_PERCENTAGE
                        Minimum percentage of coverage for tests to pass
                        [NOSE_COVER_MIN_PERCENTAGE]
  --cover-inclusive     Include all python files under working directory in
                        coverage report.  Useful for discovering holes in test
                        coverage if not all files are imported by the test
                        suite. [NOSE_COVER_INCLUSIVE]
  --cover-html          Produce HTML coverage information
  --cover-html-dir=DIR  Produce HTML coverage information in dir
  --cover-branches      Include branch coverage in coverage report
                        [NOSE_COVER_BRANCHES]
  --cover-xml           Produce XML coverage information
  --cover-xml-file=FILE
                        Produce XML coverage information in file
```

3.6.2.1 Python doctest example

```
def helloworld(name):
        """ Returns a greeting to the person passed as a parameter

        >>> print(helloworld("Teri"))
        helloworld Teri
```

```
        >>> print(helloworld("Ale"))
        helloworld Ale

        """
        return "helloworld "+name


if __name__ == "__main__":
        import doctest
        doctest.testmod()
```

Test execution can be either calling directly the script (due to the code in *__main__*, by default just if a test fails the script will produce some output otherwise it will be silent) or by using nosetests as follows:

```
fede@esopc ~/waf/example/pkg1/exPython $ nosetests --with-doctest src/hello/hello.py
.
----------------------------------------------------------------------
Ran 1 test in 0.004s

OK
```

It is important to notice that in some cases, for example when using together with Qt5 libraries, the usage of doctest can be problematic due to the necessity to scan and initialize multiple libraries. It is therefore suggested not to use doctest for testing complex situations or modules with high level library dependencies.

3.6.2.2 Python unittest example

The following unittest example is based on the previous helloworld module used as an example in the previous section:

```
#! /usr/bin/env python
# encoding: utf-8

import unittest
from hello import hello

class test_helloworld(unittest.TestCase):
        def test_helloworld_print1(self):
                self.assertEqual('helloworld Teri', hello.helloworld("Teri"))

        def test_helloworld_print2(self):
                self.assertEqual('helloworld Ale', hello.helloworld("Ale"))
```

Similarly the tests can be executed either command line for example using:

```
python -B -m unittest discover
```

Or using nosetests:

```
fede@esopc ~/waf/example/pkg1/exPython $ nosetests test/testHelloworld.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
```

3.6.3 Python tools

The tool for checking Python code for code style and common errors is pylint, included in the Anaconda distribution. The graphical frontend pylint-gui can be also useful.

Example command line execution for pylint:

```
PYTHONPATH=.;for i in `find . -name "src" -type d -exec readlink -f {} \\; |  sort | uniq`; do export PYTHONPATH=$PYTHONPATH:${i}; done; pylint -f parseable `find . -name *.py | grep -v "./build/\\|./INTROOT/\\|./wtools/"` > pylint.log
```

The command line is pretty complicated as it tries to set the PYTHONPATH in the project for the various modules, so dependant modules can see each other when they are executing an import statement.

3.6.4 Python documentation

Python documentation using doxygen is enhanced in the ELT Linux development environment using the doxypypy Python module (https://github.com/Feneric/doxypypy) that extends the Doxygen notation with specific Python language constructs.

Doxypypy is generally recalled by a wrapper script named *py_filter* which just passes some default parameters to it:

```
#!/bin/bash
doxypypy -a -c $1
```

3.7 Java Tools

3.7.1 Java Software Development Kit

The Java SDK used in the ELT Linux development environment is the OpenJDK 1.8.x. It is shipped as a system package and its compiler and bytecode interpreter are seen in the user's default path.

```
fede@esopc:~ $ java -version
java version "1.8.0_161"
Java(TM) SE Runtime Environment (build 1.8.0_161-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.161-b14, mixed mode)
```

3.7.2 Unit test framework

The unit test framework for Java in ELT Linux development environment is TestNG (http://testng.org/doc/index.html) and is located under /opt/testing directory. Execution of tests with TestNG require the preparation of an XML file containing the test description and the execution of the code using the TestNG runner class *org.junit.runner.JUnitCore*.

For code coverage calculation and reporting the JaCoCo project (http://www.jacoco.org/) is used and is located under /opt/jacoco directory. To generate the binary coverage data the tests have to be run using the JaCoCo jar as a Java agent (using the *-javaagent* command line option). The jacococli.jar package, JaCoCo command line interface, can be used to generate easily readable HTML coverage reports.

3.7.3 Java code checking tools

The ELT Linux development environment offers mainly two tools for Java code checking:

- checkstyle (in /opt/checkstyle-7.x.y) for implementing code style checking.
  Example execution:

```
java -jar /opt/checkstyle-7.1.1/checkstyle-7.1.1-all.jar -f xml -c /checks.xml *
```

- findbugs (in /opt/findbugs-3.x.y) for common coding bugs checking.
  Example execution:

```
FINDBUGS_HOME=/opt/findbugs-3.0.1 $FINDBUGS_HOME/bin/findbugs -xml Test.jar
```

3.8 GUI Toolkit

The GUI Toolkit selected for the ELT project is QT5 (https://www.qt.io/). The ELT Linux development environment provides QT5 libraries for C++ and Python language bindings. Python language bindings for QT5 are currently via the Python module PyQt5.

Further information about the GUI developing can be found in AD3.

3.9 Build system

The build system used for ELT Linux software is waf (https://waf.io). waf is a rather recent build system written in Python which natively support C, C++, Java and obviously Python, along with many other languages, such as for example D, C#, Ocaml and Fortran, and many toolkits, such as Qt or glib. Build scripts in waf are written in Python and therefore particular customizations can be made using this full programming language, making the tool very powerful and without the need to learn some specific macro language specific to the build system as in the other cases. The build system can be further customized using “tools” which expand the build system to other languages or interfaces. Nevertheless, for standard cases of the supported languages the syntax is very easy and intuitive, the learning curve is very gentle.

To further ease the development process on the ELT, ESO prepared an additional layer on top of waf that simplifies the configuration scripts of the most common ELT related software (as for example in the past the vltMakefile for VLT software and acsMakefile for ALMA software). Usage of this layer, named **wtools**, also makes future enhancements and upgrades much easier as they are concentrated into a single library instead of being spread throughout multiple configuration files. Nevertheless, when advanced features not supported by wtools are needed, native waf/Python code can be used to override and augment the functionalities for a single module.

So, usage of wtools is **highly desired**, although a basic knowledge of waf, presented hereafter, is very useful to understand the basic functions of the build system and to give the tools for very specific customization for specific advanced needs of modules. An introduction to wtools is given in the section 3.10.

The version of waf must be 1.9.5 or greater, as from this version on the full support for the QT graphical toolkit for both C++ and Python has been natively added.

The reference documentation for waf is the Waf Book at https://waf.io/book/

3.9.1 Introduction to waf scripts

The waf build system used as a configuration file a so called **wscript**. Therefore, the first step to work with waf is to prepare such a script, for example given a single file C++ in the directory src named exProgC.cpp from which we want to compile an executable named exProgC this could look like:

```
# encoding: utf-8

def configure(conf):
       # We are using C++
        conf.load('compiler_cxx')

def options(opt):
       # We are using C++
        opt.load('compiler_cxx')

def build(bld):
       # Define the main program.
       bld.program(source='src/exProgC.cpp', target='exProgC')
```

As the waf build system contains also the configuration step of the build procedure, something usually separate in other packages such as GNU Make or CMake, a wscript will usually contain a *configure* and an *options* section that contain respectively the configuration, done when explicitly requested by the *waf configure* execution, and options to be used for the build. A configuration step of the example may look as follows:

```
fede@esopc ~/waf/example/pkg1/exProgC $ waf configure
Setting top to                           : /home/fede/waf/example/pkg1/exProgC
Setting out to                           : /home/fede/waf/example/pkg1/exProgC/build
Checking for 'g++' (C++ compiler)        : /usr/bin/g++
'configure' finished successfully (0.033s)
```

The execution of the configure stage will create the build directory, where all temporary build files and the results are stored. Depending on the configuration requested waf will also check for the necessary tools needed, for example compilers or libraries, and report them in this stage. All this data is stored so further steps can then rely on them for a faster and coherent execution.

A more articulate example, using Python and PyQt extensions, of configuration may look like this:

```
fede@esopc ~/waf/example/pkg2/exPyqt5 $ waf configure
Setting top to                           : /home/fede/waf/example/pkg2/exPyqt5
Setting out to                           : /home/fede/waf/example/pkg2/exPyqt5/build
Checking for program 'python'            : /usr/bin/python
Checking for program 'pyuic5, pyside2-uic, pyuic4' : /usr/bin/pyuic5
Checking for program 'pyrcc5, pyside2-rcc, pyrcc4' : /usr/bin/pyrcc5
Checking for program 'pylupdate5, pyside2-lupdate, pylupdate4' : /usr/bin/pylupdate5
```

```
Checking for program 'lrelease-qt5, lrelease'                 : /usr/bin/lrelease
Checking for python version >= 2.7.4                          : 2.7.6
'configure' finished successfully (0.099s)
```

Once the build tree is configured the build, defined in the *build* section of the wscript, can be executed with *waf build*:

```
fede@esopc ~/waf/example/pkg1/exProgC $ waf build
Waf: Entering directory `/home/fede/waf/example/pkg1/exProgC/build'
[1/2] Compiling src/exProgC.cpp
[2/2] Linking build/exProgC
Waf: Leaving directory `/home/fede/waf/example/pkg1/exProgC/build'
'build' finished successfully (0.088s)
```

The output shows how first the cpp file is compiled and then linked, to create the desired executable. Everything is done under the build directory. By default waf if also run unit tests, if any, defined in the configuration file. The *–notests* option can be added on the command line not to execute the command line tests.

Of course as waf keeps track of file changes it will not rebuild parts of the software that are not necessary to be rebuilt. Executing immediately the same command as before once more it will therefore lead to:

```
fede@esopc ~/waf/example/pkg1/exProgC $ waf build
Waf: Entering directory `/home/fede/waf/example/pkg1/exProgC/build'
Waf: Leaving directory `/home/fede/waf/example/pkg1/exProgC/build'
'build' finished successfully (0.015s)
```

Usually used command line invocations for waf include also *clean*, to clean up the build but keeping the configuration data, *distclean*, to totally clean up every file waf generated, and *install*, to install the generated results on the final filesystem (as specified by the configuration script or in general by default in the */usr/local* directory tree).

Of course the configuration part can contain specific customization, a few examples:

```
# Check for gtest library on the system
conf.check(compiler='cxx',lib='gtest',mandatory=True, use='GTEST')

# Check that Python is at least 3.4.0
conf.check_python_version((3,4,0))

# Set some flags to the compiler flags variable
conf.env.append_unique('CXXFLAGS', ['-g', '-O2'])
```

The text encoding comment at the beginning of the wscript files is not mandatory when Python version 3.x is used as by the PEP8 Style Guide the UTF-8 is the default. As one of waf goals is to be both 2.7.x and 3.x compatible, the text encoding comment may be often found in build script examples.

3.9.2 waf and the ELT directory structure

As mentioned in section 3.1 it is highly suggested that each directory generates one single result, be it an executable, a library or a Python module.

This approach of course poses two immediate questions for the waf scripts: recursive execution and specification of dependencies.

For the recursive execution waf natively support the *recurse* function, that can be used in the configuration, options and build sections, to include other waf scripts. Using recurse waf will optimize tools that are eventually loaded multiple times and the dependencies between the different trees will also be matched. As an example given the structure:

```
fede@esopc ~/waf/example $ find . -name wscript
./wscript
./pkg2/wscript
./pkg2/exCqt5/wscript
./pkg2/exPyqt5/wscript
./pkg2/exProgC2/wscript
./pkg1/wscript
./pkg1/exProgLinkedC/wscript
./pkg1/exJava/wscript
./pkg1/exPython/wscript
./pkg1/exLibC/wscript
./pkg1/exProgC/wscript
```

The wscript at the top level would look like:

```
module_list = 'pkg1 pkg2'

def options(opt):
        # We recurse options in our submodules
        opt.recurse(module_list)

def configure(conf):
        # We recurse configurations in our submodules
        conf.recurse(module_list)

def build(bld):
        bld.recurse(module_list)
```

And following for example the in pkg2 directory will contain:

```
artifact_list = 'exProgC2 exPyqt5 exCqt5'

def options(opt):
        # We recurse options in our artifacts
        opt.recurse(artifact_list)


def configure(conf):
        # We recurse configurations in our artifacts
        conf.recurse(artifact_list)


def build(bld):
        bld.recurse(artifact_list)
```

The second interesting topic is how to specify a dependency explicitly. This is as easy as adding a *use*= indication where the usage is required, pointing the argument to the name of the file that is generated elsewhere. For example:

```
def options(opt):
       # We are using C++
opt.load('compiler_cxx')

def configure(conf):
       # We are using C++
        conf.load('compiler_cxx')

def build(bld):
# Define the main program. Note: it is using (use=) a library that is
# generated as another artifact someplace else in the build tree
       bld.program(source='src/exProgLinkedC.cpp', target='exProgLinkedC',
                    use='exLibC')
```

For the build of the program we require the usage of *exLibC,* a target generated in another directory with the following script (configuration and options have been omitted):

```
…

def build(bld):
        # Define the main program
        bld.shlib(source='src/exLibC.cpp', includes='src/', target='exLibC',
 export_includes='src/')
```

This target will create a shared library with the name *exLibC* (the operating system specific prefix and suffix will be managed by waf) and will also export automatically the include files from the *src/* directory. If another build rule in the same waf scope will add it using the *use=exLibC* option, then automatically the library will be linked and the needed includes imported.

In general, considering the base idea that each module creates just one product as it will be in the future ESO supplied additional layer, a generic pattern could be used:

```
bld.program(target='xyz', source=bld.path.ant_glob('src/*.cpp'),
                     includes=bld.path.ant_glob('src/includes/*.hpp'), … )
```

Where the target name would indeed be the module directory name itself with the eventual package name prepended.

3.9.3 Installing files

The waf build system supports natively installation of the built artefacts using the *install* directive to the waf command line executable. In a similar fashion also a very clean *uninstall* directive is present to exactly revert the installation process. Both of these directives rely on another optional command line option specifying the root prefix of the whole installation, namely *–destdir*.

If not specified otherwise, waf will try to install the artefacts accordingly to their types into a standard Unix directory tree starting from the root destination directory, therefore binaries in *bin/*, libraries in *lib/* and so on. Each build rule can have the installation position specified by adding an *install_path* parameter to the rule. Additionally to strip part of the path (for example the *src/* from the proposed directory structure) an additional parameter *install_from* is present.

```
bld(name='hello', features='py', source=bld.path.ant_glob('src/**/*.py'),
 install_path='$PREFIX/lib/python_modules/', install_from='src')
```

Additionally files can be installed which are not built as artefacts by waf using particular directives that are exemplified below:

```
        bld.install_files('${PREFIX}/include', ['a1.h', 'a2.h'])
        bld.install_as('${PREFIX}/dir/bar.png', 'foo.png')
        bld.symlink_as('${PREFIX}/lib/libfoo.so.1', 'libfoo.so.1.2.3')
```

It is important to notice that all the waf installation directives are just executed if waf is called with *install* or *uninstall* and not otherwise.

3.9.4 C++ example

The following is an example showing how to build a shared library and create a program that is a unit test based on the Google Test library. The Google Test library is defined in the configuration and then used as a normal dependency. The unit tests for C++ are handled by the standard waf_unit_test extra and are marked with the test feature. A shared library is created using the shlib directive, while a shared library can be created with a stlib directive and a program with the program directive)

The build script looks as follows:

```
def options(opt):
       # We are using C++ and Unit testing library
        opt.load('compiler_cxx waf_unit_test')


def configure(conf):
       # We are using C++ and Unit testing library
        conf.load('compiler_cxx waf_unit_test')

       # Define that the configuration stage requires Google test library
       conf.env.LIBPATH_GTEST = ['/opt/gtest/lib']
       conf.env.INCLUDES_GTEST = ['/opt/gtest/include']
       conf.check(compiler='cxx',lib='gtest',mandatory=True, use='GTEST')


def build(bld):
       # Define the main program
       bld.shlib(source='src/exLibC.cpp', includes='src/include',
target='exLibC', export_includes='src/include')

       # Define the unit test program (features='test')
       bld.program(features='test', source='test/unit_test.cpp src/exLibC.cpp',
includes=['src/include'] , use=['GTEST'], target='unit_test')
```

3.9.5 C++ and QT example

The following example compiles a simple C++ application using QT5 GUI libraries. The example takes care of generating user interface as needed from QT5 UI files. Waf also supports QT5 resources and language files generation.

```
def options(opt):
       # We are using C++ and QT5. Important: order matters!
       opt.load('compiler_cxx qt5')

def configure(conf):
       conf.load('compiler_cxx qt5')

def build(bld):
       # Define the main C++ program
       bld(features = 'qt5 cxx',
       uselib   = 'QT5WIDGETS QT5CORE',
       source   = 'src/m1uiCmdCpp.cpp src/m1uiCmdCppUi.cpp
 src/include/m1uiCmdCppUi.ui',
       target   = 'm1uiCmdCpp',
       includes = 'src/ src/include',
       defines  = 'WAF=1'
       )
```

It is important that for automatic generation of MOC files the WAF=1 is passed as from the example and then in the used C++ file the MOC is referenced so waf will generate it:

```
#if WAF
#include "include/m1uiCmdCppUi.moc"
#endif
```

3.9.6 Python example

Python modules and programs can be included in waf and as such they will be compiled to bytecode (to .pyc and .pyo by default for Python 2.x and just .pyc for Python 3.x) and will therefore be checked for formal correctness. Modules can be easily managed using the ant_glob to manage entire directory trees.

Unit tests are supported by the pytest extra which is based on the generic waf_unit_test module and therefore output of Python and C++ unit tests can be united in a common report. The parameter *pytest_source* points to the sources to be examined and *ut_str* to the command line to execute the tests. In the example it is both presented the use of tests embedded inside the main sources themselves, using doctest feature, or as truly separate source files in the *test/* subdirectory.

```
def configure(conf):
       # We are using Python and use Python unit tests
       conf.load('python pytest waf_unit_test')
       conf.check_python_version(minver=(2, 7, 4))

def options(opt):
       opt.load('python waf_unit_test')

def build(bld):
       # Example module
```

```
        bld(name='hello', features='py', source=bld.path.ant_glob('src/**/*.py'),
                    install_from='src')

        # Module tests using doctest tests embedded in the source code itself
        bld(features='pytest', use='hello', pytest_source=bld.path.ant_glob
('src/**/*.py'), ut_str='nosetests --with-doctest ${SRC}')

        # Module tests using standard separate unit test
        bld(features='pytest', use='hello', pytest_source=bld.path.ant_glob
('test/*.py'), ut_str='${PYTHON} -B -m unittest discover')
```

3.9.7 Python and QT5 example

Using the pyqt5 extra waf supports automatic generation of Python files from QT5 definitions (for UI, languages, resources). An example:

```
def options(opt):
        # Load also python to demonstrate mixed calls
        opt.load('python pyqt5')

def configure(conf):
        # Load also python to demonstrate mixed calls
        conf.load('python pyqt5')
        conf.check_python_version((3,4,0))

def build(bld):
        # Demostrates mixed usage of py and pyqt5 module, and tests also
        # install_path and install_from (since generated files go into build
        # it has to be reset inside the pyqt5 tool)
        bld(features="py pyqt5", source="src/sample.py src/firstgui.ui",
             install_path="${PREFIX}/play/", install_from="src/")
```

The pyqt5 waf module supports both PyQt5, PyQt4 and PySide2 bindings, being PyQt5 the default. To change the default value the options *--pyqt5-pyqt4* or *--pyqt5-pyside2* can be passed at waf configuration time.

3.9.8 Mixing Python and C++ and QT5

It is important to notice that natively in waf the pyqt5 and qt5 extras cannot be loaded at the same time as they will try to handle the same exact files based on their name extension (for example .ui files). In a single-artefact directory structure this also applies when recursively creating the build structure with the recurse command ad described. To overcome this restriction and additional extra shipped in the playground section of waf must be loaded as last extra named qtchainer:

```
…
def options(opt):
        # Load what needed for qt5 and pyqt5 and chainer as *last* so it
        # will chain to the proper one depending on feature
        opt.load('compiler_cxx qt5 python pyqt5')
        opt.load('qtchainer',
             tooldir='/usr/share/doc/waf-1.9.5/playground/qt5-and-pyqt5/qtchainer')

def configure(conf):
        conf.load('compiler_cxx qt5 python pyqt5 qtchainer')
        conf.check_python_version((3,4,0))
```

…

The specific feature to use should be then defined, for example:

```
…
bld(features="pyqt5", source="sampleRes.qrc")
…
```

Or

```
…
       bld(features = 'qt5 cxx cxxprogram', source="sampleRes2.qrc")
…
```

3.9.9 Java example

Using the java tool waf supports building java programs, preparing JAR archives and running Java Unit tests. So the basic setup in a wscript file is:

```
def options(opt):
       opt.load('java')

def configure(conf):
       conf.load('java')
       conf.check_java_class('java.io.FileOutputStream')
```

The check_java_class gives the possibility to check if a given class is available in the classpath. Additional classpaths can be defined in the configuration step as:

```
        conf.env.CLASSPATH_ADDNAME = ['aaaa.jar', 'bbbb.jar']
```

And then the additional name can be used as a standard *use* dependency in a java build step:

```
…
                use         = 'ADDNAME',
…
```

Following to just compile java sources for example:

```
       bld(features   = 'javac',
              srcdir     = 'src/',# folder containing the sources to compile
              outdir     = 'src', # where to output the classes (build directory)
              compat     = '1.8', # java compatibility version number
              sourcepath = ['src'],
              classpath  = ['.', '..'],
              name      = 'exJava-src',
       )
```

And to create a JAR archive:

```
       bld(features   = 'jar',
              basedir    = 'src', # folder with the classes and files to package
                                  # (must match outdir)
              destfile   = 'exJava.jar', # generated artifact name
              manifest   = 'src/exJava.Manifest',
              name      = 'exJava',
              use       = 'exJava-src',
       )
```

The operations can be also combined into a single step:

```
       bld(features   = 'javac jar',
              srcdir     = 'src/', # folder containing the sources to compile
              outdir     = 'src',  # where to output the classes (build directory)
              compat     = '1.8',  # java compatibility version number
              sourcepath = ['src'],
              classpath  = ['.', '..'],
              basedir    = 'src', # folder with the classes and files to package
# (must match outdir)
              destfile   = 'exJava.jar', # generated artifact name
              manifest   = 'src/exJava.Manifest',
       )
```

Unit testing can be done using the *javatest* waf extra:

```
def options(opt):
    …
        opt.load('java waf_unit_test javatest')

def configure(conf):
    …
        conf.load('java javatest')


        bld(features = 'javac javatest',
                srcdir     = 'test/',
                outdir     = 'test',
                sourcepath = ['test'],
                classpath  = [ 'src' ],
                basedir    = 'test',
                use = ['JAVATEST', 'mainprog'], # mainprog is the program being
 # tested in src/
                ut_str = '${JAVA} -cp ${CLASSPATH} ${JTRUNNER} ${SRC}',
                jtest_source = bld.path.ant_glob('test/*.xml'),
        )
```

Executing the test with code coverage enabled would require the unit test execution script to include JaCoCo as an agent :

```
ut_str          =          '${JAVA}          -cp          ${CLASSPATH}          -
javaagent:/opt/jacoco/lib/jacoco.jar=destfile=/some/path/jacoco.exec  ${JTRUNNER}
${SRC}',
```

The HTML report can be generated by hand or by generating a new waf task invoking the JaCoCo command line interface with something like:

```
${JAVA} -jar ${JACOCOCLI} report ${OUTDIR}/jacoco.exec --classfile ${CLASSFILE1} -
-classfile ${CLASSFILE2} … --classfile ${CLASSFILEN} --html ${OUTDIR}/jacoco --
sourcefiles ${SRCPATH}/src --sourcefiles ${SRCPATH}/test
```

3.9.10 Doxygen documentation generation

The waf build system supports Doxygen documentation generation using the doxygen extra. A Doxygen configuration file has to be supplied. For example:

```
def options(opt):
        # Doxygen extra
        opt.load('doxygen')

def configure(conf):
        # Doxygen extra
        conf.load('doxygen')
        if not conf.env.DOXYGEN:
                conf.fatal('doxygen is required, install it')
```

```
def build(bld):
        # Doxygen generation
        bld(features='doxygen', doxyfile='doxy.config',
                    install_path='${PREFIX}/doc')
```

In the Doxygen configuration file a few options are suggested to generate the recursive documentation without including in the indexes the waf generated build tree:

```
# The RECURSIVE tag can be used to turn specify whether or not subdirectories
# should be searched for input files as well. Possible values are YES and NO.
# If left blank NO is used.

RECURSIVE              = YES

# The EXCLUDE tag can be used to specify files and/or directories that should be
# excluded from the INPUT source files. This way you can easily exclude a
# subdirectory from a directory tree whose root is specified with the INPUT tag.
# Note that relative paths are relative to the directory from which doxygen is
# run.

EXCLUDE                = build/

# If the value of the INPUT tag contains directories, you can use the
# EXCLUDE_PATTERNS tag to specify one or more wildcard patterns to exclude
# certain files from those directories. Note that the wildcards are matched
# against the file with absolute path, so to exclude all test directories
# for example use the pattern */test/*

EXCLUDE_PATTERNS       = */.*/*
EXCLUDE_PATTERNS      += */build/*
```

Also setting the extensions of the files the user wants to generate the documentation for is a good idea if the defaults are not as desired:

```
# If the value of the INPUT tag contains directories, you can use the
# FILE_PATTERNS tag to specify one or more wildcard pattern (like *.cpp
# and *.h) to filter out the source-files in the directories. If left
# blank the following patterns are tested:
# *.c *.cc *.cxx *.cpp *.c++ *.d *.java *.ii *.ixx *.ipp *.i++ *.inl *.h *.hh
# *.hxx *.hpp *.h++ *.idl *.odl *.cs *.php *.php3 *.inc *.m *.mm *.dox *.py
# *.f90 *.f *.for *.vhd *.vhdl

FILE_PATTERNS          = *.c *.h *.cpp *.hpp *.py *.java *wscript
```

The latest pattern in the example, **wscript*, instructs Doxygen to pick up also the build scripts in the documentation. This is very useful to create the documentation grouping in the package build scripts and have them available in the modules, for example for our example structure in the *wscript* of *pkg1* we can define:

```
"""
@file
@brief Top level pkg1 build script
@defgroup pkg1 pkg1 module
"""
```

That will create the *pkg1* Doxygen group that can be then referenced in the modules to group documentation.

To fully support the build scripts in Doxygen documentation there are two more details to setup in the configuration file, namely tell Doxygen to treat such files as Python scripts and

to pass them through the Python documentation filter (which is doxypypy, as described in 3.6.4) for example as follows:

```
# Doxygen selects the parser to use depending on the extension of the files it
# parses. With this tag you can assign which parser to use for a given
# extension. Doxygen has a built-in mapping, but you can override or extend it
# using this tag. The format is ext=language, where ext is a file extension,
# and language is one of the parsers supported by doxygen: IDL, Java,
# Javascript, CSharp, C, C++, D, PHP, Objective-C, Python, Fortran, VHDL, C,
# C++. For instance to make doxygen treat .inc files as Fortran files (default
# is PHP), and .f files as C (default is Fortran), use: inc=Fortran f=C. Note
# that for custom extensions you also need to set FILE_PATTERNS otherwise the
# files are not read by doxygen.

EXTENSION_MAPPING      =        no_extension=Python

# The FILTER_PATTERNS tag can be used to specify filters on a per file pattern
# basis.
# Doxygen will compare the file name with each pattern and apply the
# filter if there is a match.
# The filters are a list of the form:
# pattern=filter (like *.cpp=my_cpp_filter). See INPUT_FILTER for further
# info on how filters are used. If FILTER_PATTERNS is empty or if
# non of the patterns match the file name, INPUT_FILTER is applied.


FILTER_PATTERNS        = *.py=py_filter *wscript=py_filter
```

3.10 ESO waf extension: wtools

wtools is a library that extends waf with helpers and implementation of a lot of default features. Specifically, it allows a user to declare a waf project and corresponding modules in a simplified way.

The official documentation of wtools is automatically generated for each ELT Linux Development Environment version. The latest version of the document can be found at this URL:

www.eso.org/~eeltmgr/documents/latest/wtools-docs/html/index.html

Since version 2.1.13 of ELT DevEnv, wtools documentation corresponding to the version of environment can be reached using the following format of URL:

www.eso.org/~eeltmgr/documents/<devenv-version>/wtools-docs/html/index.html

For example:

www.eso.org/~eeltmgr/documents/2.1.13/wtools-docs/html/index.html

3.11 Developer's IDE

The ELT Linux development environment officially supported IDE is Eclipse. Currently shipping the OXYGEN version, it is located under */opt/eclipse* and set by default in the user's path.

The installation is shipped with a couple of additional non-default plugins useful for the ELT development:

- pydev, Python development integration in Eclipse
- CDT, C/C++ development integration in Eclipse
- JDT, the Java Enterprise development integration in Eclipse
- SubVersive SVN plugin for Subversion integration in Eclipse
- LinuxTools Valgrind for valgrind memory checking integration in Eclipse
- LinuxTools GCOV plugin for C/C++ code coverage integration in Eclipse

3.11.1 Eclipse waf integration

Starting with version 1.9.12 the waf build system supports generation of Eclipse configuration files for C++, Python and Java.

To make sure the correct version is running either waf can be queried for the version or the RPM version can be checked, as in the following example:

```
(root) ELTjen00 root:~ 495 > waf --version
waf 1.9.12 (7641eac6c337687ada2e21655c0202345270cc22)
(root) ELTjen00 root:~ 496 > rpm -qa | grep waf
waf-1.9.12-1.noarch
```

Once verified that the waf version is correct, to generate the Eclipse configuration files starting from a wscript configuration and a configured instance (see 3.9.1) the command *waf eclipse* can be run:

```
fede@esopc ~/waf/example $ waf eclipse
Generating Eclipse CDT project files
'eclipse' finished successfully (0.021s)
```

The Eclipse configuration file will automatically contain the command line commands to launch the various phases of the build system and all the directories references needed to automatically resolve symbols in files will be added.

It is important to notice that IDE configuration files should **not** be included in the SVN repository but just used locally.

It is also important to notice that the Eclipse configuration generation can be rerun from waf at any time to regenerate dependencies. Of course any changes made by hand will be eventually lost.

3.12 Software versioning and revision control

The current versioning and revision control system for ELT is Apache Subversion (SVN). The CollabNet flavour of the command line tools to use SVN are provided in the ELT Linux development environment installed under */opt/CollabNet_Subversion/*. These tools are set by default in the user's PATH.

Additional information and guidelines on SVN usage can be found in AD2.

3.13 Integration tests

The integration tests framework in the ELT Linux development environment is Robot Framework (http://robotframework.org/) version 3.0.4-rc1. Robot Framework is a generic test automation framework for acceptance testing and acceptance test-driven development (ATDD). It has easy-to-use tabular test data syntax and it utilizes the keyword-driven testing approach. Its testing capabilities can be extended by test libraries implemented either with Python or Java, and users can create new higher-level keywords from existing ones using the same syntax that is used for creating test cases.

Robot Framework is shipped in the ELT Linux development environment in its Python flavour and installed in the Anaconda distribution described early. If the Anaconda distribution binary directory is in the path it can therefore be accessed launching the *pybot* (or *robot)* executable, for example:

```
fede@esopc ~/rtest $ robot test.rst
==============================================================================
Test
==============================================================================
Sample Code 1: launches ls -la /tmp/mustexistdir. Checks that rc 0... | FAIL |
2 != 0
------------------------------------------------------------------------------
Sample Code 2: launches ps ax and checs that ntpd is inside           | PASS |
------------------------------------------------------------------------------
Sample Code 3: start two sleeps processes, sleeps a bit, and check... | PASS |
------------------------------------------------------------------------------
Test                                                                  | FAIL |
3 critical tests, 2 passed, 1 failed
3 tests total, 2 passed, 1 failed
==============================================================================
Output:  /home/fede/rtest/output.xml
Log:     /home/fede/rtest/log.html
Report:  /home/fede/rtest/report.html
```

The execution will generate output and report files as stated at the end of the execution. The file passed on the command line is usually a file written in structured text. Writing such tests doesn't require the tester to have a knowledge of Python or Java. Documentation for the syntax can be found starting at http://robotframework.org/robotframework/#user-guide .

A very simple shell execution-based test that produces the before-mentioned output is presented below:

```
Robot Framework test script, done just to try and demonstrate Robot at work

.. code:: robotframework

    *** Settings ***
    Library           OperatingSystem
    Library           Process
```

```
    *** Test Cases ***
    Sample Code 1: launches ls -la /tmp/mustexistdir. Checks that rc 0 and
mustexistfile inside it. Logs all to robot log
       ${rc}        ${stdout}=       Run and Return RC and Output      ls -la
/tmp/mustexistdir
       Should Be Equal As Integers    ${rc}    0
       Should Contain    ${stdout}    mustexistfile
       Log                ${stdout}


    Sample Code 2: launches ps ax and checs that ntpd is inside
       ${result} =  Run Process         ps     ax
       Should Contain    ${result.stdout}    ntpd


    Sample Code 3: start two sleeps processes, sleeps a bit, and checks that one
is still there and one not. kills them. to fail put first sleep for ex to 10
       Start Process       sleep  3      alias=proc1
       Start Process       sleep  9      alias=proc2
       Sleep                      6
       Process Should Be Stopped  proc1
       Process Should Be Running  proc2
       Terminate All Processes           kill=true
```