Agile Software Management for Research Environments *

Gema Ramírez-Sánchez
Prompsit Language Engineering, S.L.
www.prompsit.com
Campus UMH. Edifici Quórum III.
Av. de la Universitat, s/n. 03203. Elx (Alacant). Spain

19 April 2013

Contents

About this workshop

The materials of this workshop on "Agile Software Management for Research Environments" have been created by Prompsit Language Engineering, S.L., as part of the Abu-MaTran (Automatic Building of Machine Translation) project[1] funded by European Union Seventh Framework Programme FP7/2007-2013 under grant agreement number PIAP-GA-2012-324414.
Special thanks to Sergio Ortiz Rojas and Antonio Toral for their ideas and for the time devoted to proofread this guide.

Overview

This guide aims at being a starting point to define best practices in software management for researchers. It is divided in 8 sections addressing

[1] www.abumatran.eu

key topics such as: version control, regression tests, standards, documentation, packages and releases, licences and authorship and maintenance and management.

Every section has an introduction to the topic and aims at giving a practical state-of-the-art view of common practices and tools to ease the adhesion to these. Some hands-on and hands-up exercises are also provided.

1 Version control

1.1 Introduction

Version control (also known as revision control) is an essential practice in software management, specially for long-term development and collaborative projects.

Version control systems track and provide control over changes in source code. They can work as stand-alone applications or be embedded into other systems, e.g. document version control inside *LibreOffice*[2] word processor.

There is a big choice of visual and shell-oriented applications that provide version control, both private and free/open-source. You will find a comparison of the most notable options in Wikipedia[3].

To decide which is the most suitable option for you, a good reading is the introduction of the book *Version Control with Subversion*.[4]

1.2 What are the basic components of version control systems?

- a *repository* where tree-structured data under revision control are stored and from which you can make working copies (check-outs, clones, exports) and contribute modifications (check-ins, commits). Access can be:
  - *centralised*: data can be checked-out and checked-in with reference to a central repository

[2] `http://www.libreoffice.org/`

[3] `http://en.wikipedia.org/wiki/Comparison_of_revision_control_software`

[4] `http://svnbook.red-bean.com/en/1.7/svn.intro.whatis.html#svn.intro.righttool`

  - *distributed*: data can be checked-out and checked-in to any repository

- a *concurrency policy* to prevent conflicts derived from simultaneous editing. There are two main policies:
  - *lock model*: users ask and receive the right to lock a file from the master repository before editing it.
  - *merge model*: users can edit files freely, but are informed of possible conflicts before their changes are contributed to the repository. The version control system may merge changes if no conflict is found.

Two of the most widely used open-source version control systems are Subversion[5] (centralised, merge or lock model) and Git[6] (distributed, merge model). Let's see some basic tasks and commands for them:

TASK	**SUBVERSION**	**GIT**
Create a new local repository	**svnadmin create**	**git init**
Check out a repository	**svn checkout/co**	**git clone**
Add files	**svn add**	**git add**
Commit and push files	**svn commit -m " "**[a]	**git commit -a -m " "**[b] **git push origin master**[c]
See status	**svn status**	**git status**
See changes	**svn diff**	**git diff**
Delete files	**svn delete**	**git remove**
Move or rename files	**svn mv**	**git mv**
Update your local copy	**svn update/up**	**git pull**
Revert changes	**svn revert**	**git checkout –**
Create a branch	**svn copy svn+ssh**	**git checkout -b**
Tag particular revisions	**svn copy**	**git tag**

Table 1: *Subversion* and *Git* basic commands.

[a] to a remote repository
[b] to a local repository
[c] to a remote repository

[5] `http://subversion.apache.org/`
[6] `http://git-scm.com/`

Task 1. Trying Git. [15 min.] There is a user-friendly online version of *Git* that we will try out now.

- Go to http://try.github.com
- Follow the instructions and perform the 15 min. test.
- Write down any doubts and new commands you learn.

2 Regression tests

2.1 Introduction

Regression testing[7] aims at detecting bugs in a piece of software after a modification. Living software is being continuously modified and regression testing helps us to make sure that we've not lost any of the previous behaviour or functionalities performance and that we've extended or modified our programme as we were intending to.

There are many types of regression tests, but the most widely known are:[8]

- Functional tests (also called Quality Assessment – QA):
  - used to assess user-visible behaviour of a system
  - system black-box examination, what it does
  - written from the user's perspective
  - do not change frequently, only with major scheduled releases
  - example: in a hotel booking form, when a user clicks on the *arrival day field*, a list of possible `days of the week` must be displayed along with an `Any day` option to allow for flexible dates
- Unit tests:
  - used to assess the behaviour of individual pieces of a system source code
  - system white-box examination, how it does it

[7] `http://en.wikipedia.org/wiki/Regression_testing`

[8] Further reading: `http://stackoverflow.com/questions/2741832/unit-tests-vs-functional-testing`

- written from the developer's perspective
- change as frequently as the code grows
- example: function `validate_day_of_the_week()` should return `false` when the parameter is `January` and `true` when the parameter is `null`

2.2 Functional testing

Creating functional test sets should include:

- identification of functionalities that the software is expected to perform
- creation of input data based on functionatilies's specifications – test set
- determination of output based on functionalities's specifications – output set
- execution of the test case
- comparison of actual and expected outputs

2.3 Unit testing

One may define a *unit test* as the test of the smallest piece of code that can be run in isolation, e.g, a method, a class, a function, etc.

Ok, and now, what to test?

There is a mantra that says *one unit test per method...* but this can be very time consuming and unfocused.

Some rules of thumb:

- at least, write a unit test each time you discover a bug and cover the code paths that you believe are most likely to contain a bug
- write unit tests that check expected behaviour in "usual" and "not so usual" scenarios, like boundary or error conditions: test for success, for failure and for sanity

- make your unit tests create their own test data to execute against
- locate unit tests in separate files

Each unit test should:[9]

- set up the conditions for testing
- call the method being tested
- verify that the results are correct

Task 2. Unit testing a Roman numeral conversor. [10 min.]

Converting from Arabic to Roman numerals seems a straightforward task: for each Arabic numeral there is a Roman numeral, and a few rules apply... We are going to write a set of unit testing for a Roman numeral conversor.

- Think about 6 good test cases for a Roman numeral conversor: 2 for success, 2 for failure and 2 for sanity. Some examples have been given to help.
- Fill the form below with each unit test case and the expected output according to your knowledge.
- Execute the test cases using the conversor `http://www.thecalculatorsite.com/misc/romannumerals.php`.
- Assess that the expected output is the actual expected output or if the conversor fails.

Test case	**Expected output**	**Actual output**
II	2	2
9	IX	IX

Table 2: Test cases for a Roman numeral conversor.

[9] Further reading: `http://wiki.developerforce.com/page/How_to_Write_Good_Unit_Tests`

Most coding languages have their own mechanisms to make unit testing. Take a look, for example, at the way Python would have defined some of our test cases for the Roman numeral conversor: `http://www.diveintopython.net/unit_testing/testing_for_failure.html`

3 Standards

3.1 Introduction

Using standards is another key for success in software management.

What is a standard?

ISO/IEC Guide 2:1996, definition 3.2 defines a standard as:

A document established by consensus and approved by a recognized body that provides for common and repeated use, rules, guidelines or characteristics for activities or their results, aimed at the achievement of the optimum degree of order in a given context.

This is the definition of what is known as a *de jure* standard. Many *de jure* standards reach this status after being *de facto* standards: they are not approved but nonetheless they are widely used.

What do we want standards for?

According to Andrew S. Tanembaum *Standards define what is needed for interoperability: no more, no less.*, and this is the ultimate sense about using them: to boost interoperability.

Who defines *de jure* standards?

Many organizations do it, some of the most notable ones in the software domain are:

- ISO (International Standards Organization): founded in 1946 as a voluntary organization. It groups the standards organizations of 162 member countries including ANSI (U.S.), DIN, etc.
- IEEE (Institute of Electrical and Electronics Engineers): founded in 1963. It is a professional association with 400,000 members in more than 160 countries.

- W3C (World Wide Web Consortium): founded in 1994. It groups 379 member organisations.

3.2 What are the benefits of using standards?

Interoperability is the main benefit, but also:

- **ease of understanding**: others did the effort to document and explain standards for you and people may already know them
- **ease of learning**: barriers for sharing your software, having other people developing it, etc. are lowered, interconnection and interoperability is boosted, you motivate people!
- **robustness**: you are building your software upon solid foundations, enforcing of correctness
- **development speed up**: time devoted to create *ad hoc* ways to do things is saved, new developers will be able to dive into your code faster

Which standard to use? Although there is no answer to this question, Tanembaum provides a clue: *The nice thing about standards is that you have so many to choose from.*[10]

3.3 Some rules of thumb

- use established languages/technologies that fit your purposes: C/C++, JAVA for programming; HTML, PHP for web MySQL, MariaDB as databases, XML for data, etc.
- be modular: modules that do one thing and do it well, that are replaceable, that have a clear input/output way of communicating between them
- separate code from data: it will ease testing, maintenance, development, etc.

Task 3. Famous standards: who did what and when? [10 min.]

[10] From *Computer Networks*, 2nd ed., p. 254

In table 3 we've got some famous Internet and software standards. Go and discover what do they stand for, who developed them or which is the current release version to fill the the empty cells:

STANDARD ID	NAME	ORGANISATION & CURRENT VERSION
IEEE Std 1003.1-1988 ISO/IEC 9945		IECC
REC-soap12-part0-20070427	SOAP - Simple Object Access Protocol	
ISO/IEC 14882:1998		ISO / 2011
WD-xml-961114.html	XML (Extensible Markup Language)	
ISO/R 639:1967		ISO / 2002
TMX 1.1 DTD (April 23, 1999)	TMX (Translation Markup Language)	

Table 3: Some standards.

4 Documentation

4.1 Introduction

Documenting software is definetely a key for success for a software project. But, where to start?

Let's see it from a pragmatical point of view. We might define 3 phases for the life of a software project. These are not linear; actually, living software switches between them constantly. So, depending on where we are...

4.2 Which software documentation should we take the pleasure to write?

- Development phase:
  - requirements document: used throughout the development to define what a software does or will do.
  - architecture/design document: gives an overview of the software and of the environment where it will be set up.
  - technical documentation: for code, algorithms, interfaces, APIs, etc.

- Release phase: manuals for end-users/developers, administrators and support staff
- Dissemination phase: scientific papers, marketing material, a website!

It seems overwhemly, but conveniently divided in scheduled and defined phases and milestones, it is just another task in our schedule: well, a crucial task.

4.3 Documenting software is not a trivial task!

Researchers spend a long time writing dissertations, project ideas, papers, theses, presentations, etc. It is a compromise with science and a good way to make a project more attractive for collaborators and for funding.

In *Fifteen Thoughts and Tips on Writing Software Documentation*[11], Borja Sotomayor tell us about things that make software documentation a non trivial task:

- **it takes time**: well invested time. Always think about documentation when making a schedule and allocate a reasonable time for it.
- **it is a thorough exercise of empathy**: your target audience will be different in each case so you'll have to write for them. But you have also been playing these different roles. Just think about what do you usually like to find when reading other's code, when having to install a third-party software, when reviewing others' papers...
- **it provides no immediate gratification**: indeed, feedback will frequently address issues on your documentation... because people wants to use your software! Enjoy the moment and reuse users' issues to write a FAQ or to make a troubleshooting annex.

4.4 How to make documenting a more simple task?

- **You've got the power...** think about the effect you want to cause by writing documentation in each step of the way and go for it.

[11] `https://plus.google.com/106052512842507340767/posts/heH3pT2pczt`

- **...but you can also give the power to others**: some companies offer documentation writing services, specially for technical documentation and release-related documentation and if you come with a community of developers, they are most than often volunteers that help you to improve your documentation.
- **Don't panic about the empty document idea**: build on top of template documents. For example, if you are going to write a man/help page take a look a one of the historical ones and use it as a template, in other words, use *de facto* standards!!!
- **Automatise as much as possible**: there are documentation generators that automatically create software documentation for both developers and end-users directly from commented source code files.[12]

Task 4. Compare *Git* and *Subversion* documentation style: [10 min.]

- Go to *Subversion* home page at `http://subversion.apache.org/`. Which documentation do you find in it?
- After two or three clicks you will be able to arrive to the comprehensive *Version Control with Subversion* book at `http://svnbook.red-bean.com/`. Which layouts does it have? Do you have a preferred one?
- Go to *Git* home page at `http://git-scm.com/`. Which documentation do you find? How is it different from Subversion style?
- In both cases, do they cover the expected documentation for all 3 software phases?

5 Packages and releases

5.1 Introduction

Packaging and releasing software are among the most important non-development tasks in the life of a software project and the key for letting others know your software.[13]

[12] For a comparison of documentation generators see `http://en.wikipedia.org/wiki/Comparison_of_documentation_generators`

[13] Further reading: `http://producingoss.com/en/development-cycle.html`

The native form of software is source code. Software packages are always derived from source code to which they add building, installing and maintaining mechanisms.

There are tools that help covering every aspect of making software packages but, before that, we should decide what we are going to include in a package, that is, to unambiguously define a release.

5.2 Defining a release

An important step for packaging software is deciding what a release will contain. This decision can be made by a single authority (release owner) or by a voting system (release committee).

Releases are made after:

- known bugs have been fixed
- new features have been added
- new configuration options have been added

Once we've decided what to include in a release, some tasks apply:

- **release freezing or stabilizing**: usually by creating a release branch as a way to not interfere in the natural course of software development
- **release numbering**: a numbering scheme should be decided, it should not only reflect the ordering of releases, but possibly also how deep and which nature are the changes of a release
- **release scheduling**: you should plan and schedule releases depending on whether you follow a *Release early, release often*[14] or a *only polished, bug-free releases* strategy.

5.3 Packaging

When preparing a software package, there are some *de facto* standards that one may follow to make users or developers have a positive first interaction.

[14] From Eric Raymond book *Release Early, Release Often*. ISBN 1-56592-724-9.

These are related to:

- **format**: compressed files (*TAR*, *.tgz*, *.gz* for Unix/Mac OSX and *zip* for Windows).
- **name**: usually made by the software name, release version and compressed file extension, e.g. *mySoftware-2.3.1.tar.gz*. The use of capital letters can be coherent with the project name (MySQL, xpdf, GTK+) but lower-cased letters are preferred.
- **layout**: directory having the same name as package project name plus: README, INSTALL, COPYING or LICENSE, CHANGES or NEWS, source code tree, files for configuration and compilation, third-party software.
- **compilation and installation**: `./configure && make && make install` for programs written in C/C++ under Linux or `ant` for JAVA will please any developer. For Windows, the easiest way is to adhere to any of its native development environments (Visual Studio, VS.NET, etc.). For languages having their own ways to do things, e.g. Python, just use the standard method.
- **binary packages**: most users will install software directly from APT (.deb) for Debian GNU/Linux, RPM for RedHat GNU/Linux or .MSI or.exe files. for Windows.

5.4 Releasing

Once the package is ready, the process of making it publicly available starts. It is suggested to:

- apply regression tests and see that everything works fine
- ask others to download the package and do what users will do: decompress, build, compile and use
- sign the package: digital signature can be done, for example, using *GnuPG*[15] and is good to author the package
- compute MD5/SHA1 checksum: can be computed with several tools (e.g. the ones provided by *GnuPG*) and will help verifying the integrity of the package.

[15] http://www.gnupg.org/

- publish the package along with the digital signature and the MD5/SHA checksum code in a website
- anounce it! On website, distribution lists, through press releases, etc.

Task 5. Naming and numbering. [5 min.]

The last release of *ByThon* has been named *bython-fixes-3.5.0.tar.gz* and includes a few useful bugs solved. The previous version was *bython-3.4.9.tar.gz*.

What's wrong with the name and numbering? How should the new release be called? How could the new release be interpreted by a newcomer?

Task 6. Agree or disagree. [5 min.]

- Binary packages should be based on the last version of the source code.
- CHANGES file should list every change ever made to a project.
- To make a release, capture a tree at a moment in time, make a package, and release it with a proper version number.

6 Licenses and authorship

6.1 Introduction

When planning the distribution of a software: 1) you should state what you allow to do with it and 2) you may want to say who did it!

6.2 Licensing

In many countries, copyright laws automatically impose severe restrictions (all rights reserved) on the distribution and modification of released software. Changing this default behaviour is possible through a licence.

A licence defines the rights/freedoms that you want to transfer to any potential user or developer. You will have to think about it and choose a licence accordingly.

6.3 Alternatives to copyright

The most simple alternative to copyright is the public domain, which literally means uncopyrigthed. In some countries, you can also register evidence that you are the creator of a software.

Other alternatives emerged in the late 80's, when the Free Software Foundation created the concept of *copyleft*[16] and the GNU General Public License, a license tightening up the four freedoms of open-source (execute, examine and modify, distribute and improve and release a software).

Since then, at least 60 different licenses for software have been created[17] both for propietary and free/open source software.

For propietary software, the most common used licences are the end-user license agreement (EULA) licenses where certain rights regarding the software are reserved (number of copies and installations, terms of distribution, etc.).

For free/open-source software, the most general and used licenses are:

- **The GNU General Public License (GNU GPL for short):** a general copyleft license used by most GNU programs, and by more than half of all open-source software programs.
- **The BSD-like licenses:** simple, permissive non-copyleft open-source software licenses. They come in many versions (FreeBSD License, Modified BSD License), whose permissiveness varies. It is used by the Apache server and the BSD, Unix operating system.
- **Creative Commons Licenses:** a set of licenses created in principle for artistic works (documentation included). Depending on the particular restrictions applied to each license (attribution, non-commercial or share-alike) it may be considered as copyleft (attribution, share-alike use) or non-copyleft (any combination of the three restrictions different from attribution, share-alike).

Remember to put a LICENCE or COPYING file in the source tree of any software code you write.

[16] `www.gnu.org/copyleft/copyleft.html`

[17] See, for example, the list made by the Free Software Foundation at `http://www.gnu.org/licenses/license-list.html`

6.4 Authorship

Along with the license, when releasing a software, it is very important to clearly identify who is the author or authors of that software. This is usually made through the AUTHORS file which should also be present in the source tree of any software code.

6.4.1 Shared authorship

The AUTHORS file, in the case of collaborative development, can change to incorporate new authors. In these kind of environments, authorship is shared.

6.4.2 Who should appear as author?

The AUTHORS file can contain your name or the institution, company or organization name for which you work. The authorship policy might appear in your contract. Otherwise, you will have to discuss it internally.

Task 7. Copyleft: does it give freedom or impose limitations? [10 min.]

One of the main differences among them is their copyleft or non-copyleft nature; copyleft licenses force users to pass on their modifications, whereas non-copyleft licenses allow users to make their modifications private and distribute them as closed-source products. What do you think about it?

7 Sharing

7.1 Introduction

Research is more than often funded publicly and software developed for research environment is usally done to launch scientific experiments. By sharing our software you'll be:

- **giving something back to society**: a software that can turn into a useful end-user application, into an opportunity for service-centered bussines models or for further research

- **contributing to science**: by radically guaranteeing the reproducibility of experiments and by avoiding starting from scratch or reinventing
- **contributing to the creation of *de facto* standards**
- **creating an opportunity for collaboration**
- **increasing your work visibility, e.g. more citations**

All these positive outputs are part of the development model boosted by free/open-source software.

7.2 Is there an easy way for sharing software?

Many projects or software applications are hosted on personal websites but there are other web-hosting alternatives[18] which provide versions control, bug tracking, release management, mailing lists or wiki-based documentation.

Two of the most commonly used web-based hosting services for software projects are:

- **SourceForge.net**, launched in 1999, is the world's largest open-source software development website, hosting more than 320.000 projects, 3.4 million registered users and over 4,000,000 downloads a day.[19] Run on open-source software, it provides source-code browser, wikis, ticketing, mailing, forum, branching, etc.[20]
- **Github** launched in 2008, is a central point for development, distribution and maintenance of open-source (free) and private (paid) software. It hosts more than 6 million repositories and has over 3.5 million users.[21] It includes source-code browser, in-line editing, wikis, and ticketing.

Task 8. Benefits & disadvantages about self-hosting a software project. [10 min.]

[18] See a comparison of software projects web-hosting options at `http://en.wikipedia.org/wiki/Comparison_of_open_source_software_hosting_facilities`

[19] `https://sourceforge.net/apps/trac/sourceforge/wiki/What is SourceForge.net`

[20] `http://sourceforge.net`

[21] `https://github.com/blog/1470-five-years`

Self-hosting gives you the total control over your software project but most projects decide to use third-party hosting websites. Why do you think they do that? Which could be the main advantages and disadvantages of going self-hosted?

8 Maintenance and management

8.1 Introduction

Maintaing and managing a software project are crucial tasks for long-term succes. And, when successful, these tasks tend to last more than the development time itself!

Maintenance includes all the activities related to *"the modification of a software product after delivery to correct faults, to improve performance or other attributes"*.[22] Managing here refers to planning and organising the activities of a software project.

Due to the limited life of research projects, software maintenance or management are hardly ever addressed in research environments but, by releasing your software, you'll have access to a very valuable workforce: the community.

8.2 How to define a maintenance policy?

The minimum maintenance and management plan should at least address *corrective maintenance*,[23] i.e.: bug support.

This plan must clearly define:

- how will users request modifications or report problems: bug-tracker, mailing list, etc.
- what will be maintained
- how frequently maintenance activities are expected

[22] `http://en.wikipedia.org/wiki/Software_maintenance`

[23] According to ISO/IEC 14764, there are 4 categories of software maintenance: corrective, adaptive, perfective and preventive. Corrective maintenance is the reactive modification of a software product performed after delivery to correct discovered problems.

- who is the maintainer of what parts of the software
- who is the project manager: in charge of membership, releases, backups, etc.
- what is planned for future work: a brief TODO list is really useful

8.3 How to define a managing policy

A managing policy should, at least, define

- who is who in the project: who makes decisions, who can contribute, etc.
- which are the governing by-laws (e.g. in case of conflict or forks)
- the general lines of interest, activities and aims of the project

Task 9. Solving conflicts among contributors. [10 min.]

Different points of view on how to do things in a software project arise frequently. How would you cope with a conflict between two contributors if you were the final decision maker?