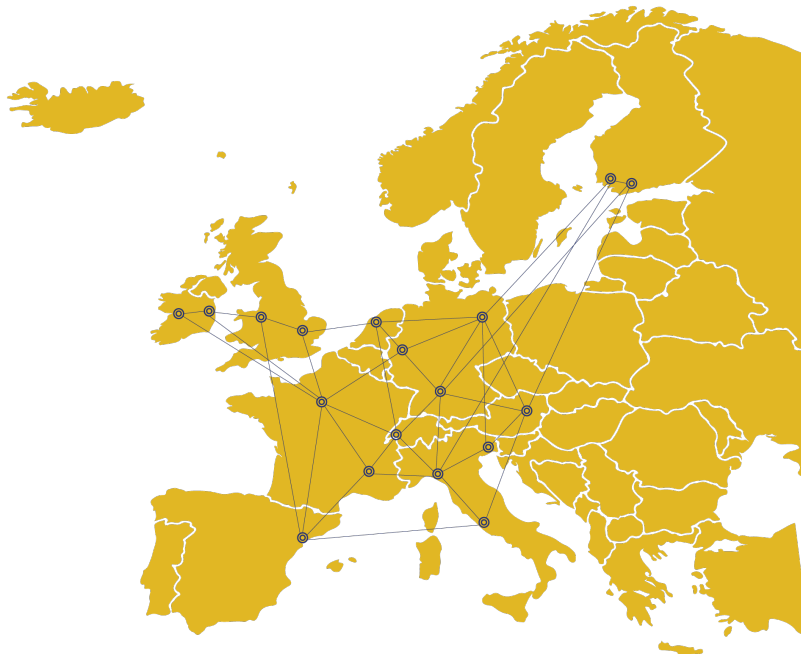




ESDW Technical Software Guidelines
Delivered in Month 6 – March 2016



E-CAM
The European Centre of Excellence for
Software, Training and Consultancy
in Simulation and Modelling



Funded by the European Union under grant agreement 676531

This report outlines the initial description of the technical framework within which the Extended Software Development Workshops (ESDWs) of E-CAM will operate.

The efforts of the software group within E-CAM target different aspects within the user community

- Users seeking general programming advice and guidelines
- Users seeking advice on methods, algorithms, applications, performance analysis and other such topics
- Users looking for a consistent building and testing environment for their applications
- Users wishing to add functionality to the E-CAM application and software libraries

The guidelines within this document are intended to provide a starting point for the technical framework for the ESDWs to target all aspects of this list. In it's online implementation, this information is intended to be a living document which evolves to reflect experience gained as the project progresses and thus this first iteration is not overly prescriptive.

Contents

1 Introduction	1
1.1 Contribution of the Software Team	1
2 General Programming Guidelines	2
2.1 The Unix Philosophy	2
2.2 Language-independent best practices	2
2.2.1 Versioning of Releases	3
2.3 Version Control	3
2.3.1 Branching Strategy	3
2.3.2 Contributing to a Library	4
2.4 Coding Guidelines and Code Review	4
2.4.1 Variable naming	4
2.4.2 Naming Conventions	5
2.4.3 Code Review and Checklist	5
2.5 Integrated Development Environment	6
3 Adding contributions to the E-CAM Application Libraries	7
3.1 ReStructured Text	7
3.2 EasyBuild	8
3.3 JUBE	8
3.4 Branching Strategy for Version Control	9
4 Contributing to the E-CAM Software Libraries	10
4.1 Regression Testing	11
4.2 Refactoring	11
4.2.1 Language Specific Standards	11
4.2.2 Documentation	12
4.2.3 Unit Testing	12
4.3 Branching Strategy for Version Control	13
5 Outlook	14

March 31, 2016

This deliverable was co-ordinated by Alan Ó Cais¹ (Forschungszentrum Jülich) on behalf of the E-CAM consortium with contributions from Dominic Tildesley (CECAM), Yann Pouillon (University of the Basque Country) and Matthieu Haefele (Maison de la Simulation).

¹a.ocais@fz-juelich.de

1 Introduction

This document is intended as a starting point for contributions from the Extended Software Development Workshops (ESDWs) of E-CAM. The scope of these workshops is expected to vary significantly depending on the research area, as well as over the project lifetime. For this reason the document itself is quite broad and introductory in nature. A more comprehensive version of the information in this document, with more detailed descriptions and references, will be made available through the project website and continuously updated.

The specific (sub-)topics of interest to a particular ESDW will be highlighted in advance to the participants of the ESDW. Part of content of the ESDWs will include any necessary training in the tools required to contribute to the E-CAM libraries (and these tools are highlighted in this document).

In general, E-CAM will produce two different type of libraries: an application library and a more traditional software library. The purpose of the application library, described in more detail in Section 3, is to act as a reference library for software developed and/or used by the research communities of E-CAM. In addition to this the library will provide "build recipes" for these software packages so that anyone using the library can reproduce the software (and it's dependency tree) in an identical context to that in which it is presented in the library. For this reason the application library will be "released" in a similar manner to software: each public update will come with a release number and there will be an archive of previous releases. The application library will also include, where feasible, regression tests for applications so that some measure of correctness, reproducibility and performance can be guaranteed.

An explanation of how to contribute to the more traditional software library is described in Section 4. The majority of code for this library is expected to come from the communities themselves. In addition to the general software guidelines of Section 2, this section provides specific contribution guidelines for the library, as well as outlining what a contribution work-flow might look like. The software library will have a continuous integration mechanism in place to streamline the inclusion of contributions from the community. Each release of the software library will also be integrated into the application library, and the application library itself will have also a continuous integration work-flow built in.

1.1 Contribution of the Software Team

The Software Team of E-CAM (consisting of the Software manager and 2 programmers) will provide the technical framework for contributions to the E-CAM libraries as well as training, guidance and consultancy in the tools required to use them effectively.

In addition, they will seek to provide a number of services including guidance and consultancy on topics such as performance analysis, HPC architectures, large scale I/O and other topics as well as hands-on assistance. While these services are not detailed in this document, they will be highlighted in the online platform that will incorporate this document.

2 General Programming Guidelines

To begin with, we highlight some general programming guidelines that we feel are worth reviewing before we get into the specific context of contributions to the E-CAM library.

2.1 The Unix Philosophy

When we develop software in a community we must consider that our work is not just for ourselves but is expected to be used, adapted and (even) improved by others in that community. Much of the Unix philosophy [Raymond, 2003] is worth considering when developing in such a context:

- Rule of Modularity: Write simple parts connected by clean interfaces.
- Rule of Clarity: Clarity is better than cleverness.
- Rule of Composition: Design programs to be connected to other programs.
- Rule of Separation: Separate policy from mechanism; separate interfaces from engines.
- Rule of Simplicity: Design for simplicity; add complexity only where you must.
- Rule of Parsimony: Write a big program only when it is clear by demonstration that nothing else will do.
- Rule of Transparency: Design for visibility to make inspection and debugging easier.
- Rule of Robustness: Robustness is the child of transparency and simplicity.
- Rule of Representation: Fold knowledge into data so program logic can be stupid and robust.
- Rule of Least Surprise: In interface design, always do the least surprising thing.
- Rule of Silence: When a program has nothing surprising to say, it should say nothing.
- Rule of Repair: When you must fail, fail noisily and as soon as possible.
- Rule of Economy: Programmer time is expensive; conserve it in preference to machine time.
- Rule of Generation: Avoid hand-hacking; write programs to write programs when you can.
- Rule of Optimization: Prototype before polishing. Get it working before you optimize it.
- Rule of Diversity: Distrust all claims for “one true way”.
- Rule of Extensibility: Design for the future, because it will be here sooner than you think.

If you're new to Unix, these principles are worth some meditation (and I would recommend reading the fuller descriptions of the [programming "rules" derived from the Unix philosophy](#)).

2.2 Language-independent best practices

The Unix philosophy applies to the creation of the software, but there are also some universally recommended best practices when it comes to the software development work-flow itself [Wikibooks, 2016]:

- Use a version management tool,
- Make a build in one step,
- Test suites to make sure what you are working on does what you think it should,
- Test-first programming: writing the test for a new line of code before writing that new line of code.

For our specific use case we will support [Git](#) (with repositories hosted on [GitHub](#)) as our version management tool, [CMake](#) and [Autotools](#) (complemented by [EasyBuild](#) for HPC environments) as our supported build environments and unit/regression testing through [Travis](#) continuous integration. Our continuous integration mechanism will function “*behind the scenes*” so will not be further described here, the others will be revisited where appropriate in the other sections of this document.

2.2.1 Versioning of Releases

Software versioning is the process of assigning either unique version names or unique version numbers to unique states of computer software. Adopting a logical system for releasing versions provides information to users that allows them, for example, to predict whether it is *safe* for them to move to a new release.

We follow the "[Semantic Versioning 2.0.0](#)" strategy. Given a version number x.y.z (MAJOR.MINOR.PATCH), increment the:

- MAJOR version x when you make incompatible API changes,
- MINOR version y when you add functionality in a backwards-compatible manner, and
- PATCH version z when you make backwards-compatible bug fixes.

The approach relies on bumping the correct component up at the right time. Therefore, determining which type of version you should be releasing is simple. If you are mostly fixing bugs, then this would be categorized as a patch, in which case you should bump z. If you are implementing new features in a backward-compatible way, then you will bump y because this is what's called a minor version. On the other hand, if you implement new stuff that is likely to break the existing API, you need to bump x because it is a major version.

Additional labels for pre-release and build meta-data are available as extensions to the MAJOR.MINOR.PATCH format. We begin at MAJOR version 0 and API changes may be allowed without incrementing it until we are ready to release a first stable version. Once we release MAJOR version 1 we intend to strictly follow the API policy.

2.3 Version Control

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. Many people's version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they're clever). This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you're in and accidentally write to the wrong file or copy over files you don't mean to.

[Git](#) is a widely used source code management system for software development. As with most other distributed version control systems, and unlike most client-server systems, every Git working directory is a full-fledged repository with complete history and full version-tracking capabilities, independent of network access or a central server.

We will use Git together with [GitHub](#) within E-CAM. Git will be our version control utility and GitHub will help us manage our work-flow by allowing us to create/label/follow/assign issues, review integration of new features, create milestones, etc.

There are many excellent guides to Git and GitHub so we will not go into any great detail in this document. For detailed information we refer you to [Software Carpentry's git lessons](#) and the [GitHub user guides](#).

2.3.1 Branching Strategy

We will follow a revision control strategy similar to [Vincent Driessen Branching strategy](#), focussed around Git. At the core, the development model is greatly inspired by existing models out there. The central repository (repo) holds two main branches with an infinite lifetime:

```
master
develop
```

The master branch at origin should be familiar to every Git user. Parallel to the master branch, another branch exists called develop. We consider origin/master to be the main branch where the source code of HEAD always reflects a production-ready state.

We consider origin/develop to be the main branch where the source code of HEAD always reflects a state with the latest delivered development changes for the next release. Some would call this the "integration branch".

When the source code in the develop branch reaches a stable point and is ready to be released, all of the changes should be merged back into master somehow and then tagged with a release number.

Therefore, each time when changes are merged back into master, this is a new production release by definition. We tend to be very strict at this, so that theoretically, we could use a Git hook script to automatically build and roll-out our software to our production servers every time there was a commit on master.

2.3.2 Contributing to a Library

We want people who contribute back to use a "branch-hack-pull request" cycle, the [GitHub Flow](#).. Our website will contain greater detail on the exact steps required but the basic concept is:

- Create your own copy of the repository on GitHub (fork)
- clone your repository to your machine
- Create a new branch for your feature. Feature branches should have descriptive names, like `animated-menu-items` or `issue-#1061`.
- Hack
- push your changes back to GitHub
- Create a [Pull Request on GitHub](#)² against the `develop` branch of the E-CAM library. This gives other developers an opportunity to review the changes before they become a part of the main codebase.

Code review is a major benefit of pull requests, but pull requests are actually designed to be a generic way to talk about code. You can think of pull requests as a discussion dedicated to a particular branch. This means that they can also be used much earlier in the development process. For example, if a developer needs help with a particular feature, all they have to do is file a pull request. Interested parties will be notified automatically, and they'll be able to see the question right next to the relevant commits.

2.4 Coding Guidelines and Code Review

While this document can be used as a starting point for coding best practices, it is really intended for those who want to contribute code to the E-CAM project, and describes the starting point for our coding standards and code review checklist (adapted from [[Khmer, 2016](#)]). We try to adopt best practices from existing projects with plenty of relative experience. [PyCogent](#) has useful coding guidelines that will act as a starting point for us.

Code, scripts, and documentation should have their spelling checked. All plain-text files should have line widths of 120 characters or less unless that is not supported for the particular file format. It is typical in many projects that a limit of 80 characters is given but we consider this excessively restrictive.

2.4.1 Variable naming

- Choose the name that people will most likely guess. Make it descriptive, but not too long: `curr_record` is better than `c`, or `curr`, or `current_genbank_record_from_database`.
- Good names are hard to find. Don't be afraid to change names except when they are part of interfaces that other people are also using. It may take some time working with the code to come up with reasonable names for everything: if you have unit tests, its easy to change them, especially with global search and replace.
- Use singular names for individual things, plural names for collections. For example, you'd expect `self.Name` to hold something like a single string, but `self.Names` to hold something that you could loop through like a list or dict. Sometimes the decision can be tricky: is `self.Index` an int holding a position, or a dict holding records keyed by name for easy lookup? If you find yourself wondering these things, the name should probably be changed to avoid the problem: try `self.Position` or `self.LookUp`.
- Don't make the type part of the name. You might want to change the implementation later. Use `Records` rather than `RecordDict` or `RecordList`, etc. Don't use Hungarian Notation either (i.e. where you prefix the name with the type).
- Make the name as precise as possible. If the variable is the name of the input file, call it `infile_name`, not `input` or `file` (which you shouldn't use anyway, since they're keywords), and not `infile` (because that looks like it should be a file object, not just its name).
- Use `result` to store the value that will be returned from a method or function. Use `data` for input in cases where the function or method acts on arbitrary data (e.g. sequence data, or a list of numbers, etc.) unless a more descriptive name is appropriate.

²Pull requests let you tell others about changes you've pushed to a GitHub repository. Once a pull request is sent, interested parties can review the set of changes, discuss potential modifications, and even push follow-up commits if necessary.

- One-letter variable names should only occur in math functions or as loop iterators with limited scope. Limited scope covers things like `for k in keys: print k`, where `k` survives only a line or two. Loop iterators should refer to the variable that they're looping through: `for k in keys, i in items, or for key in keys, item in items`. If the loop is long or there are several 1-letter variables active in the same scope, rename them.
- Limit your use of abbreviations. A few well-known abbreviations are OK, but you don't want to come back to your code in 6 months and have to figure out what `sptxck2` is. It's worth it to spend the extra time typing `species_taxon_check_2`, but that's still a horrible name: what's check number 1? Far better to go with something like `taxon_is_species_rank` that needs no explanation, especially if the variable is only used once or twice.

2.4.2 Naming Conventions

It is important to follow the naming conventions because they make it much easier to guess what a name refers to. In particular, it should be easy to guess what scope a name is defined in, what it refers to, whether it's OK to change its value, and whether its referent is callable. The following rules provide these distinctions.

- `lowercase_with_underscore` for modules and internal variables (including function/method parameters).
- `MixedCase` for classes and public properties, and for factory functions that act like additional constructors for a class.
- `mixedCaseExceptFirstWord` for public methods and functions.
- `_lowercase_with_leading_underscore` for private functions, methods, and properties.
- `__lowercase_with_two_leading_underscores` for private properties and functions that must not be overwritten by a subclass.
- `CAPS_WITH_UNDERSCORES` for named constants.

Underscores can be left out if the words read OK run together. `infile` and `outfile` rather than `in_file` and `out_file`; `infile_name` and `outfile_name` rather than `in_file_name` and `out_file_name` or `infilename` and `outfilename` (getting too long to read effortlessly).

2.4.3 Code Review and Checklist

Contributors also make good reviewers so we'd like you to be aware of what the review process looks like. We try to follow the best practices as described in "[11 Best Practices for Peer Code Review](#)". The most important to mention are:

- Pull requests should be small, the target is to review fewer than 400 lines of code at a time.
- Authors should document source code before the review.
- Embrace the subconscious implications of peer review. The knowledge that others will be examining their work naturally drives people to produce a better product.

Copy and paste the following into a pull request comment when it is ready for review (in our case that will be on [GitHub](#)). This lists helps ensure that we try to reach many of our targets in terms of

- [] Is it mergeable? (i.e., there should be no merge conflicts)
- [] Did it pass the tests? (Are there unit/regression tests? Do they pass?)
- [] If it introduces new functionality, is it tested? (Unit tests?)
- [] Is it well formatted? (typos, line length, brackets,...)
- [] Did it change any interfaces? Only additions are allowed without a major version increment. Changing file formats also requires a major version number increment.
- [] Is it documented in the [ChangeLog] (<http://en.wikipedia.org/wiki/Changelog#Format>)?
- [] Is the Copyright year up to date?

Note that after you submit the comment you can check and uncheck the individual boxes on the formatted comment in GitHub; no need to put x or y in the middle.

2.5 Integrated Development Environment

An Integrated Development Environment (IDE) is not a *necessary* development tool but it is a useful one. While we understand that people are familiar with a particular editor or work-flow, use of an IDE can help the development process flow more easily. An IDE is a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of a source code editor, build automation tools and a debugger. We are making this recommendation purely because IDEs can save you lot of **time**, particularly when you are contributing to someone else's code. There are many good reasons to use one:

- Integrated source control (this is major since the Git UI is frequently not intuitive)
- Quickly navigating to a type without needing to worry about namespace, project etc.
- Navigating to members by treating them as hyperlinks
- Auto-completion when you can't remember the names of all members by heart
- Automatic code generation
- Refactoring (major advantage)
- Warning-as-you-type (i.e. some errors don't even require a compile cycle)
- Hovering over something to see the documentation (provided by Doxygen)
- Keeping a view of files, errors/warnings/console/unit tests etc. and source code all on the screen at the same time in a useful way
- Ease of running unit tests from the same window
- Integrated debugging
- Navigating to where a compile-time error or run-time exception occurred directly from the error details.

If you are developing for a parallel environment for multiple HPC systems the [Eclipse IDE](#) even has a plugin specifically designed for this, the [Eclipse Parallel Tools Platform](#).

3 Adding contributions to the E-CAM Application Libraries

To ensure that E-CAM has maximum impact, we will provide a repository for software applications relevant to the E-CAM fields of research. The framework is an extension of the concept behind the [Electronic Structure Library](#). Each addition to the software library will consist of three components:

- Documentation (using ReStructured Text)
- A recipe for building the software and any dependencies it may have (using [EasyBuild](#))
- Regression Tests (optional but heavily encouraged, delivered by [JUBE](#))


The technical tools mentioned above will be supported by the software team of E-CAM, who will provide detailed templates for generating appropriate content.

The goal of this library is to provide a low-overhead mechanism to integrate existing applications and knowledge into E-CAM without forcing existing projects and codebases to adopt the licensing, standards and policies of E-CAM.

3.1 ReStructured Text

[ReStructuredText](#) is an easy-to-read, what-you-see-is-what-you-get plaintext markup syntax and parser system. It is useful for **in-line program documentation**, for quickly creating simple web pages, and for standalone documents. It also understands \LaTeX math formatting.

To give you an example of the syntax and result of reStructuredText, here's a simple example:

<pre> ***** My <code>rst</code> file ***** subtitle ##### subsubtitle ***** * This is a <code>bulleted</code> list. * It has two items, the second #. This is a numbered list. #. It has two items too. .. code-block:: python import <code>math</code> print 'import done' +-----+-----+-----+ +-----+-----+-----+ 1 2 3 +-----+-----+-----+ 6 5 4 +-----+-----+-----+ +-----+-----+-----+ .. image:: http://tinymce.com/h9bp735 :width: 200px :align: center :height: 100px :alt: alternate text To include simple Latex code in the text use the following code: <code>math:</code>`\alpha > \beta` </pre>	<h2 style="text-align: center;">My rst file</h2> <h3 style="text-align: center;">subtitle</h3> <h3 style="text-align: center;">subsubtitle</h3> <ul style="list-style-type: none"> • This is a bulleted list. • It has two items, the second <ol style="list-style-type: none"> 1. This is a numbered list. 2. It has two items too. <pre>import math print 'import done'</pre> <table style="margin-left: auto; margin-right: auto; border-collapse: collapse;"> <tr><td style="border-right: 1px solid black; padding: 2px 5px;">1</td><td style="border-right: 1px solid black; padding: 2px 5px;">2</td><td style="padding: 2px 5px;">3</td></tr> <tr><td style="border-right: 1px solid black; padding: 2px 5px;">6</td><td style="border-right: 1px solid black; padding: 2px 5px;">5</td><td style="padding: 2px 5px;">4</td></tr> </table>  <p>To include simple Latex code in the text use the following code:</p> $\alpha > \beta$	1	2	3	6	5	4
1	2	3					
6	5	4					

3.2 EasyBuild

EasyBuild is a software build and installation framework that allows you to manage (scientific) software on High Performance Computing (HPC) systems in an efficient way. It is motivated by the need for a tool that combines the following features:

- a *flexible framework* for building/installing (scientific) software
- fully automates software builds
- divert from the standard `configure/make/"make install"` with custom procedures
- allows for easily reproducing previous builds
- keep the software build recipes/specifications simple and human-readable
- supports co-existence of versions/builds via dedicated installation prefix and module files
- enables sharing with the HPC community
- automagic *dependency resolution*
- retain logs for traceability of the build processes

To illustrate the power of easybuild it is perhaps best to show a build recipe for an application commonly used within our target community:

```

1 name = 'GROMACS'
2 version = '5.0.5'
3 versionsuffix = '-hybrid'
4
5 homepage = 'http://www.gromacs.org'
6 description = """GROMACS is a versatile package to perform molecular dynamics,
7 i.e. simulate the Newtonian equations of motion for systems with hundreds to millions of particles."""
8
9 toolchain = {'name': 'intel', 'version': '2015a'}
10 toolchainopts = {'openmp': True, 'usempi': True}
11
12 source_urls = ['ftp://ftp.gromacs.org/pub/gromacs/']
13 sources = [SOURCELOWER_TAR_GZ]
14
15 builddependencies = [
16     ('CMake', '3.2.2'),
17     ('libxml2', '2.9.2')
18 ]
19
20 dependencies = [('Boost', '1.58.0', '-Python-2.7.9')]
21
22 moduleclass = 'bio'

```

Used with EasyBuild this recipe is enough to build GROMACS and all of the dependencies necessary to run it.

3.3 JUBE

Automating benchmarks is important for reproducibility and hence comparability between builds of software, which is the major goal. Furthermore, managing different combinations of parameters is error-prone and often results in significant amounts work especially if the parameter space gets large.

In order to alleviate these problems **JUBE** helps to perform and analyse benchmarks in a systematic way. It allows custom work flows to be adapted to new architectures.

For each benchmark application the benchmark data is written out in a certain format that enables JUBE to deduce the desired information. This data can be parsed by automatic pre- and post-processing scripts that draw information, and store it more densely for manual interpretation.

The JUBE benchmarking environment provides a script based framework to easily create benchmark sets, run those sets on different computer systems and evaluate the results.

Again, we illustrate things with an example:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <jube>
3   <benchmark name="hello_world" outpath="bench_run">

```

```

4   <comment>A simple hello world</comment>
5
6   <!-- Configuration -->
7   <parameterset name="hello_parameter">
8     <parameter name="hello_str">Hello World</parameter>
9   </parameterset>
10
11  <!-- Operation -->
12  <step name="say_hello">
13    <use>hello_parameter</use> <!-- use existing parameterset -->
14    <do>echo $hello_str</do> <!-- shell command -->
15  </step>
16 </benchmark>
17 </jube>

```

This configuration file with `jube run hello_world.xml` would result in the output:

```

#####
# benchmark: hello_world

A simple hello world
#####

Running workpackages (#=done, 0=wait):
##### ( 1/ 1)

stepname | all | open | wait | done
-----+-----+-----+-----
say_hello | 1 | 0 | 0 | 1

>>> Benchmark information and further useful commands:
>>>   id: 0
>>>  handle: bench_run
>>>   dir: bench_run/000000
>>> analyse: jube analyse bench_run --id 0
>>>  result: jube result bench_run --id 0
>>>   info: jube info bench_run --id 0
#####

```

3.4 Branching Strategy for Version Control

Please keep in mind that the strategy we will follow for versioning of the libraries here is identical to that of Subsection [2.3.1](#).

4 Contributing to the E-CAM Software Libraries

E-CAM is expected to generate software libraries in each of the targeted research areas. These libraries could include, for example:

- Interfacing libraries for intercommunication between various software packages.
- An optimised I/O subsystem for efficient I/O on HPC systems
- Algorithm implementations and interfaces for user-specified common cases
- Additional language binding for interfaces

While some of such items will come from the core software team of E-CAM, it is expected that the majority of the software within library will be actively contributed by the research communities themselves (of course with the assistance of the software team where necessary).

The software team will provide the build, performance analysis, continuous integration and testing environment and infrastructure for this effort. Coupled with the similar infrastructure provided by the E-CAM Application Libraries of Section 3, it can be leveraged as a continuous integration infrastructure by those aggressively adding to it.

The proposition of a module to include in the E-CAM libraries is under the responsibility of the scientist. Once proposed, a module needs to fulfill three requirements to pass to the next state and these requirements are under the responsibility of the scientists as well.

- Planned usage of the module in at least two different applications
- Proper documentation of the module
- Proper test suite for the module

This sounds like a lot of work of work on the scientist side, and it is, if neither documentation nor test suites are available. Once the module in a relatively mature state, the software team can take over and improves the module in term of interoperability, interface homogenisation and performance optimization.

A potential use-case is the case where the researcher has an existing application/method that they would like to share with the community. We will use this case to help structure a typical workflow of how to approach adding software to the E-CAM Software Libraries:

- Firstly, get your application included in the Application Libraries of E-CAM (3)

This will also greatly help in automating your workflow. The application library provides a method to build and test your application across multiple platforms. Integrating your application there is designed to be relatively straightforward with detailed examples available. It will give you better insight into how we do things and us better insight into what you intend to do and how to go about it.

- Regression testing

Part of adding your application to the Application Libraries includes the possibility of creating regressions tests. If you intend to contribute code to the Software Libraries of E-CAM, these tests are now essential. They help you to ensure that any changes you make are not impacting the performance of your application or introducing bugs.

- Refactoring

The Software Libraries of E-CAM are intended to be integrated into applications as libraries. When you contribute code to the E-CAM library your goal should be to extract the code from your application, get it included in the E-CAM libraries and then have your application link against this library instead of including the code directly. For this to work, you first need to *refactor* your code so that the methods are extracted from the main body of the code. It is during this process that you will begin to appreciate the value of having regressions tests. You should refactor keeping in mind the best practices of Section (2)

- Documentation

As you refactor, you should also document your functions, their parameters and your interfaces using the appropriate documentation tool. In-source documentation is a requirement of the E-CAM Software Libraries so it's best to start early and be consistent.

- Unit testing

Testing the functionality of individual functions or methods is highly desirable, since this greatly simplifies debugging at a later stage (if a regression test fails we need to quickly pinpoint what exactly is broken).

In the following subsections we give more specific details on the components of this workflow.

4.1 Regression Testing

Regression testing is a type of software testing that verifies that software that was previously developed and tested still performs correctly after it was changed or interfaced with other software. Changes may include software enhancements, patches, configuration changes, etc. During regression testing new software bugs (or *regressions*) may be uncovered [[Wikipedia, 2016](#)].

The purpose of regression testing is to ensure that changes such as those mentioned above have not introduced new faults. One of the main reasons for regression testing is to determine whether a change in one part of the software affects other parts of the software.

Common methods of regression testing include rerunning previously completed tests and checking whether program behaviour has changed and whether previously fixed faults have re-emerged. Regression testing can be performed to test a system efficiently by systematically selecting the appropriate minimum set of tests needed to adequately cover a particular change.

In our case we use JUBE as an additional framework for our regression tests. This allows us to also submit regression tests to a batch system and keep track of the impact on performance that changes may have.

4.2 Refactoring

Refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure.

Refactoring improves non-functional attributes of the software. Advantages include improved code readability and reduced complexity; these can improve source code maintainability and create a more expressive internal architecture or object model to improve extensibility. Typically, refactoring applies a series of standardised basic micro-refactorings, each of which is (usually) a tiny change in a computer program's source code that either preserves the behaviour of the software, or at least does not modify its conformance to functional requirements. Many development environments (see [Section 2.5](#)) provide automated support for performing the mechanical aspects of these basic refactorings. If done extremely well, code refactoring may also resolve hidden, dormant, or undiscovered computer bugs or vulnerabilities in the system by simplifying the underlying logic and eliminating unnecessary levels of complexity. If done poorly it may fail the requirement that external functionality not be changed, and/or introduce new bugs (and that's why we have regression tests!).

4.2.1 Language Specific Standards

The current understanding is that the most popular language in the community is Fortran, followed by C and C++. Therefore, all libraries developed within E-CAM must provide a full C implementation of the API, along with Fortran 2003 interfaces. Providing Python bindings is highly recommended but should be implemented only once the C and Fortran implementations are complete. Exception to this are allowed if the library is meant to be only used from a specific language (for example a Python for a post-processing method).

We are conscious that it may not be constructive to introduce excessive standards at such an early stage. Nevertheless, consider the following some guiding principles.

In order to ensure libraries can be used in as many existing and future codes as possible, we prefer libraries to be written in C and/or Fortran. Further, we strongly recommend that libraries provide both C and Fortran bindings. Most languages have good support for C foreign function interfaces and so C is an excellent lingua franca. Our standard for C is C11 for new code and C99 for legacy code, and our standard for Fortran is Fortran 2003 with the strong recommendation to upgrade legacy code. Fortran 2003 provides portable interoperability with C — please use this rather than the hacks required with earlier versions of Fortran.

For Python, [PEP 8](#) with 120 character lines is our standard. We will attempt to make our project single-source using the [Python 2/3 compatibility guidelines in the Python documentation](#).

For C++, any feature in C++11 is fine to use. We do our best to follow [Todd Hoff's coding standard](#), and "[One True Brace Style](#)" indentation and bracing. "One True Brace Style" adds brackets to unbracketed one line conditional statements. Opening brackets are broken from namespaces, classes, and function definitions. Brackets are attached to everything else including statements within a function, arrays, structs, and enums. Here's an example:

```

1 int Foo(bool isBar)
2 {
3     if (isFoo) {
4         bar();
5         return 1;
6     } else {
7         return 0;
8     }
9 }

```

For other languages we will define and follow standards as necessary.

4.2.2 Documentation

All code accepted into the codebase is expected to be documented. [Doxygen](#) is the de facto standard tool for generating documentation from annotated C++ sources, but it also supports other popular programming languages such as C, Python, and Fortran. Doxygen is our tool of choice for a number of reasons:

- It can generate an on-line documentation browser (in HTML) and/or an off-line reference manual (in \LaTeX) from a set of documented source files. The documentation is extracted directly from the sources, which makes it much easier to keep the documentation consistent with the source code.
- You can configure doxygen to extract the code structure from undocumented source files. This is very useful to quickly find your way in large source distributions. Doxygen can also visualize the relations between the various elements by means of include dependency graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically.
- You can also use doxygen for creating normal documentation

Doxygen can produce beautiful documentation if the code comments are written in its custom format. Thankfully, even if this is not the case it can still produce documentation that can be useful for understanding a mass (or mess) of code written by someone else. Below is an example of a C++ class that uses doxygen:

```

1 /*! A test class */
2 class Afterdoc_Test
3 {
4     public:
5     /** An enum type.
6     * The documentation block cannot be put after the enum!
7     */
8     enum EnumType
9     {
10     int EVal1,    /**< enum value 1 */
11     int EVal2    /**< enum value 2 */
12     };
13     void member(); /*!< a member function.
14
15     protected:
16     int value;    /*!< an integer value */
17 };

```

The [resulting documentation from this doxygen example file](#) can be seen at the linked url.

4.2.3 Unit Testing

Unit testing is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use. The goal of unit testing is to isolate each part of the program and show that the individual parts are correct. A unit test provides a strict, written contract that the piece of code must satisfy. As a result, it affords several benefits:

- Unit testing finds problems early in the development cycle. The process of writing a thorough set of tests forces the author to think through inputs, outputs, and error conditions, and thus more crisply define the unit's desired behaviour. In test-driven development (TDD), which is frequently used in both extreme programming and

scrum, unit tests are created before the code itself is written. When the tests pass, that code is considered complete.

- Unit testing allows the programmer to refactor code or upgrade system libraries at a later date, and make sure the module still works correctly (e.g., in regression testing). The procedure is to write test cases for all functions and methods so that whenever a change causes a fault, it can be quickly identified. Unit tests detect changes which may break a design contract.
- A unit test case, in and of itself, documents critical characteristics.
- When software is developed using a test-driven approach, the combination of writing the unit test to specify the interface plus the refactoring activities performed after the test is passing, may take the place of formal design. Each unit test can be seen as a design element specifying classes, methods, and observable behaviour.

4.3 Branching Strategy for Version Control

Please keep in mind that the strategy we will follow for versioning of the libraries here is identical to that of Subsection [2.3.1](#).

5 Outlook

This document is considered a first draft for the information that will be provided to contributors through the E-CAM website and repositories. The information stored there will be updated based on the project developments and supplemented by additional training material when it is available.

In particular the creation of this document has led to a number of interesting points being raised that would require significant elaboration and are probably best left to the collaborative environment of the online version. These topics include:

- test driven design;
- iterative approaches;
- continuous delivery;
- shortening feedback loops;
- deployment automation.

A discussion is also required on how E-CAM can encourage incremental adoption by the community, i.e. slowly and smoothly changing the community culture towards better practices.

Index of URLs referenced

Page 2

programming "rules" derived from the Unix philosophy ... <http://www.faqs.org/docs/artu/ch01s06.html>
Git ... <https://git-scm.com/>
GitHub ... <https://github.com/>
CMake ... <https://cmake.org/>
Autotools ... https://www.gnu.org/software/automake/manual/html_node/Autotools-Introduction.html#Autotools-Introduction
EasyBuild ... <http://easybuild.readthedocs.org/>
Travis ... <https://github.com/travis-ci>

Page 3

"Semantic Versioning 2.0.0" ... <http://semver.org/>
Git ... <https://git-scm.com/>
GitHub ... <https://github.com/>
Software Carpentry's git lessons ... <http://swcarpentry.github.io/git-novice/>
GitHub user guides ... <https://guides.github.com/>
Vincent Driessen Branching strategy ... <http://nvie.com/posts/a-successful-git-branching-model/>

Page 4

GitHub Flow ... <https://guides.github.com/introduction/flow/>
Pull Request on GitHub ... <https://help.github.com/articles/using-pull-requests/>
PyCogent ... http://pycogent.org/coding_guidelines.html

Page 5

"11 Best Practices for Peer Code Review" ... <http://smartbear.com/SmartBear/media/pdfs/WP-CC-11-Best-Practices.pdf>
GitHub ... <https://github.com/>

Page 6

Eclipse IDE ... <https://eclipse.org/ide/>
Eclipse Parallel Tools Platform ... <https://eclipse.org/ptp/>

Page 7

Electronic Structure Library ... <http://esl.cecam.org/>
EasyBuild ... <http://easybuild.readthedocs.org/>
JUBE ... http://www.fz-juelich.de/ias/jsc/EN/Expertise/Support/Software/JUBE/_node.html
ReStructuredText ... <http://docutils.sourceforge.net/docs/user/rst/quickref.html>

Page 8

EasyBuild ... <http://easybuild.readthedocs.org/en/latest/>
JUBE ... <https://apps.fz-juelich.de/jsc/jube/jube2/docu/index.html>

Page 11

PEP 8 ... <http://www.python.org/dev/peps/pep-0008/>
Python 2/3 compatibility guidelines in the Python documentation ... <https://docs.python.org/3/howto/pyporting.html>

Page 12

Todd Hoff's coding standard ... <http://www.possibility.com/Cpp/CppCodingStandard.html>
"One True Brace Style" ... <http://astyle.sourceforge.net/astyle.html>
Doxygen ... <http://www.stack.nl/~dimitri/doxygen/>
resulting documentation from this doxygen example file ... http://www.stack.nl/~dimitri/doxygen/manual/examples/afterdoc/html/class_test.html

References

- [Khmer, 2016] Khmer (2016). Coding guidelines and code review checklist.
<https://khmer.readthedocs.org/en/latest/dev/coding-guidelines-and-review.html>.
- [Raymond, 2003] Raymond, E. S. (2003). The art of unix programming.
<http://www.faqs.org/docs/artu/index.html>.
- [Wikibooks, 2016] Wikibooks (2016). Computer programming/standards and best practices.
https://en.wikibooks.org/wiki/Computer_Programming/Standards_and_Best_Practices.
- [Wikipedia, 2016] Wikipedia (2016). Regression testing.
https://en.wikipedia.org/wiki/Regression_testing.