Report on solutions for developing a common software infrastructure (Task 2 D6.2)

NMI-3 Workpackage 6 FP7/NMI3-II project number 283883
March 5th, 2014 - R. Leal and E. Farhi (with input from members of the workpackage)

Version 0.1

Abstract

This report documents the Task 2 of the Data Analysis Standards workpackage (NMI3-II/WP6). It focuses on aspects of the infrastructure currently used for neutron/muon software development. Gathering this information allows to derive recommendations for a common infrastructure layout that may be used as guide lines for a future common development framework.

Table of Contents

Introduction

In order to ease the development of a common software by a set of teams distributed

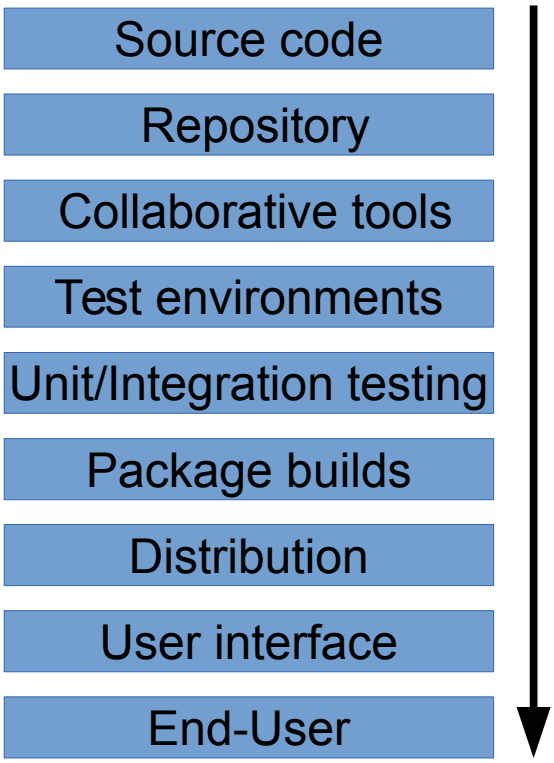
in different computing groups, it is advisable to define standards to organise the common work, as well as seeds for coding conventions.

In the following, we focus on different aspects of 'infrastructures', a terminology ranging from the low level coding work, to distribution of software, and specific uses of the software by the end-user.

A Infrastructure for code development

As stated in our previous Task 1 report available at <http://nmi3.eu/about-nmi3/networking/data-analysis-standards.html>, as well as other similar analyses, there exist currently many neutron/muon codes being developed by either single authors, or small development teams.

Many older codes have been developed by a single author, on their own local computer, without version control, and coherent distribution means. With time, some of these projects have adapted to current coding standards, and make use of 'modern' development tools. We list below elements which today are considered to help development.



Today, some of these elements in the software production work-flow exist as apparently free, cloud-based solutions, especially for code hosting and collaborative tools. Such tools do not need to be set-up and maintained by our neutron/muon computing groups, which may represent a net increase in productivity. On the other hand, all of these external tools hide commercial solutions, of variable longevity, whereas neutron/muon facilities are non-profit organisations with a better determined life-time.

The major component of the software production work-flow resides in its code repository. We recommend to provide and maintain a neutron/muon software repository at the community scale, for instance at the <http://www.neutronsources.org> website.

Let us review the key items of a software production work-flow.

Source code version control and repository

Today, most actively developed software use a version control system (Git, SVN, Mercurial). The Git version control system may lead to numerous branches which can be left un-merged for a long time. However, it is certainly more adapted to large projects.
Recommendation: favour Git and SVN over Mercurial, which is less adopted.

Source code repository

The code repositories associated to the version control system can either be local (single computer), facility/community based or rely on a 'free' cloud solution such as GitHub, BitBucket, SourceForge, Google Code, etc. If free services become 'paying' or otherwise disappear, it should be straightforward to transfer to another service.
Recommendation: favour facility/community based repositories to retain full code property.

Platform collaboration

In order to follow the code development, is it usual to employ collaborative development tools. These tools should provide a ticket tracking system (for bugs and feature requests), which should be used to follow the software development and release. In addition, communication channels are advisable to gather development messages between the developers, and these can be in form of a wiki, stored reference documents, and mailing lists. Other communication channels such as phone, Chats and Video calls are possible, but do not usually record the information flow.

Recommendation: free collaborative tools such as Trac and Redmine should be preferred over commercial products such as Jira. Trac is well suited for a single project, Redmine can handle a whole repository of projects. Ohloh provides a free code analysis tool.

Unit and Integration tests

The unit and integration tests are an essential step to identify bugs during the development, and a validation of the software integrity along versions.

Recommendation: write a test for every functionality in the software. The test should preferably use a JSON or XML output to be readable by e.g. Jenkins.

Test and Build servers

Using the project testing routines, it is possible to build automatically the software packages for easy distribution. Build servers should be available for e.g. Mac, Windows and Linux machines, with different flavours. One current, free solution is to make use of Jenkins, which provides both test and build features. In addition, the integrity of the automatically generated documentation, and the memory leaks can also be tested.

Recommendation: use Jenkins. Use a set of virtual machines for building the packages, e.g. as slaves for Jenkins.

Code review

It is advisable to review the code of the project by other members of the project, on a regular basis. This review must make sure the commits comply with the coding standards defined for the whole project, e.g. provide sensible solutions to design specifications, contain

enough comments, provide both a technical and user oriented documentation, keep control of complexity to lower maintenance, do not introduce new dependencies, etc.

A regular project review may be performed by independent programmers and scientists, in order to decide on the acceptance of implementations, as well as future orientations from design documents and feature requests. The different code components may also be rated by users on the basis of a 'like/dislike' electronic vote, to get a continuous satisfaction measurement.

Recommendation: commits may be reviewed by other programmers in the project. However, systematic reviews may introduce a high workload overhead, especially when the number of commits per month gets large.

Documentation (technical)

The produced code must contain technical documentation to ease the maintenance for future developers. This documentation should be processed and exposed to help the development of future contributions.

However, a technical documentation, even extensive, can not replace a proper user documentation which details the fundamental theory, as well as usage examples and tutorials.

Recommendation: use e.g. Doxygen and GraphViz to generate the technical/API documentation. Use a wiki for dynamic content editing.

B Infrastructure for distribution

Once the code has been produced, tested and packaged, the resulting software must be easily installable. Any difficulty in the installation will constitute a failure as the user then often turns to other solutions. So, the installation must be smooth and well documented.

As stated above a build server should produce distributable packages as part of the test process. Regularly, the project enters a stabilisation stage in which no new feature is implemented and the code is stabilised, before being released to the community.

Packages for e.g. Debian, RPM, SRC, Windows, MacOSX should be made available in both 32 and 64 bits systems. In the not too distant future, Android and iOS mobile systems may be considered as target systems.

The packages can be distributed via centralized repositories for e.g. Debian and RPM based systems, Windows and Mac OSX App Stores. Installation documentation is essential for an easy adoption of the software, which is intended to be widely used.

When releasing project versions, it is essential to devote enough time to a proper user documentation. This should not be fully, automatically generated, but should contain 'human written' text, describing in common words the underlying theory of a piece of code, with usage example, tutorials, … This documentation must be reviewed by external readers on a regular basis (possibly by electronic ranking with like buttons), as opposed to the technical documentation which is intended to guide the developer/contributor to the project.

Recommendations: distribute packages for 'modern' target systems, together with up-to-date installation instructions. Take special care to the User Documentation, which should be reviewed regularly.

C Infrastructure for user contributions and community

Once installed, when the software has a high usability, it is advisable to favour the emergence of a user community. Indeed, every user contribution demonstrates the acceptance of the software, and may also reduce the amount of work for the developers by transferring some of the high level tasks to the users.

Contributions may be of basically two kinds. Low level contributions use the project internals (API) to provide more advanced features. These must then be included at some point in the main development branch to become effective. Such contributions should follow the same standards as that of the project development team (including documentation and testing). Also, once transferred into the project, a minimal maintenance must be provided when the initial author can not ensure it. Higher level contributions, such as scripts, only use existing features of the project, and should be considered as examples. These require less maintenance than the low level contributions, but are often used by new users to acquire knowledge about the project.

The methodology for contributing must be clearly visible.

Low level contributions should follow the same rules as any coding activity in the project, that is e.g. authentication as a developer in the revision control infrastructure, in a private contribution branch. The final integration is then performed by the project developers, after checking (and possibly correcting) for coding standards.

A sensible way to propose contributions can use the tickets channel (as enhancements), which often requires authentication and also ensure that tickets are followed until proper integration. Any user can also be encouraged to simply send emails to the developer mailing list.

High level contributions such as scripts may be uploaded to a shared area, after authentication.

Recommendations: favour user contributions by providing dedicated upload channels. Clarify the coding standards (documentation) and maintenance responsibility of contributions over time.

D Infrastructure for user interfaces

The way software is rendered at the user level can rely both on a local installation, which is then limited in performance by the host computer, or on a cloud infrastructure, which requires network bandwidth and distant hardware credentials.

A cloud based service provides a uniform rendering to all users, whatever their system. However, the maintenance of the system must be ensured at all time, whereas a local installation is under the responsibility of the user. A cloud based service allows to install and upgrade software for all users at once, but this can also be achieved when providing proper software repositories to users. The connection to a cloud based service requires a user authentication, which can be kept for other services (storage, proposal and data access, …).

In order to ease the adoption of a software suite, it may be envisaged to design common templates for better visual identification. This way GUI's could follow the same specifications and keep a coherent appearance across functionalities.

Recommendations: design user interface templates when the software belongs to a coherent group of products (logo, menu layout, normalised interfaces and dialogues). Provide both local installation (packages) as well as distant clients (e.g. virtual machines via web browser).

Recommendations

We here list a set of recommendations that may be used to help the development and adoption of a

software.

- Use *Git* as version control software
- Set-up a *RedMine* repository for neutron/muon codes. Use *Ohloh* for code analysis.
- Write test routines as part of the project.
- Set-up a *Jenkins* server to report on testing and package build. Use virtual machines as slave build machines.
- Set-up a Wiki per project to expose technical documentation.
- Set-up package repositories for easy distribution, with installation instructions. Complement package distribution with cloud based services.
- Write and maintain a proper User documentation, including tutorials and extensive examples.
- Provide an attractive contribution mechanism to build a user community (upload area, tickets, mailing lists, code injection in code repositories ...).
- Define interface templates, to be adopted across a set of projects.

We recommend to install most of these services at a community level such as at <http://www.neutronsources.org>.

Some of these recommendations coincide with that listed in the PaN-data report D2.2 <http://wiki.pan-data.eu/imagesGHD/7/71/PaN-data-D2-2.pdf> '*Common policy framework on analysis software*'.

The resulting total infrastructure may seem overwhelming to individual users, and such a high directivity and control over the code production may hinder self initiative. In order to keep a fraction of freedom to help the emergence of new ideas alongside the official code production work flow, coders in a project may be given some freedom for other ideas of their own, using any tool they like, for say 10% of their time, in a philosophy close to that used at Google.