SGPEMv2 Developer Manual

for version 1.0, 8 September 2006

Filippo Paparella (ironpipp@gmail.com)
Paolo Santi (psanti@studenti.math.unipd.it)
Matteo Settenvini (matteo@member.fsf.org)
Marco Trevisan (evenjn@gmail.com)
Djina Verbanac (betalgez@yahoo.com)
Luca Vezzaro (lvezzaro@studenti.math.unipd.it)

Table of Contents

History **1**

1 Directory overview **2**

2 Writing documentation **3**

2.1 Formal documents and draft proposals 3
2.1.1 Introduction 3
2.1.2 Technical document format 3
2.2 Documenting code 4
2.2.1 C++ code 4
2.2.2 Python code 4
2.3 Reporting anomalies 5
2.3.1 Roles solving anomalies 7

3 Coding style **9**

3.1 Editors 9
3.2 Coding in C++ 9
3.2.1 C++ Coding Style 9
3.2.2 C++ Coding Conventions 11

4 Committing changes **14**

4.1 Introduction and goals 14
4.2 Repository layout 14
4.3 Basic svn usage 15
4.4 Gold rules on committing 15
4.5 How to write good log messages 16
4.6 Conflicts resolution 16

5 Using the Mailing List **17**

5.1 Introduction and goals 17
5.2 About the language, some useful conventions 17
5.3 Steer the wheel 17
5.4 A democratic discussion 18
5.5 Keeping the archives clean 18
5.6 Diving in technicalities 18

6 Writing your own plugins **20**

Appendix A License **21**

A.1 *Addendum*: How to use this License for your documents 27

Concept Index **28**

History

2006, February 21st, — Djina Verbanac
Added anomaly classes in the subsection anomaly solving process.

2006, February 7th, — Matteo Settenvini
Added subsection about the anomaly solving process. Fixed C++ code conventions about how to document classes, and the note about static non-const members in classes.

2006, February 2nd, — Luca Vezzaro
Updated coding style with a point on static non-POD objects.

2006, January 27th, — Matteo Settenvini, Djina Verbanac
Add section about conventions to be followed when documenting anomalies.

2006, January 27th, — Luca Vezzaro
Added entry on initialization lists and C-style comments in C++ Coding Style section.

2006, January 26th, — Matteo Settenvini
Added reference subsection about documenting code. Added decisional and communicative norms.

2006, January 26th, — Djina Verbanac
Added section about the writing of documentation

2006, January 24th, — Luca Vezzaro
Updated some code snippets whose style were inconsistent with our coding rules. Added the point on operator overloading to the coding conventions.

2005, December 26th, — Matteo Settenvini
Changed directory layout for '`src`'. Added '`swe/prototypes`' to repository layout. Added one more convention about C++ header files and their licensing.

2005, December 11th — Matteo Settenvini
Added sources' directory description and repository usage guidelines. Included full FDL license text.

2005, November 8th — Matteo Settenvini
First draft of this document

1 Directory overview

If you need to work on SGPEM sources, you'll probably be interested in understanding how this package directory structure is organized.

What follows is the tree you'll find after uncompressing the SGPEM tar archive.

'`config/`'	Files used by Autotools while configuring and compiling SGPEM.
'`data/`'	Various data SGPEM will use at runtime, like icons, images, XML DTDs, User Interface ('`*.ui`') definition files, and so on.
'`doc/`'	Inside this directory you'll find the User and Developer Manuals, like the one you're reading, ready to be compiled.
'`desktop/`'	The desktop menu entries for FreeDesktop compliant Desktop Environments.
'`distro/`'	Files used to prepare a package for a specific platform, maybe containing the installer data.
'`m4/`'	M4 macros used by Autoconf.
'`po/`'	Here are stored the Gettext PO catalogs. If you are a translator, you should first look here in order to localize SGPEM into your language.
'`src/`'	The source files of SGPEM.

2 Writing documentation

2.1 Formal documents and draft proposals

2.1.1 Introduction

For writing and editing technical documents we use a subsection of the free (as in speech) office suite *OpenOffice.org 2.0.0*, namely *OpenOffice.org Writer*.

We sporadically also use other parts of the suite, such as *OpenOffice.org Calc* for calculations and charts, and *OpenOffice.org Impress* for presentations.

2.1.2 Technical document format

All technical documents must respect the style and formats explained hereafter.

- *All the documents start with a Cover Page* : The **cover page** contains in the right corner the logo, name and address of the company. In the middle of the page there's the **title** of the specific technical document. Under the title goes the **acronym** of the project (SGPEMv2), followed by the **version** of the document.

  Versions use the format '`x.y`', where both '`x`' and '`y`' are integers. Minor changes to the document, like grammatical and spelling corrections comport the increment of '`y`', while structural or significant changes (for example, changing the way to resolve a specific problem) comport the increment of '`x`'.

  After the version number, it comes the **date** of the last modification made to the technical document. At the end of the cover page it should be reported the **status** of the document, either *Formal* or *Informal*, followed by one of the tags *External* or *Internal*.

  The **footer** of the cover page should contain the name of the **Author** of that technical document.

- The page after the cover page has to contain:
  1. **Distribution List**: to whom the document goes. It includes the Surname and Name of the person to be reached by the document, along with the role of that person.
  2. **History of changes**: where to trace the changes made to the document, including the date of the edit, the person who made the change, and the reason(s) of it.
  3. **List of Approved Versions**: here are traced all the approved versions of the document, reporting the date and the name of the person who approved that particular revision.

- An **Abstract** has to follow the List of Approved Versions: it's a short one-liner of what the document is about.

- On a new page, there's a **Table of Contents** that shows all the **Headings** in the file.

  If the document includes a lot of figures, then after the **Table of Contents** there should be an **Illustration Index**.

- After that, every document has to start with the **Introduction** section, where it will be described the purpose of the document and the purpose of the product, along with a list of used **References** to other documents or resources.

- Other parts of the document are specific to the technical document itself, and therefore should be managed by the editor. Mostly, the general structure for the more common documents can be found at http://www.math.unipd.it/~tullio/IS-1/2005/Progetto/Documenti.html (in Italian).

  In the '`docs/misc/`' directory of the Subversion repository, there's an OpenOffice.org **template** with some already defined styles you should inherit from. Its name is '`templatedocument.ott`'.
- At the end of every document there should be a **Glossary**. The Glossary is a file on its own (usually, '`docs/externals/Glossary.odt`' off the repository), so new entries must be inserted into this latter.

  Every technical document should "include" the afore mentioned file. Please refer to the *OpenOffice.org* manual about *Sections* to know how to do so. The Glossary will so be linked to the current document dinamically, and every change done on the Glossary will reflect in an automatic update in every referer.

  In this way all the technical documents are always able to have the latest up-to-date and synchronized revision of the Glossary.
- The **Header** of all pages, except the one on the cover page, contains:
  - in the left corner, the name of the company;
  - in the right corner, the chapter number and name.
- The **Footer** of every page, except on the cover page, contains:
  - in the left corner, the title of the document followed by the date of the last modification;
  - in the right corner it contains the number of the page due to the total number of pages.

To make it easier to abide these rules, please remember to inherit from the template document. It lays down a foundation for all the style guidelines mentioned above.

2.2 Documenting code

In order to generate documentation straight from the source code:

2.2.1 C++ code

Please refer to *Doxygen* manual (http://www.stack.nl/~dimitri/doxygen/manual.html) in order to learn how to use this automatic tool for extracting documentation from code.

Every function, class, class member, class method, and enumeration in code should be documented, less of trivial static global functions visible only to the current compilation unit.

In particular, for functions and methods, you should use `\param` and `\return` to describe respectively parameters and return value.

2.2.2 Python code

It is usually possible to annotate Python classes so that calling the `__doc__` method of an object returns its documentation string. Please refer to the Python manual at http://www.python.org/ to learn more of it.

However, if we want to generate good documentation straight from code, with the help of Doxygen, we should use the '`#`' documenting style. Please look at the end of `http://www.stack.nl/~dimitri/doxygen/docblocks.html` for an example.

2.3 Reporting anomalies

Note: sometimes we will refer to an anomaly with the less-formal term "*bug*".

Anomaly documentation is divided in two parts. The first part concerns the **Anomaly Investigation** and the second one the **Anomaly Resolution**. All informations about anomalies are held in the document named '`AnomalyRecords.odt`'. If you find an anomaly you should start a new page via a page-break, and create a header as follows:

ID Identification number of the anomaly, a unique integer

Summary A one-liner with a summary of the anomaly

Severity This field describes the impact of a bug.

'`Blocker`'	Blocks development and/or testing work
'`Critical`'	crashes, loss of data, severe memory leak
'`Major`'	major loss of function
'`Minor`'	minor loss of function, or other problem where easy workaround is present
'`Trivial`'	cosmetic problem like misspelled words or misaligned text
'`Enhancement`'	Request for enhancement

Anomaly Class

This field contains the name of the class to which the anomaly belongs to. There is no need to describe the function of this classes, because as you can see the names are self explanatory.

'`Extra (superfluous)`'
'`Missing`'

'`Ambiguous`'
'`Inefficient`'
'`Improvement needed`'
'`Not conforming to standards`'
'`Risk-prone`'
not wrong but there are known, safer, alternative methods

'`Incorrect`'
'`Not implementable`'
'`Safety`'

'`Not adequately documented`'

Reporter Name and email of the person who discovered the anomaly

Date of Identification
Timestamp at which the anomaly was identified

Milestone The milestone affected by the anomaly (e.g., m0.1)

Document Document (with version number) affected by the anomaly (e.g., Tecnical Specification, Product Definition, code, documentation, . . .).

Status Please choose one and only one of the following states.

These are used **before** reaching a resolution:

‘`UNCONFIRMED`’
This bug has recently been added to the database. Nobody has validated that this bug is true. The manager will usually confirm this bug, changing its state to ‘`NEW`’, but in case the bug existance is quite clear, any stakeholder can mark it as such.

The anomaly may be marked also ‘`RESOLVED`’ by the manager, if it is ‘`INVALID`’, ‘`DUPLICATE`’ or so on, but **not** ‘`RESOLVED FIXED`’.

‘`NEW`’ This bug has recently been added to the assignee’s list of bugs and must be processed. Bugs in this state may be accepted, and become ‘`ASSIGNED`’, passed on to someone else, and remain ‘`NEW`’, or it can be made also ‘`RESOLVED`’.

‘`ASSIGNED`’
This bug is not yet resolved, but is assigned to the proper person. From here bugs can be given to another person and become ‘`ASSIGNED`’ with a different assignee, or resolved and become ‘`RESOLVED`’.

‘`REOPENED`’
This bug was once resolved, but the resolution was deemed incorrect. For example, a bug is ‘`REOPENED`’ when more information shows up and the bug is now reproducible. From here bugs are either marked ‘`ASSIGNED`’ or ‘`RESOLVED`’.

Instead, **after** a resolution, the state should change to one of:

‘`RESOLVED`’
A resolution has been taken, and it is awaiting verification by QA. From here bugs are either re-opened and become ‘`REOPENED`’, are marked ‘`VERIFIED`’, or are closed for good and marked ‘`CLOSED`’.

‘`VERIFIED`’
QA has looked at the bug and the resolution and agrees that the appropriate resolution has been taken. Bugs remain in this state until the product they were reported against actually ships, at which point they become ‘`CLOSED`’.

‘`CLOSED`’ The bug is considered dead, the resolution is correct. Any zombie bugs who choose to walk the earth again must do so by becoming ‘`REOPENED`’.

Assigned to
Who's working to resolve this anomaly, name and email. Usually, it's initially left empty.

Description and comments
Latin creativity at work – text describing the reasons for assuming the presence of an anomaly. Please be as clear as possible. Unfortunately, giving guidelines in this sense is quite difficult . . .

As for follow-ups, always start with the name of the editor, email and the timestamp of the comment, assigning a numerical ID to it. An example of a comment would be:

```
-- Comment #1, Slartibartfast <slarti@go.eu>, 2006 Jan 27th - 16:13 --
Okay, I'll go and fix the time-space continuum on last Friday 17th.
Just give me the time to put the gas in my spaceship, will you?
```

Important: please report it clearly in a new comment if you change any field of the anomaly header.

When the anomaly is resolved, a footer is added to it, with:

Date Timestamp of the resolution date

Resolution
Choose one and only one of these tags. Please remember to change bug state accordingly and to report you've done so in a comment.

'`FIXED`' A fix for this bug is checked into the tree and tested.

'`INVALID`' The problem described is not a bug.

'`WONTFIX`' The problem described is a bug which will never be fixed.

'`DUPLICATE`'
The problem is a duplicate of an existing bug. Marking a bug duplicate requires the bug# of the duplicating bug and will at least put that bug number in the description field.

'`WORKSFORME`'
All attempts at reproducing this bug were futile, and reading the code produces no clues as to why the described behavior would occur. If more information appears later, the bug can be reopened.

'`MOVED`' The problem was specific to a related product whose bugs are tracked in another bug database. The bug has been moved to that database.

Fixed by Usually it's the same person as the assignee.

2.3.1 Roles solving anomalies

When solving an anomaly, we mostly follow the pattern therein explained:

1. **Anybody**, having any role, can report an anomaly about the work done by someone else. It doesn't make a lot of sense reporting a bug about what you're doing: you just fix it and carry on.

It may be sensible to report problems about your work only if you discover them in a second moment, when you don't cover the same role anymore, so the assignee would be different by the reporter.

2. The discussion starts. Usually the verifier will check the anomaly exists, but if enough people confirm the bug, the **manager** can decide to mark the bug as '`NEW`' as well.
3. Once it is clear who's the responsible for fixing the anomaly, the **manager** assign the bug to them, marking it as '`ASSIGNED`'.
4. A fix/patch may come up from different sources, even by the reporter, but only the **assignee** can apply it to the tree and mark the bug as '`SOLVED`'. They take full responsability of the fix.
5. The **verifier** does the needed QA (Quality Assurance), and then either reopens the bug or marks it as '`VERIFIED`'.
6. When and only if the product ships, a '`VERIFIED`' anomaly may become '`CLOSED`', by hand of the **manager**.

3 Coding style

In this chapter we explore some self-imposed coding standards we tried to follow when coding SGPEM. If you plan to extend it in any way, you should conform to the guidelines explained thereafter.

3.1 Editors

IDEs are a bad choice, since usually they leave your directory dirty, and full of temporary or project files. Please avoid their use if not strictly necessary.

A good choice for an editor is GNU Emacs, but every other editor that both insert spaces instead of tabulation characters and has a good Unicode support will do.

Your files should be in “UNIX mode”; that is, only a char is used for a newline. On DOS-based systems, usually two chars are employed: the newline char and the carriage return one. Failure to check your text files are correctly saved wastes space and others’ patience, so please take care.

This command usually fixes the problem (*note*: run it as it is only if no binary files are present in the current directory!):

```
for i in *; do tr -d '\r' < $i > $i.d; mv $i{.d,}; done
```

Using spaces instead of tabs in indentation is useful to ensure that your file will be correctly shown if another developer on another machine opens it with Emacs, Vim, Notepad, or what else he likes. A good idea is to use an editor which substitutes the ⟨TAB⟩ character with spaces. Most UNIX editors indent text files cleverly. The tab size is set to 2, the Emacs default.

You may want to set `indent-tabs-mode` to `nil` in your ‘`$HOME/.emacs`’ initialization file. It’s an option you can find via ⟨M-x⟩ `customize-group fill`.

GNU Emacs has another nice property: it can automatically indent code in a whole region for you, with *`M-x indent-region`*. Moreover, if you can get accustomed to it, you can activate the automatic indentation while in a programming mode, by typing *`C-c C-a`*. Experienced programmers find it saves quite a lot of time, but we guess it’s just a matter of taste.

3.2 Coding in C++

SGPEM is mostly written in C++, an Object Oriented language ideated by Bjarne Stroustrup and standardized in 1998 by ISO. Here are explained some guidelines you should keep well in mind if you want to contribute to this project.

3.2.1 C++ Coding Style

These are some notes on coding style, and things you should keep in mind whenever extending SGPEM source code. Patches to the source that don’t uniform to these guidelines are likely to be rejected and need rework. Coding styles are highly subjective and are often the cause of harsh holy wars. Here we try also to give a rationale of these choices, supporting our statements.

1. Left curly braces go on a newline, respect to the statement that comes before them. Right curly braces are to be put into a newline too. It may make you feel uneasy at

first, but this behaviour is preferable because it clearly let you identify code blocks. Moreover, it is observed that putting left curly braces on the same line of a statement isn't a rule you always follow: a lot of exceptions are raised by particular situations, like: "should my brace go on the same line after a class initialization list? and after a `for` loop declaration? what happens after namespaces declaration?" So it's best to stick to a well known practice and put it always on a newline. A lot of complex software projects follow this rule because it also increases manutenibility: keep in mind that you aren't writing code for yourself, but for others to read.

2. The return type for every function goes on a line, while the function name and its parameters go on the following, without any leading space. This makes easier to `grep` the source. For example, if you're searching for a declaration of `int foo::bar()` inside a large directory, grepping for: '`'/^foo::bar/g'`' will immediately pop out the correct result, whereas, if you didn't follow this rule, you would have searched for '`'/foo::bar/g'`', thus finding **all** recurrences of the function in the code, even function calls.
3. Use the following example to understand how we want you to space expressions:

   ```
   type var = exp1 OP (exp2 OP fun1(exp3));
   ```

   And for parameters, the following spacing is preferable:

   ```
   function(par1, par2 = value, ...)
   ```
4. Please define pointers like `type* var` instead of `type *var`.
5. Labels go indented on the same level of the containing code block. For example:

   ```
   switch(x)
   {
   case 1:
        // foo
   case 2:
        // bar
   default:
        // baz
   }
   ```

   Remember that also `public`, `protected` and `private` are labels.
6. Put incomplete class declarations before their interface, documenting it. For example:

   ```
   /** \brief I'm a useless class
    *
    *  This class is completely useless.
    */
   class C;
         // [...]

   class C
   {
   public:
         // [...]
   };
   ```

7. All header files of the libs and the engine follow a common model. Try to adhere to it by looking at existing headers. Document them fully, even if it is tedious, using a Doxygen-like syntax. The payback will be high, we assure you.
8. Class names are composed of UpperCamelCase words. Member functions are composed of lowercase words, separated by an underscore, since both the STL and Gtk– use this convention. Member data are lowercase words (can be separated by an underscore). Enums members are lowercase and they have a prefix that tells something about their function, e.g., in an enum named *signal*, "`signal_*;`". Macro names are written all in capital.
9. Private member object and function names always begin with an underscore. Public and protected ones don't.
10. Some (broken?) versions of autotools had problems with extensions other than '`.cc`' for C++ implementation files (e.g. automake didn't produce correct implicit rules for them). Consequently, in order to avoid problems, we require you to use the following extensions:
    - '`.cc`' : C++ implementation files
    - '`.hh`' : C++ header files
    - '`.tcc`' : Template implementation files

    You can also add to the end of your '`~/.emacs`' the line:

    ```
    (add-to-list 'auto-mode-alist (cons "\\.tcc\\'" 'c++-mode))
    ```

    to automatically associate the '`.tcc`' extension to the '`c++-mode`'.
11. Constructor initialization list uses the following format:

    ```
    C::C(T1 arg1, T2 arg2, ...) :
            m1(arg1), m2(arg2)...
    {
       //...
    }
    ```

3.2.2 C++ Coding Conventions

Some common rules you should keep in mind when writing new code:

1. Never use `std::cout/cerr` if you can do without them. Use `printf()` and `fprintf()` from `cstdio` instead,so that marking strings for `gettext` translation will be easier.
2. Don't use "`using`" directives into the global space, even in a '`.cc`', and neither in other namespaces. If you don't keep this in mind, you're buying to everybody a (big) problem. "`using`" makes sense at the beginning of a function to improve code readability, and you should be specific:
    1. "`using namespace std;`" is bad
    2. "`using std::string;`" is good
3. When treating long template names, remember that `typedef` (along with `typename` as needed) are your best friends, expecially at the beginning of class declarations or function definitions.
4. "*Syscalls*" are evil for portability. Thread calls can sometimes escape this rule (since they're quite different from system to system, and GNU/Linux ones are really good),

but remember that every UNIX/Win32/Solaris/etc. native call you use means extra-work somewhere in the near future. If a portable toolkit we're using provides the same functionality, it should be preferred to a system call.

5. You should start all your source files, both header and implementation ones, with a license notice, like this (no leading white lines):

```
// path/from/topsrcdir/file.ext - Copyright <year>, University
//                                of Padova, dept. of Pure and Applied
//                                Mathematics
//
// This file is part of SGPEMv2.
//
// This is free software; you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation; either version 2 of the License, or
// (at your option) any later version.
//
// SGPEMv2 is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with SGPEMv2; if not, write to the Free Software
// Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
```

   The style you use to comment this out obviously changes with the language you're employing.

6. Only exportable classes and functions inside a library should be marked with correct visibility attribute (usually a macro we defined as `SGP_DLLEXPORT`). All the others are marked with '`--visibility=hidden`' from the compiler, and aren't available outside the DSO (Dynamic Shared Object) they live in, so that they don't clutter the DSO namespace.

7. When you do something, remember to update the *ChangeLog*. This is essential. More on this on Chapter 4 [Committing changes], page 14.

8. Remember macros for inclusion at the beginning of header files, as well in template implementation files. In the latter case, you may want to include the template implementation files in the corresponding header files. An example of a correct inclusion directive is:

```
#ifndef HELLO_WORLD_HH
#define HELLO_WORLD_HH 1

// interface definitions

#endif
```

9. Please follow this order when declaring a class interface:
   1. Incomplete declarations of nested classes and friend functions / classes declarations.
   2. Typedefs
   3. Enums

4. Non-static member functions
5. Static member functions
6. Static constant data members
7. Variable data members

Static non-const public data members shouldn't exist. Nested classes go declared **outside** their containing class. For example:

```
class C;

class C
{
        class D;
        // ...
};

class C::D
{
        // ...
};
```

The order for visibility should be: `public`, then `protected`, then `private`.

10. Use operator overloading with care, only define overloaded operators if the use of that operator is a natural way to perform a particular operation on the object(s). In case operator overloading is considered the best choice, a lot of operators should still be avoided due to their profound integration with the language, and their tendency to lead to ugly bugs. Some examples of these operators are: the casting operator, `operator delete`, `operator new`, `operator ^`.
11. The C++ standard library is your best friend. For example, if you need to allocate some temporary variable on the heap, use an `auto_ptr<>` that does the right thing even when an exception is raised, and that respects RAII (Resource Acquisition Is Initialization).

    Also using extensively algorithms like `for_each` and `copy` greatly helps.
12. Use `glib::ustring` in place of `std::string` as often as possible, to ensure Unicode support.
13. If you need a smart pointer, be sure to check out `glib::RefPtr<>`.
14. "C-style" comments are useful but are also problematic when you need a fast way to exclude code from execution. Since this kind of comments cannot nest, C-style comments cannot be used to exclude code already commented with these old-way comments. For this reason there is no need to complicate our lifes with two styles of comments, the C++ comment (//) is more than enough, and typing isn't a problem since most editors support batch-commenting for multiple lines of code.
15. Never use global static variables of non-POD (Plain Old Data) type. The reason why not doing this is fundamental, and it is well described, along with a possible alternative here: `http://www.parashift.com/c++-faq-lite/ctors.html#faq-10.12`.

4 Committing changes

SGPEM sources are held in a repository managed by Subversion. This is not an introduction on how to use this tool. For that, you should refer to its own manual, located at http://svnbook.red-bean.com/. Rather, it sets some "best practices" you ought to follow committing changes to the repository.

4.1 Introduction and goals

The Subversion repository, commonly referred to just as "repository", is the place where all the material produced within this project will live.

There is a strong need to maintain an history of development data, even on plain documentation, whether it is a proprietary file format describing an UML chart, or some lovely C source code.

Mantaining versioned files for everything makes developers more free to cut, modify, hack, and revamp them, with the safety that older versions can always be fetched again.

This document describes some guidelines for maintaining the repository as clean as possible, by defining some restrictive rules that developers must respect in order to avoid abusing of the customizability this tool offers.

4.2 Repository layout

The layout you'll find inside the repository will be:

'`swe/branches`'

> This is the same as tags, except that commits are allowed inside the branch. Please refer to common development models to decide what should or should not be done inside a branch. Note that branching isn't something everybody should do: it should be agreed together with the project administrator.
>
> The format of a branch is:
>
> `<version_number>-r<revision_number>--<branch_name>`
>
> Example:
>
> `1.2-r164--guirestyle`

'`swe/docs`'

> (*subdirectories*: '`internals`', '`externals`', '`manuals`', '`misc`')
>
> Contains all drafts intended for the developers. This directory doesn't support tagging and branching because drafts has "eternal" life. If needs arise, they'll rather need to be renamed appending their version to their filename (-01, -02, etc.).

'`swe/prototypes`'

> A number of explorative prototypes we've set up to assess availability and workingness of needed technologies.

'`swe/tags`'

> It contains copies of the '`trunk/`' directory. Note that tagging the trunk directory reflects in a double space only for your local working copy, while it is

a $O(1)$ operation for the server. Changes and commits are NOT allowed here. Please note that tagging must be agreeded with project administrator.

The format of a tag is:

```
<version_number>
```

Example:

```
1.0
```

'`swe/trunk`'
: (*subdirectories*: '`doc`', '`src`', . . .)
This is the main development area where source files are held. Usually, official releases spin off the trunk. For a list of the directories layed out therein, please refer to Chapter 1 [Directory overview], page 2.

4.3 Basic svn usage

In your daily use of the repository you will mostly need to know a very small subgroup of svn commands. Other ones are usually for fine tuning operations (like setting file binary flags, keyword expansion, managing the repository tree, etc.), which are tasks carried out by the repository administrator.

Here there's a quick reference about such commands:

'`svn checkout` *`http://svn.thgnet.it/swe/drafts`*'
: Downloads a copy of the current '`drafts/`' directory contents. Checking out the root repository dir ('`swe/`') may result, in near future, to a **big** download, as branch and tags will be stored inside the root directory.

'`svn update` (*`short form`*: `svn up`)'
: Recursively updates the current directory to the latest revision. Use option '`-r N`' to update to a specific revision.

'`svn status`'
: Shows the status of **your** working copy against the repository.

'`svn diff`'
: Shows differences in unified diff format between your working copy and the base version of the current revision selected (usually it means the latest). If you want to compare two different revisions you can add a '`-r N:M`' parameter.

'`svn lock` *`filename`*'
: Locks a file so that everybody except the lock owner can't commit over it. This is particularly useful for binary files: you should always try to acquire a lock before starting editing.

4.4 Gold rules on committing

A versioning system, by definition, records everything that goes through it. So it may be a good idea not to commit garbage or make changes that will probably be rejected by the team. The repository mustn't be used as a temporary file storage system (so just don't use it to transfer files from work to home or vice-versa!).

When you commit something, it should be an acceptable piece of work. Of course, it can happen that later inspection demonstrates it's better to revert some changesets, but that's the purpose of having a centralized versioning system.

Avoid big commits altogether. A detailed description of your intentions should hit the mailing list *before* even starting to write such a patch. Then, committing your work must happen via **small incremental patches**.

Also please avoid making structural tree changes (creating, moving, removing, renaming directories) without asking first the repository administrator.

Everything can be reverted by `svn`, but that's not an excuse for sloppiness.

4.5 How to write good log messages

Be as descriptive as possible, concisely. If your changeset was discussed on the mailing list or at a meeting, make a clear reference to it.

Please be consistent with the message format. Use a clean english language, employing only abbreviations contained in the glossary document.

Always prepend a dash (−) for each changeset description, followed by a space. This will make clear, in case of line wrapping, what is part of the list and what is simply a new line.

Every sentence must end with a full stop. If a particular description is composed by several sub-descriptions, use a colon ('`:`'), and use a tab space to indent the inner list.

You can use only one level of nesting for lists. If you need more, you are probably making an oversized commit.

An example of the log format is the following:

```
- Change description 1.
- Change description 2:
 <tab>  - Part 1 of change description 2.
 <tab>  - Part 2 of change description 2.
- Change description 3. This particular change has a very long message
  describing this atomic commit.
```

4.6 Conflicts resolution

As in any other concurrent development system, conflicts may happen. this will be demanded to the official *Subversion Book (ch. 3 sect. 5.4)* for an explanation on how to handle them. We will just quote something to keep well in mind, however:

> In the end, it all comes down to one critical factor: user communication. When users communicate poorly, both syntactic and semantic conflicts increase. No system can force users to communicate perfectly, and no system can detect semantic conflicts. So there's no point in being lulled into a false promise that a locking system will somehow prevent conflicts; in practice, locking seems to inhibit productivity more than anything else.

Version Control with Subversion, a.k.a. "The Subversion Book"

5 Using the Mailing List

5.1 Introduction and goals

This mailing list (often referred from now on simply as “ML”) has been created with the specific aim of coordinating the effort of the members of our Software Engineering steering committee.

This chapter focuses on describing a set of guidelines of what should be considered best practices whenever writing to the ML, in order to avoid possible troubles and thornful outcomes.

This very chapter (an approved draft) you’re reading can be amended too. For more informations, please refer to the corresponding subsection, called “A democratic discussion”.

5.2 About the language, some useful conventions

The English language (either the UK or the US one, but the slangwords) is believed to be the best choice to deliver our internal drafts and user manuals, while the external ones are kept in Italian to help our Customer in his revision work.

The decision of internally using the English language descends from two considerations:

1) English is probably the most widely used language in the world (at least taking in account geographical extension). Making an effort to use it correctly during the development of this project is a good training ground for the future, and makes SGPEMv2 usable also by non-Italian speakers.

2) It encourages you to use a cleaner and simplier form, since you’re writing in a foreign language: you actually have to think about what you’re doing in a more careful way; not only to express your ideas, but also to express your ideas *in a way you can be understood*. Most people that don’t read again what they typed if writing in their mothertongue, usually look through an email composed in English at least twice, in order to check out for grammar mistakes. Therefore, it often makes easier to find also conceptual disasters.

Please avoid strong words or offensive language. They don’t help the discussion anyway, they represent a waste of bytes on the server, and at least some of us consider them to be childish behaviour.

If some of us ask you to explain what you meant, or correct your grammar, please don’t lose your temper or feel blue: we’re all here to learn.

As a side note, we encourage the use of the "singular they" as a neutral form. You can find a nice article on this by searching *Wikipedia*.

This doesn’t forbid you to use Italian for normal communications over the ML.

5.3 Steer the wheel

Most discussions start with a draft, like this. The talking then goes on amendating the document and, when no-one opposes it anymore, the draft is marked as an approved guideline, and everybody is meant to (as strictly as possible) conform to it.

Exceptions or revisions to guidelines due to a rework done by some other process pertain to the KA (Knowledge Area) of Process Improvement (see SWEBOK 9).

The committer could ask for some particular modalities in carrying out this procedure, although unlikely (usually, it’s an internal task).

This draft is hence subject to change in this aspect.

5.4 A democratic discussion

Of course, not everybody has to agree with everyone else on every subject. We’re just humans, after all; we’re entitled to our opinion. Hence, it may be necessary, at times, to set up a votation.

In a votation, if ‘`n`’ options are available, each voter has ‘`n*10`’ points expendable. They can thus weight their decision prioritizing one or more of the listed choices, and distributing these points as they wish. The highest scoring option wins the votation. In case of even scores, ballot can happen until a clear decision has been reached.

Usually a maximum time to amendate a draft or to vote is declared; if no further points to discuss are raised in such time, the document is marked as agreed. The minimum number of days for a votation / to amend a draft is set to three.

5.5 Keeping the archives clean

If, by writing your response, you’ve to comment on things that pertain to *different* KAs, or if the discussion requires to be `"branched"` over different subjects, please also start a new thread, change the subject line, and split your own letter, even if the answer to a question just takes two lines.

Not only this makes easier to search throughout the archives; it makes easier for others to reply only to those matters that concern they, tidily continuing the tree structure of threads.

5.6 Diving in technicalities

E-mails should be sent out text-only, not in HTML format. This makes them faster to download, store, manipulate, and it even saves us from kaki-on-pink kitsch backgrounds. Moreover, text-only mails are (almost always) monospaced; this allows the more aestethical-inclined of us to integrate ASCII-art in them, like handmade diagrams.

Virtually all email clients available let you choose how you want to send your messages to a certain address: if in HTML, plain-text or both. Please choose plain-text only.

Another important thing to keep in mind, is to set your encoding to UTF-8. This makes interoperability between different and exotic encodings possible by using a standard base. Yes, even between different operating systems, like the one called “98” and the one named “2000” *;-)*.

A simple discussion shouldn’t be marked in whatever way: it has just a meaningful subject. For example: *“hello, world!”* is a bad subject; it doesn’t tell much of the contents of the message, besides that probably it has some form of salute into it. Also, a line like *“i’ve a problem”* is a bad one: it could be a problem in installing an operating system, in finding a bug in a program, or in carrying out the rubbish and feeding the cat.

An example of a good subject is *“Possible buffer overflow in recursive-sort.cc:374”*. Also *“How can I stuff my cat with friskies?”* is a good one, albeit maybe a little bit off topic.

For documents like this, a subject line of the form *"(draft) <description>"* is a good choice. It makes it easily recognizable, and it's like automatically asking for people to amend.

Approved drafts are marked with a date, and are copied in a separate directory of the repository (tipically, a '`doc`' subdirectory). A quick TeXinfo rewrite can be performed if need arises, or if someone is willing to spend five minutes doing so.

Every approved document contains the full text correctly amended (whereas, during the discussion just snippets of it are changed by diff), and starts with a date and a progressive number in its header. The header should be in the form:

```
----------------------------------------

(approved draft) #xxxx, [<area>]
Initially submitted by: <Name Surname>
On date: <YYYY, Month DD>
Approved by:
     <Name Surname 1>
     <Name Surname 2>
     ...
Rejected by:
     <Name Surname 1>
     ...
Reason of refusal:
    <description of the reason>
Finally approved on date: <YYYY, Month DD>

----------------------------------------
```

Of course, the '`rejected by`' list should ideally be empty, or just listing '`nobody`'. The subject area is up to the writer to purpose, wisely. Different areas are instantiated as need arises. Please don't create a forest of areas; it's no use whatsoever.

Have fun, and ...

... happy hacking!

6 Writing your own plugins

TODO: write me.

Appendix A License

A.1 *Addendum*: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
  Copyright (C)  year  your name.
  Permission is granted to copy, distribute and/or modify this document
  under the terms of the GNU Free Documentation License, Version 1.2
  or any later version published by the Free Software Foundation;
  with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
  Texts.  A copy of the license is included in the section entitled ``GNU
  Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
    with the Invariant Sections being list their titles, with
    the Front-Cover Texts being list, and with the Back-Cover Texts
    being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Concept Index

A

anomaly . . . 5

B

bug . . . 5

C

c++ . . . 9
coding conventions . . . 11
coding style . . . 9
coding, style . . . 9
committing . . . 15
conflicts . . . 16

D

directory layout . . . 2, 14
documentation . . . 3
documenting C++ code . . . 4
documenting Python code . . . 4
draft proposals . . . 3

E

editors . . . 9
emacs . . . 9

F

FDL, GNU Free Documentation License . . . 21
formal documents . . . 3

H

history of changes . . . 1

L

locking files . . . 15
log messages . . . 16

M

mailing list . . . 17
merging . . . 16

R

repository . . . 14, 15

S

source code organization . . . 2
subversion . . . 14
svn . . . 15