Creating and maintaining tutorials with DEFT

Andreas Bartho
Department of Computer Science
Technische Universität Dresden
Dresden, Germany

Abstract

For frameworks or software libraries to be used correctly, good documentation is crucial. One important kind of framework documentation are tutorials, textual explanations of how to use the framework, enriched with source code examples. Unfortunately tutorials are rare in comparison to other kinds of documentation such as API documentation due to the effort of creating and maintaining them.

In this paper we highlight the problems and difficulties of writing and maintaining tutorials and identify the potential for tool support. Then a tutorial development tool, DEFT, is presented, which was created to address the identified points and make tutorial creation and maintenance easier.

1 Introduction

It is essential for frameworks or software libraries that there is enough documentation for developers to understand them and use them as intended. Tutorials, textual explanations enriched with source code examples, are one possible kind of framework documentation.

While API documentation, another kind of source code related documentation, is usually extensive, tutorials are rarely written in comparison. We believe this is because API documentation can be automatically generated from source code and specially formatted comments, whereas writing and maintaining tutorials is mainly a manual task, which is detached from actual programming. Our goal is therefore to bring programming and tutorial writing closer together.

In section 2 we identify different kinds of tutorials and present existing approaches related to tutorial writing. Section 3 analyzes the manual process of writing and maintaining a tutorial and highlights its drawbacks. Possible improvements are presented in section 4, involving our tutorial tool DEFT[1]. Finally the paper concludes with section 5, presenting an outlook and further work that must be done.

[1] `http://deftproject.org`

2 Classification and related work

According to Meusel et al. documentation should exist on different abstraction levels, following the so-called pyramid principle [3]. For example, high-level descriptions, located at the *Framework Selection Level* give a rough overview and should answer the question, whether the framework will be able to solve a given problem. For actual programmers it is most important to learn *how* to accomplish a specific goal. They need very detailed, code-centric documentation, which is addressed at the *Detailed Design Level*.

There are two main types of such code-centric documentation: API documentation and tutorials. API documentation, e.g. Javadoc, consists of a listing of framework classes and methods, along with a description of their purpose and behaviour. A tutorial on the other hand is a textual explanation of a program (or an excerpt) accompanied by source code examples. Tutorials can be written for various purposes.

Explaining language features To give novices an introduction to a programming language, there are tutorials that explain the language in question step by step, e.g. how to write simple statements, how to use loops, how to subclass a class, and more. An example are the comprehensive Java tutorials from Sun[2].

Internal documentation A complex software product, be it a program sold to customers, or a framework intended to be used by third parties, needs to be maintained over time. Important parts of the software and complex algorithms which cannot be well described by inline comments are best documented with a tutorial. This helps both experienced maintainers to look up details they have forgotten and new colleagues to get a grasp of the software.

Framework usage documentation Unlike the *internal documentation*, this documentation is intended for end

[2] `http://java.sun.com/docs/books/tutorial/`

users who wish to build applications using a framework (or software library). A tutorial describes how to develop an application which makes use of various extension points of the framework. This example application can either "grow" during the tutorial, i.e. first there is only a skeleton, which is extended and filled with life as the tutorial introduces new functions of the framework. The code that must be written to make use of these functions is inserted as code examples. In contrast, it is also possible that there is one example program which is not changed throughout the tutorial. When a functionality of the framework is introduced, the according parts of the example program are presented as code examples.

A similar, often used approach are cookbooks. These are a collection of recipes describing in text and code how to solve a specific problem. In contrast to tutorials they usually do not refer to a continuous example program but are independent of each other. The advantage is that much more examples and a discussion of alternatives can be easily presented. In most cases, a normal tutorial only shows one possible solution to a specific problem. The disadvantage is that it is difficult in cookbooks to show how different aspects of the framework can be combined. A tutorial with a well-chosen example application is better suited for that.

One way to create a tutorial would be to write the program being documented, write the tutorial text and copy code fragments into the tutorial text. This has several drawbacks that will be highlighted in section 3. There are, however, also different approaches known from literature.

A famous attempt to bring textual documentation and source code together is the Literate Programming approach by Donald E. Knuth [2]. Programs are considered work of literature. In Literate Programming a program is described for humans to understand. A literate program is mainly explanatory text with (manually) embedded source code fragments. With the help of tools one can both produce a formatted document (weave) and an executable program (tangle).

The Elucidative Programming approach was suggested by Kurt Nørmark [4] as Literate Programming was never adopted by programmers. Despite its name it is not a programming technique but rather a documentation technique. The goal of Elucidative Programming is also to bring text and source code together, but unlike in Literate Programming the entire program exists at the time of writing and is not generated afterwards. This conforms much more to the way programmers work. Therefore Literate Programming was identified to be a good solution for publishing programs as technical literature, whereas Elucidative Programming is better suited for use in real software projects.

When writing text the author can refer to named entities of the program, e.g. classes or methods, and arbitrary locations in the code, which contain comments with markup. Using special tools an HTML representation of the documentation can be created, which allows to navigate via hyperlinks from the textual explanation to a program entity and from a program entity back to the text which references it.

Originally, Elucidative Programming was intended to be used for *internal software documentation* [4]. This scope has broadened, after the usefulness for *framework usage documentation* has been realized. Having source code snippets embedded in the explanatory text has been found a help for the reader, resulting in documentation that looks very much like a literate program [6]. The code snippet inclusion does also use references, nothing is copied and pasted.

3 Requirements for a tutorial editor

Writing tutorials "by hand" has a lot of serious drawbacks which could be diminished with appropriate tool support. In order to see what this tool support should look like we will first analyze the manual process of tutorial writing. We will concentrate on *framework usage documentation* here, but the other kinds of tutorials presented in section 2 can be handled similarly.

3.1 Tutorial creation

Consider the following scenario: A new framework which provides a lot of extension points has been developed. To help developers become familiar with the framework, a tutorial should be written. The first step is to create an example program which makes use of the most important extension points. Then explanatory text is written, describing what the program does and why it has been implemented the way it is. Code snippets are copied and pasted from the IDE into the tutorial text to illustrate the described implementation. Once finished, both text and code are formatted for a visually appealing result.

This naive approach has a number of disadvantages. First of all, the tutorial writer has to switch constantly between text editor and IDE to copy code into the text. Furthermore, formatting possibly lot of code snippets by hand is boring and error-prone. The conversion to different formats, e.g. from an HTML version to a Word version means additional effort.

A tutorial tool should offer both the functionality to write text and to access the code snippets to be included. Formatting of the embedded code snippets should be done automatically. Export to different output formats should be possible.

3.2 Tutorial maintenance

After the tutorial has been written and published, development of the framework does usually not stop. The next framework release is likely to have some of its interfaces changed, rendering the existing tutorial, at least partially, useless. Therefore the tutorial must be updated. First of all, the example program has to be adapted to work with the new framework version. This is usually easy because of refactoring support modern IDEs provide. The next step is to proofread the tutorial, replace code snippets that have been changed and update the corresponding tutorial text. This, however, is an annoying, tiresome task with lot of possibilities to introduce inconsistencies or errors. Usually only a fraction of the API changes, nonetheless the complete tutorial must be checked in order to not overlook a change. Compare that to driving for hours on a long straight highway with cruise control enabled, monotonous landscape around. Nevertheless you must stay alert all the time in order to brake for the deer that might or might not suddenly jump in front of your car.

A tutorial tool should offer support for tutorial maintenance. Embedded code snippets should be updated automatically, if possible. Furthermore, the tutorial writer should be informed which code snippets have been updated. That way he can concentrate on proofreading the surrounding tutorial text, which is likely to need some changes, instead of having to read the whole tutorial, hoping to find all inconsistencies between text and code snippets.

4 The DEFT approach

Inspired by the Elucidative Programming approach we have developed the tool DEFT (Development Environment For Tutorials), that aids in creating and maintaining tutorials. The author can not only write text but also drag and drop source code snippets into the tutorial. Internally this creates a reference to the code, but both the tool itself and the compiled (HTML) output display the actual code snippet.

4.1 Parts of a tutorial project

The process of writing a tutorial is similar to the naive approach as outlined in section 3. First the example program using the framework has to be developed. The actual tutorial writing is done in DEFT, however.

The first step is to create a DEFT *Project*. A *Project* consists of various *Fragments* such as *Chapters*, *CodeFiles*, *CodeSnippets*, *Images* and *Tutorials*.

A *Chapter* is a file containing the tutorial text. It is basically an XHTML document that allows some additional tags to represent embedded or linked source code.

A *CodeFile* is a file containing source code to be documented, e.g. a C# code file. In principle, DEFT can work with any kind of structured file. It is necessary to have an according plugin that can handle the parsing of the file format. Currently there are plugins for C# and Java.

A *CodeSnippet* is an excerpt of a *CodeFile*, for which a reference has been created. That is, *CodeSnippets* represent the code fragments to be included in a *Chapter* or hyperlinks referring to certain positions in the code. Internally a *CodeSnippet* is mainly a set of XPath path expressions pointing to nodes of a *CodeFile*'s syntax tree.

An *Image* is exactly that, an image. Images can be added to *Chapters*.

A *Tutorial* is a list of *Chapters* that belong together. Splitting a tutorial among several *Chapters* allows to document different concerns of the example program separated from each other. This makes the *Project* more structured. If well-written, the individual *Chapters* are independent and can be arbitrarily combined and arranged in multiple *Tutorials*. This concept is called Documentation Threads [5].

4.2 Writing a tutorial with DEFT

Now that the main concepts have been introduced, we shall analyze how the manual process presented in section 3 is actually improved by using DEFT.

4.2.1 Tutorial creation

After the *Project* has been created, the program to be documented must be imported. All files of which the program consists are stored inside DEFT as *CodeFiles*. Then a *Chapter* has to be created. DEFT has a built-in WYSIWYG XHTML[3] editor for writing tutorial text. To add code examples to the tutorial text, the according *CodeFile* must be selected. This causes its syntax tree to be displayed, from which the relevant nodes can then be dragged and dropped into the editor. In the background a *CodeSnippet* is created. Furthermore, a reference to this *CodeSnippet* is inserted into the tutorial text. That happens transparently for the user. The code is immediately displayed as if it had been copied, pasted and beautified by hand. Additionally, DEFT gives the tutorial writer the possibility to change the appearance of the code by applying a so-called *Format*. *Formats* are sets of rules that cause some parts of the embedded code to be hidden or replaced. It is, for example, possible to display only the signature of an included method, or to remove all inline comments.

After all *Chapters* have been written and grouped together to a *Tutorial*, the *Tutorial* can be exported, for example to HTML. In its current version, DEFT offers to export in two layouts. In the Single Frame Layout the output looks

[3] VEX, available on http://vex.sourceforge.net/

exactly as in DEFT itself, i.e. text with embedded code. In the Two Frame Layout frames are used. The left frame contains the tutorial text with embedded code, the right frame shows complete source files. The embedded code in the left frame and the corresponding positions in the right frame are mutually hyperlinked, i.e. the reader can navigate from explained code to its position in the source file, and he can also see which parts of the source file have been documented and jump to the according position in the tutorial text.

In summary, using DEFT when writing tutorials offers the following advantages.

- no copying and pasting of code between different applications
- easy and fail-safe embedding of arbitrary code snippets into tutorial text
- automatic formatting and easy customization of embedded code snippets
- export into various (hyperlinked) formats possible

4.2.2 Tutorial maintenance

When the framework to be documented evolves and interfaces change, the tutorial has to be updated. The first step is to make the example program work again with the new framework version. The next step is to make the outdated tutorial consistent with the updated example program. Thanks to the referencing mechanism this is much easier with DEFT than doing it manually.

To update the tutorial the underlying *CodeFiles* must be updated. This is done similarly to the import of *CodeFiles*. Once the *CodeFiles* are updated, the embedded references (i.e. the *CodeSnippets*) are automatically reevaluated and the code embedded in the *Chapters* corresponds to the updated *CodeFiles*. It is, however, possible that some code references point to nowhere, e.g. if a method has been referenced but its name has changed. To find such cases easily, all occurrences of *CodeSnippets* that have changed are displayed as warnings in a list. Selecting an item from the list shows the *Chapter* with the embedded *CodeSnippet*. If necessary, the old reference can be refined or deleted and recreated. If the tutorial writer has approved a changed *CodeSnippet* and made sure that it is still consistent with the surrounding text, the warning can be marked resolved and the next warning can be processed.

In summary, using DEFT for tutorial maintenance has the following advantages.

- update of changed code files updates embedded code snippets automatically
- notification which code snippets have been updated

5 Conclusion and future work

In this paper we motivated the need for tool support in tutorial writing and presented our tutorial development tool DEFT. It is based on the Elucidative Programming approach. Its implementation goes beyond previously available tools. For once, DEFT is not restricted to a single programming language, it can easily be extended. Furthermore, arbitrary portions of source code can be addressed without the need to mark up source code.

Along the development of DEFT a number of interesting problems have occurred, which are not only relevant to tutorial creation but also related fields. Some solutions have already been described ([1], not yet published), a number of others are planned to be published.

However, the tool is still work in progress and has yet to be evaluated. Current development focuses on user acceptance, i.e. providing all the necessary functions to make tutorial creation easier, along with a good user interface. Once the tool is mature enough, a study will be conducted to identify strengths and weeknesses. Additionally it must be investigated in a long-term study and a "real" framework whether DEFT does really fulfill its main goal: making tutorial maintenance easier. This includes an analysis of how stable the *CodeFragments*' source code references are when *CodeFiles* are updated. Under which circumstandes will they still point to the right code positions? When will the reference mechanism fail and what can be done to diminish negative effects?

The results of these studies will be used to further improve the tool and hopefully help making software documentation a little better.

References

[1] A. Bartho. Adding preprocessor directives and comments to syntax trees. `http://bartho.net/publications/Adding_preprocessor_directives.pdf`, 12 2008.

[2] D. E. Knuth. Literate programming. In *The Computer Journal*, volume 27(2), pages 97–111, May 1984.

[3] M. Meusel, K. Czarnecki, W. Kpf, and D. benz Ag. A model for structuring user documentation of object-oriented frameworks using patterns and hypertext. In *Proceedings of ECOOP'97. LCNS 1241*, pages 496–510. Springer-Verlag, 1997.

[4] K. Nørmark. Requirements for an elucidative programming environment. In *Eight International Workshop on Program Comprehension*, June 2000.

[5] T. Vestdam. Documentation threads - presentation of fragmented documentation. *Nordic Journal of Computing*, 7(2):209–230, 2000.

[6] T. Vestdam. Elucidative programming tutorials. *Nordic Journal of Computing*, 9(3):209–230, 2002.