Agile Documentation Tools

V Jornades Programari Lliure

Sergio Talens-Oliag
sto@iti.upv.es
Instituto Tecnológico de Informática

8 July 2006

1 Introduction

This article is about the importance of *documentation* on *software* related projects and the tools and systems that developers can use to be more productive when writing that *documentation*.

We will start by explaining what are we talking about when we use the term *documentation* on the context of software related projects and why we believe it is important; later we will move to the formats and tools problem, that is, we will talk about the systems used to write documentation, centering our discussion on the types of markup languages available and their advantages and problems.

After this general exposition we will move into our main topic, *agile documentation tools*. We will enumerate the features we want from a documentation system (i.e. we want *lightweigh markup formats* to be able to write fast) and will compare some of the available formats and tools that provide the desired characteristics, discussing their advantages and drawbacks.

Note that we will simply talk about *tools* and *formats* useful to write and manipulate documentation, not about what we should document nor when or how we should do it; that would be the subject of another kind of article, probably related to project development methodologies.

2 What is documentation?

When talking about software projects, the term *documentation* refers to the written text that accompanies the software produced by the project. This text can be of various types:

- *Architecture / design*: this kind of documents provide an overview of how the project is going to be developed and why it is going to be done that way; those documents are usually generated at the beginning of the project, but, to be useful, must be reviewed when things change during the project life cycle.

  The documents don't need to be very detailed, the idea is to have a high level description of the components that are going to be used, why they have been chosen, what is their expected functionality and what are the relationships between them. The design document can also include notes about how they should be implemented (data types, algorithms, etc.).

- *Technical*: documentation of code, algorithms, interfaces, APIs, etc. It is more detailed and usually has to be written while developing the code; generally the best approach is to keep it inside the source code, at least for the API documentation; there are two approaches to do that:

  - Write the source of the program using a format parseable by *documentation generators* that can be used with multiple languages (doxygen, robodoc or naturaldocs) or that are tied to specific languages (epydoc, javadoc or perlpod); those tools parse the source code and generate documents in one or more output formats (`html`, `pdf`, etc.) from the programming language constructs and *specially marked comments*.
  - Use the *literate programming* model, a technique first proposed by Donald Knuth with his `WEB` software system (the current version is CWEB, a rewrite of `WEB` for documenting C, C++ and Java programs instead of Pascal). On this system the source code is embedded inside descriptive text, rather than the reverse; the main idea is to regard a program as a communication to human beings rather than as a set of instructions to a computer. To generate compilable source the program code is extracted from the document by literate programming tools.

  Of course there are always some documents that should be in separate files like, for example, the project guidelines for developers; those guidelies can include documents explaining how is the project code organized, how to use the SCM system, how the code should be formatted and commented, etc.

- *End User*: Manuals for the end-user, system administrators and support staff. Usually written by specialized personnel, this kind of documentation usually has nothing to do with the source code, it simply describes how to to use the software produced by the project.

We are mostly interested on the first two types of documents, as those are the ones that should be written by computer science specialists (analysts, developers, system and network administrators, testers, etc.), but the tools and formats discused can also be used to write end user documentation.

Note that we will limit the scope of this article to formats and tools useful to write standalone documents, leaving the *documentation generators* and the *literate programming* tools for another time.

Anyway, from the writer point of view, *documentation generators* are not too different from the tools we will talk about, they tend to be very specialized but the markup formats they use are quite similar to the ones used on the systems we will be discussing.

Similarly, *literate programming* tools can also use *lightweigh markup languages* or more complex systems like **TeX**, but that is usually not the main problem when discussing the use of those systems, their main problem is that using them means changing the way developers work, and that is a totally different issue. The reader interested in the ideas and available tools to work on *literate programming*, there is a very interesting website devoted to the subject on http://www.literateprogramming.com/.

3 Why is documentation important?

There are a lot of valid reasons to write documentation on a software related project:

- It is a requirement of a commercial project where the clients want different parts of the system documented.

- To define a contract model that specifies how two different subsystems interact; contract models are usually needed when there are different groups working on different components of a system (i.e. when an application needs to intereact with external databases or legacy applications). When using a contract model the documentation has to be kept up to date to be useful (changes have to be documented).

- To support communication with an external groups. When working with people geographically disperse there needs to be a way to communicate them and shared documentation is often part of the solution, but always in combination with other channels like face-to-face discussions, teleconferencing, emailc or the use of collaborative tools.

- To think something through. Many people will write documentation to clarify things discussed in a group or simply to increase their own understanding of a problem. Usually putting ideas down on paper can help to solidify them and discover the problems they could have.

In fact, on the traditional development models documentation is quite important on all the phases of the software life cycle:

- It helps to define the problems we are trying to solve and to arrive to an agreement with the people that wants us to solve them.
- The requirements have to be writen somewhere, as that is the only way to have a reference when developing and known which ones are valid.
- Coders, testers and external groups need the documentation produced on the analysis, design and development phases, as that is the base of their work. In fact reallity says that they usually need to review and refine that documentation as they go, as requirements and expectations change during the life of the project, forcing us to iterate and repeat previous phases to adapt to the changes.
- People working on mainteinance also needs a good understanding of how the system has been designed and implemented, and that is usually taken from the documentation produced on the previous phases. Besides, depending on the kind of project, mainteinance people will also need to document the changes they make and the tasks they do or need to do on a regular basis.

That being said, keep in mind that our idea is to have and work with light documents, that is, the documentation should be short, terse and easy to understand by qualified individuals; the documentation has to serve the needs of new project developers, that is, to help someone not involved on the project to start working on it and also to keep a history of important events (requirement changes, design decisions, etc.) for the people working on it.

4 Markup languages

Almost all texts written on a computer use some kind of markup system to include information about the structure and characteristics of the text; this markup is usually used to know how to display and process the document content.

Depending on the complexity of the markup format the text can be written using a plain text editor (an editor that simply manipulates characters and lines using a fixed width font), a programmable plain text editor (when the text is readable but the format has so many rules than we prefer to use the help of the computer to mark the text) or a word processor that hides the markup complexity and usually lets us work using a graphical view that is approximately equivalent to the desired output format (what we know as *WYSIWYG*).

On the article titled *Markup systems and the future of scholarly text processing* by James H. Coombs, Allen H. Renear, and Steven J. DeRose, published in the November 1987 CACM (available on the URL <http://xml.coverpages.org/coombs.html>) the authors identify six different types of markup:

- *Punctuational.* Punctuational markup consists of the use of a closed set of marks to provide primarily syntactic information about written words. This kind of markup is part of writing systems and is not usually considered a special markup language (punctuation is included on all the other types of markup).
- *Presentational.* This kind of markup is used to identify the structure of the document using *cues* on the text encoding, like leaving empty lines between paragraphs or putting spaces before a string to center it to denote a title.

  This kind of markup clarifies the presentation of a document and makes it suitable for reading, but it does not provide information about the structure of the document (word-processing and desktop

publishing products sometimes attempt to deduce structure from presentational conventions, but it is almost an impossible task, as there is no common standard).

- *Procedural.* In many text-processing systems, *presentational markup* is replaced by *procedural markup*, which consists of commands indicating how text should be formatted; documents written using this kind of markup can be viewed as *programs* to generate a final document, as the commands are usually visible to the user and, in most cases, the procedural markup capabilities comprise a Turing-complete programming language.

  As happens with presentational markup, the procedural markup by itself does not provide information about the logical structure of the document, although some of them define commands that implicitly give some structural information (i.e., a command to build a title).

  Systems like `nrof/troff`, `TeX` and `PostScript` are examples of this kind of markup systems.

- *Descriptive or semantic.* This kind of markup systems identify the element types of text tokens, not how they should be presented. The output format of documents written using *descriptive markup* is generated by processing the descriptive elements and applying to them *procedural markup* rules. The good thing about this type of markup is that the procedural rules can be replaced by others, allowing us to have multiple output formats and styles without extra effort.

  Examples of this kind of systems are `SGML` and `XML`.

- *Referential.* Referential markup refers to entities external to the document and is replaced by those entities during processing; that is the kind of markup used to replace special characters or abbreviations. Almost all text formatters that support procedural markup support some kind of referential functionality using variables or macros, but usually referential markup is associated with descriptive markup systems like `SGML`.

- *Metamarkup.* Metamarkup provides facilities for controlling the interpretation of markup and extending the vocabulary of descriptive markup languages. Metamarkup provides tools to define macros, tags, valid and default attributes, etc. Almost all nontrivial systems support metamarkup, but most do not provide a suitable interface for non-programmers.

From this six types of markup we will only take into account three of them, *presentational*, *procedural* and *descriptive*, as the other ones are usually always available on the systems that use one of those three paradigms.

So, what are the advantages or disadvantages of using one of those three markup systems?

It depends on what is the author interested in, if he or she is interested on the *content* of the document (usually the case for developers), *descriptive markup* is better, as it helps the author to focus his attention on the structure and content of the document (it has to be declared explicitly), while *presentational* and *procedural markup* fail to support the writer in developing the structure; even worse, they distract from the content.

As is discussed on the article cited before, the first step of marking up a document, element recognition, is the same for all forms of markup, but the next step, markup selection, always involves some additional effort, but while *descriptive markup* keeps this effort focused on the element and its role in the document, *presentational markup* turns the author's attention toward typographic conventions and style sheets and *procedural markup* leads even further away from the document toward the special markup required to make a particular text formatter produce the selected *presentational markup*.

5 Lightweight Markup Languages

We are basically looking for a document format that allows a developer to write structured documents with the following characteristics:

- the document source should be writable using standard plain text editors (ideally it has to be very easy to write, allowing developers to use the same tool used for writing code),

- the *markup language* used has to be easy to read and learn (someone not familiar with the markup has to be able to read the source document without problems and be able to write documents using it on a few hours).
- the *document source* should be easily manipulated in a collaborative way and different revisions should be comparable or mergeable without special tools.
- the *document source* should be transformable into other formats suitable for output systems like text terminals, web browsers or printers, directly or using intermediate formats (usually the use of intermediate formats has the advantage that we can re-use existing tools and style sheets).

As those goals are shared by many people, a lot of *lightweight markup languages* formats and tools have been developed on the recent years; the main idea behind almost all of them is to avoid the use of complex *descriptive markup* formats based on **SGML** or **XML** like html and xhtml or DocBook and the aproach taken is to define *markup* formats that use a set of simple rules to provide a subset of the information that can be given using the *heavyweigh* languages.

Usually the *markup* used by those systems that can be seen as *punctuation* for the casual reader, some times quite weird punctuation, but generally not distracting if you are only interested on the text content.

Once the format is defined we just need a parser for the markup rules and conversion tools to transform the parsed document into another format directly usable by output systems like text terminals, web browsers or printers or into an intermediate format like `TeX` or `XML` that has to be processed by other tools to be readable on the output systems.

Inside this category we find a lot of formats and tools with different features; there are a lot of tools that are only used to transform text into HTML an are used on web based systems like blogging software, content managers or wiki systems.

The use of those systems has multiple advantages; it is a good way to allow normal users to edit dynamic web content without forcing them to learn HTML and avoids some of the problems of letting them to use HTML directly (it is easier to keep a consistent look and feel and avoids harmful or annoying java script code on the pages).

On the following section we will talk about some free *Lightweight Markup Languages and their related tools*.

6 Lightweight Markup Languages

The following table lists some lightweight markup languages and the URL of tools and documentation related to them:

Language or Tool	**URL**
Almost Free Text (aft)	http://www.maplefish.com/todd/aft.html
AsciiDoc (adoc)	http://www.methods.co.nz/asciidoc/
Markdown (mdwn)	http://daringfireball.net/projects/markdown/
StructuredText (stx)	http://sange.fi/~atehwa/cgi-bin/piki.cgi/stx2any
reStructuredText (rst)	http://docutils.sourceforge.net/rst.html
Textile (txtl)	http://www.textism.com/tools/textile/

As I am a Debian GNU/Linux user and advocate and all the formats have tools to handle them on that distribution I'm also listing here the packages that a user has to install to be able to write and process documents using those formats on a Debian Sid system; note that the tools can need additional programs to generate final formats, i.e. to generate a `pdf` file from a `reStructuredText` document we generate a `.tex` file using `rst2latex` (a tool included on the `python-docutils` package) but later we need to process the resulting file with `pdflatex`, a program that is included on a separate package not

listed here:

Tool	Package
Almost Free Text (aft)	`aft`
AsciiDoc (adoc)	`asciidoc`
Markdown (mdwn)	`markdown` or `python-markdown`
StructuredText (stx)	`stx2any`
reStructuredText (rst)	`python-docutils`
Textile (txtl)	`python-textile`

To compare those languages and tools we should look at the following features:

- Output formats supported: html on one file, html on multiple files, xml, latex, troff, ...
- Customizable output: how is the output format generated? can we customize the output? can we apply stylesheets to the output format?
- Types of documents: articles, books, manpages, ...
- Does the format support metadata (author, title, ...)
- Does the format support different text styles: bold, italic, underlines, ...
- Which kind of logical structures does the format support: - Paragraphs - Lists: enumerated, numbered, definition lists, ... can they be nested? - Blocks (code samples, citations, notes, ...) - Tables - Figures - Images - ...
- Does the format support advanced features like: - Table of contents, - Table of figures, - Indexes, - Bibliographies
- Does it support some form of mathematical notation.
- Does the system support the use of URLs and internal references?
- Can we use comments
- Can we split the document into multiple subdocuments?
- Does the language provide an extension mechanism to provide new markup
- Can we include markup on the output format on our document to overcome limitations?
- Others?

7 An example system: `reStructuredText`

Quoting the reStructuredText main page:

> reStructuredText is an easy-to-read, what-you-see-is-what-you-get plain text markup syntax and parser system. It is useful for in-line program documentation (such as Python docstrings), for quickly creating simple web pages, and for standalone documents. reStructuredText is designed for extensibility for specific application domains. The reStructuredText parser is a component of Docutils. reStructuredText is a revision and reinterpretation of the StructuredText and Setext lightweight markup systems.
>
> The primary goal of reStructuredText is to define and implement a markup syntax for use in Python docstrings and other documentation domains, that is readable and simple, yet powerful enough for non-trivial use. The intended purpose of the markup is the conversion of reStructuredText documents into useful structured data formats.

The main advantage of **reStructuredText** over other lightweight markup systems is that it has a well defined syntax and there are a lot of tools already available to transform `rst` text files into different output formats like `html`, `xml`, `pseudo xml`, `latex`, `s5` (for making html slides).

Besides, as the parser is a component of docutils and can be easily used from other programs implemented in Python, a lot of web systems written in Python are including plugins to enable the use of reStructuredText and generate HTML form it.

Some of the programs that have support for `rst` include:

- MoinMoin: a Wiki System,
- Plone: a CMS based on Zope,
- PyBloxsom, a blogging software system,
- Trac: An integrated SCM and project management tool that includes an enhanced wiki and an issue tracking system for software development projects.
- Zope: a Python application server,

Some examples of the markup rules of the `reStructuredText` format:

- *paragraphs* are chunks of text that are separated by blank lines (one is enough).
- *text styles* like *italics* or **bold** are marked surrounding words between asterisks for "`*italics*`" or double asterisks for "`**bold**`".
- *numbered lists* are built putting at the begining of a paragraph a number or letter followed by a period ".", a right bracket ")" or surrounded by brackets "( )"; a list paragraph can have multiple lines if the lines following the first one have the same level of indentation of the fist letter that follows the period.
- *enumerated lists* are built as the *numbered ones*, but the lines have to start with a bullet point character, either "-", "+" or "*".
- the text is divided into sections using **section headers**, that are a single line of text (one or more words) with an underline or an underline and an overline together, built using dashes "`----`", equals "`======`", tildes "`~~~~~~`" and other characters. The heading level is determined automatically, giving to each decoration a different level in order of appearance.

For a complete reference of the reStructuredText syntax see the *reStructuredText Markup Specification* available on the URL http://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html.