The Vicinity of Program Documentation Tools

Kurt Nørmark
Department of Computer Science
Aalborg University
Denmark
normark@cs.aau.dk

Abstract

Program documentation plays a vital role in almost all programming processes. Program documentation flows between separate tools of a modularized environment, and in between the components of an integrated development environment as well. In this paper we discuss the flow of program documentation between program development tools. In the central part of the paper we introduce a mapping of documentation flow between program development tools. In addition we discuss a set of locally developed tools which is related to program documentation. The use of test cases as examples in an interface documentation tool is a noteworthy and valuable contribution to the documentation flow. As an additional contribution we identify several circular relationships which illustrate feedback of documentation to the program editor from other tools in the development environment.

1. Introduction

Program documentation tools can be categorized in two flavors: (1) Tools which document the external interface of a program library, also known as the application programming interface (API). (2) Tools which document the internal details of a program, either a library or an application. Inspired from the vocabulary used in program testing we call tools of the first category for *black box program documentation tools*. Tools of the second category are called *white box program documentation tools*.

This paper researches the vicinity of program documentation tools in order to track the interaction between documentation tools and other program development tools. We are primarily interested in a setup with separate, cooperating development tools. Such environments will be called *modularized environments*.[1] A modularized environment is attractive because it can be composed of tools which suit the needs and preferences of the individual programmer. Within a modularized environment one tool may be substituted by another similar tool as long as certain protocols and conventions are obeyed in relation to other tools.

The primary goal of the work described in this paper is to understand the flow of program documentation in between tool of a modularized environment. Based on this understanding we wish to improve the use of program documentation across a set of tools. The insight of the paper can also be used to design the flow of program documentation between a more tightly integrated set of tools. The concrete experience behind the paper comes from the author's work on program documentation tools for the programming language Scheme, used from the Emacs text editor.

In the paper we will first briefly review the area of black box and white box program documentation tools. In this part of the paper we will draw the attention to related work in the area. Following that, in Section 3, we analyze the vicinity of program documentation tools in order to understand how program documentation flows between a variety of different program development tools. In Section 4 we describe a concrete set of program development tools from which we have drawn the experience described in this paper. In section 5 we identify and discuss some circular relationships which represent documentation feedback to the editing tool. Finally, in Section 6, we draw the conclusions of the work.

[1] A *modularized environment* is seen as a contrast to a tightly *integrated environment*. In an integrated development environment the identities of individual tools are blurred.

2. Program Documentation Tools

The most widespread program documentation tools - *interface documentation tools* - extract information about the abstractions in a source program which are relevant for external use. The focus on program interfaces - APIs - reflects the black box nature of such tools: The internal program details are "blacked out" in the documentation. Typically, the extracted parts include signatures of public procedures/functions/methods in public modules/classes. The extracted parts of the source programs are augmented with information from designated *documentation comments*. The documentation comments are written in a *documentation language* which provides structural and typographical means of markup. The documentation comments are located in close proximity to the extracted source code fragments. The extracted information is aggregated as interlinked HTML pages, intended to be viewed in an internet browser.

Javadoc [3] made interface documentation tools popular for the Java programming language in the mid-nineties. Prior to Java and Javadoc, similar approaches were used for production of paper-based program documentation via text formatting tools such as TeX or Nroff. Most programming languages are today accompanied by interface documentation similar to Javadoc. Doxygen [17] is one of the most elaborate and full-feature tools in this genre. Doxygen can be used together with several programming languages, most notably languages in the C family.

The tools that support documentation of the inner workings of a program are less widespread than interface documentation tools. In this paper such tools are referred to as white box documentation tools. Literate programming [8], as pioneered by Donald Knuth, represents the most important school in the area of internal program documentation. A literate program is integrated with the "story about the program". In a literate program, the actual pieces of programs are annotations of the program explanations. In more conventional programs, some pieces of explanations - represented as comments - annotate the source code. Numerous literate programming tools have been developed [1, 20, 9, 5, 15] but - in contrast to interface documentation tools - none of them have been adopted by the software industry. Inspired by literate programming, other techniques for internal program documentation have been proposed. Elucidative Programming [11] relies on a separation of documentation and source programs, with bidirectional links in between them. Theme-based Literate Programming [6] and its supporting tool, supports documentation of several paths through (or aspects of) a program. Simonis and Weiss [16] cover part of this landscape as of 2003. Vestdam and Nørmark [18] discuss additional aspects.

3. The Vicinity of Documentation Tools

In this section we will discuss a map of program documentation tools and the information that flows in between these and other program development tools. With this map we wish to zoom in on the flow of program documentation in a complex web of program development tools. Throughout the paper we will address and discuss the map which is shown in Figure 1. For ease of identification, capital letters in the figure refer to specific tool, and numbers refer to information that flow between the tools.

The interface documentation tool (A) is the starting point of our exploration. As it is shown in the figure, the interface documentation tool produces *interface documentation pages* (1) which can be examined interactively in a documentation browser (F). The documentation browser is typically a general purpose internet browser. The interface documentation pages are usually regarded as the primary deliveries of the interface documentation tool.

The internal documentation tool (B) supports descriptions and explanations of the implementation of a program (a library or an application). The internal documentation tool is directed towards internal or private details of a program component. As such, the internal documentation creates (more or less integrated) presentations[2] (3) and explanations (4) of the source program.

In many situations it makes good sense to connect the presentation of the source programs (3) to the documentation pages (1) via navigatable links. This allows for source program browsing from a starting point of the program interface documentation. In many respects this is a very useful way to approach the program details. In situations where the documentation of the API is not precise or comprehensive, some users may want to inspect the actual code behind the interface. This may be necessary in order to find answers to questions, which are not covered by the interface documentation. The other way around, it is also useful to consult the interface documentation (1) from a

[2] In this work, the presentation of a source program (3) is different from the actual source program (7). In the presentation of the source program applied names are cross-linked to their definitions. In addition, typographic means of expressions (including coloring) may be applied. The actual source program represents the view of the source program in the program editor. Integrated development environments blur the distinction between these two renderings of the source program.

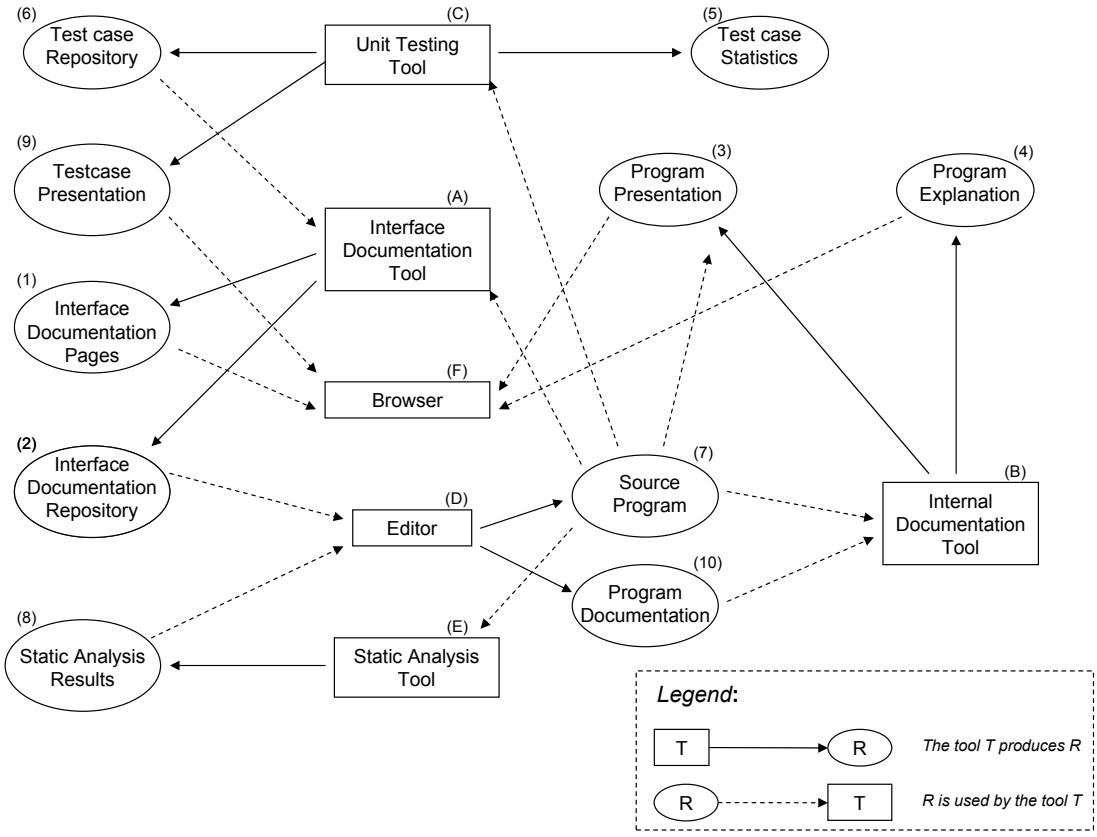

Figure 1. *A map of the program development tools, with special emphasis on documentation tools, together with the information that flows in between them.*

presentation of a source program (3), instead of understanding all pieces of the program at the most detailed level.

The interface documentation pages (1) produced by the interface documentation tool (A) contain information which is useful for other tools as well. It is not effective for such tools to access the documentation pages (1) as such. The documentation pages are intended for human reading. Therefore, the interface documentation tool should also create contributions to an *interface documentation repository* (2). The interface documentation repository should organize program documentation in such a way that it can be accessed efficiently from other tools.

Concrete examples that illustrate uses of program abstractions are very useful in the context of program interface documentation (1). An example shows how to use a documented abstraction, and it somehow presents the results or effects of the use of the abstraction. In principle, it is possible to manually author the examples side-by-side with other pieces of the interface documentation. In such as setup the examples could be organized inside the documentation comments of the source program. In reality, it is not realistic to go for such an organization. The primary reason is that it would be difficult to maintain the correctness of the examples. We will now explain how to provide a better solution via integration with a unit testing tool.

The unit testing tool (C) works on test cases which typically are organized in ordinary source programs (7) and produced by the program editor (D). The primary artifacts produced by a unit testing tool (C) is the test case statistics (5). A test case consist of a program fragment and its expected result. The unit testing tool is able to execute a program fragment and compare the outcome with the expected result. The pieces of information in a test case may directly serve as an example in interface documentation. It may, however, require

some "digging" to extract the example program fragment and its intended result from a unit test. As an alternative approach, it may be possible to organize the testing process or the testing tool such that these informations are readily available (see Section 4). We propose that the unit testing tool (C) derives an *test case repository* (6) which is organized side by side with the interface documentation repository (2).

As it is shown in Figure 1, the test case repository (6) is connected to the interface documentation tool (A). Via this connection, the interface documentation tool is able to extract relevant test cases and to use the extracted test cases as examples in the interface documentation. The extraction can either be done automatically, simply by selecting those test cases where a given function appears in the program fragment of the test case. Alternatively, it may be possible to augment the entries in the test case repository with hints about the relevant documentation context of a given test case. This possibility can be seen as fine tuning of the examples in the interface documentation (1).

The programmer is supposed to consult the program documentation pages (1) when he or she uses the program editor (D) in the programming process. This may be disruptive because it typically enforces a context shift from the editor (D) to the documentation browser (F). It is attractive to provide "easier access" to the most important information in the program documentation pages. Easy access means that the information appears more or less automatic when needed, and that it disappears again when not needed anymore. Modern IDEs, such as Visual Studio from Microsoft, have paved the way to "easy access" via so-called *intellisense*. Intellisense also involves flexible name completion, either automatically when possible in some given context, or via selection from a list of possibilities.

As just discussed, the information from the interface documentation repository (2) is a valuable contribution to the editor (D), for presentation of API documentation and for name completion purposes. It represents knowledge of relevant public definitions from classes and modules, on which a given program relies. The information taken from the interface documentation repository (2) does not, however, account for full documentation in a given programming situation. It needs to be augmented with information about nearby definitions in the current class/module/application and information about locally bound names in the definition under development (local variables and parameters). This information can be provided to the editor by means a static analysis tool (E) which is applied on the current source file.

Figure 2 illustrates the program editor in a situation

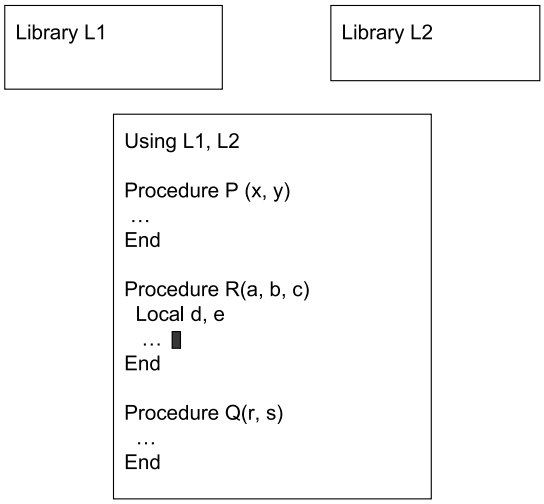

Figure 2: *A program with the procedures P, Q, and R which depend on two libraries L1 and L2. The programmer currently works inside R at the place of the solid cursor.*

where a program (represented as the large box in the figure) uses two libraries L1 and L2 (the smaller boxes). The programmer is currently working on the procedure R, which is a sibling of the procedures P and Q. Inside R, the parameters a, b, c and the variables d and e make up the local name context. The documentation of the libraries L1 and L2 is taken form the interface documentation repository (2). Information about P and Q as well as the information about the local names a, b, c, d, and e is provided by the static analysis tool (E). The documentation from the program documentation repository changes slowly, whereas the documentation of local names in the context of the working point changes rapidly. These observations affect the scheduling of the recalculation of the documentation relative to the point of interest in the editor.

4. Concrete tools in the vicinity of SchemeDoc

In this section we will discuss the concrete tools and the concrete experiences behind the general observations in Section 3. The documentation tools, and the connections in between them, have been constructed and elaborated in the slipstream of the work on LAML [13]. LAML is a software package which makes XML languages available in the programming language Scheme [7]. Scheme is a functional program-

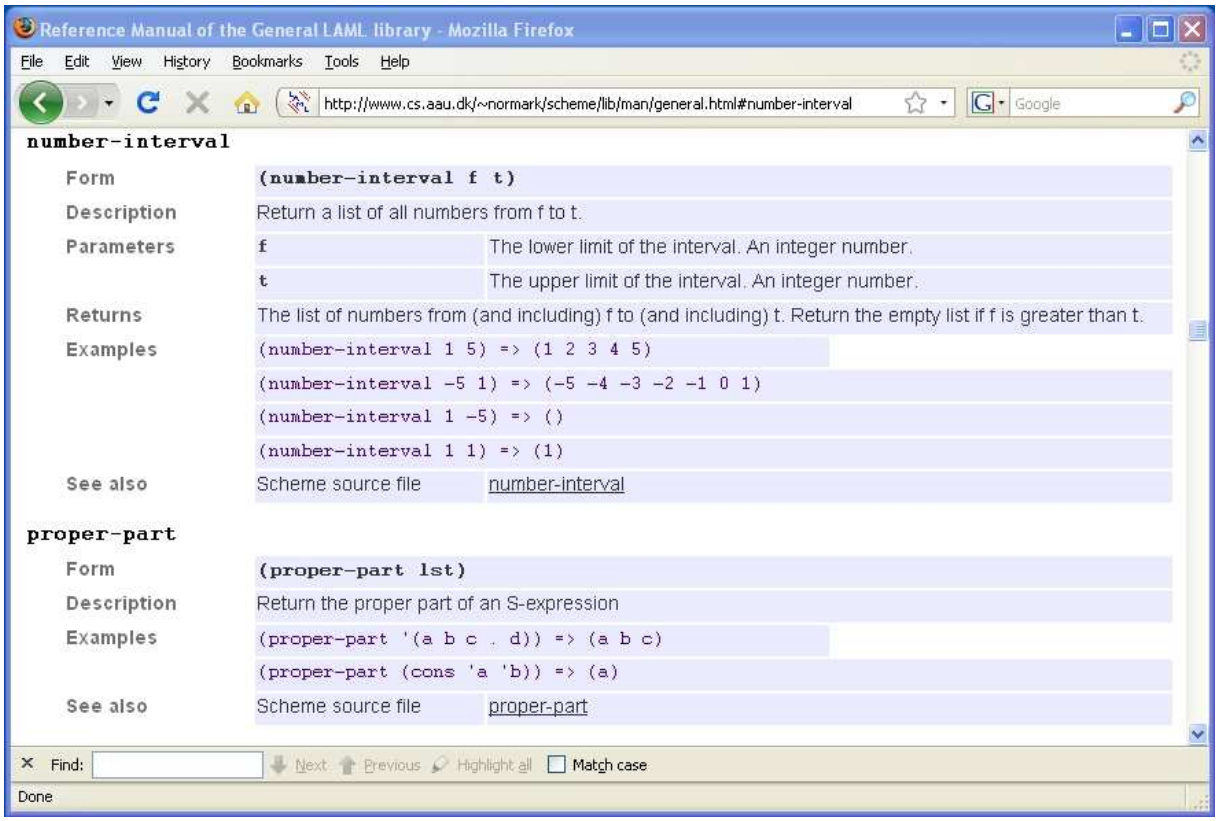

Figure 3. *Two documentation entries augmented with examples captured from interactive unit testing.*

ming language in the Lisp family. LAML is a non-trivial collection of software (it currently counts approximately 80.000 lines of code) and it is therefore essential that available documentation is used to its full potential. As a consequence, the LAML project has involved substantial tasks in the area of program comprehension [11, 12]. In the discussion below we refer to the same numbering of tools, documents, and repositories as used in Figure 1 of Section 3.

The interface documentation tool (A) is LAML SchemeDoc [12] which we have developed to support the development of LAML. SchemeDoc is similar to most other interface extraction tools, and originally inspired by Javadoc [3, 10]. The documentation produced by SchemeDoc has played a vital role for the documentation and the management of the LAML software. SchemeDoc produces HTML documentation (1) which can be accessed from any internet browser (F). The SchemeDoc tool also produces an internal data structure (a nested list structure) which plays the role of the interface documentation repository (2).

The tool used for internal documentation of the LAML software is the Scheme Elucidator (B) [11]. It generates strictly separated program explanation (4) and program presentation pages (3), which are mutually navigatable in a two-framed browser setup. Internally, SchemeDoc (A) activates the Scheme Elucidator (B) with the purpose linking the interface documentation pages (1) to source program presentations (3). From an entry in the interface documentation there is access to the source program details of the entry. The links in the source program presentation connect applied names to defined names (in the same source file presentation, or in presentation of other source files); "Reserved names" in the Scheme programming language are linked to the relevant entries in the Scheme reference manual; And defined names in a program presentation (3) are linked to entries in the interface documentation (1) if available.

It is an non-trivial task to produce richly linked presentation of the source program (3) because it requires simultaneous traversals of the parsed program and its textual form. From the point of view of the interface documentation tool (A) it therefore makes good sense to produce the source program presentation (3) via use of the internal documentation tool (B), for which the

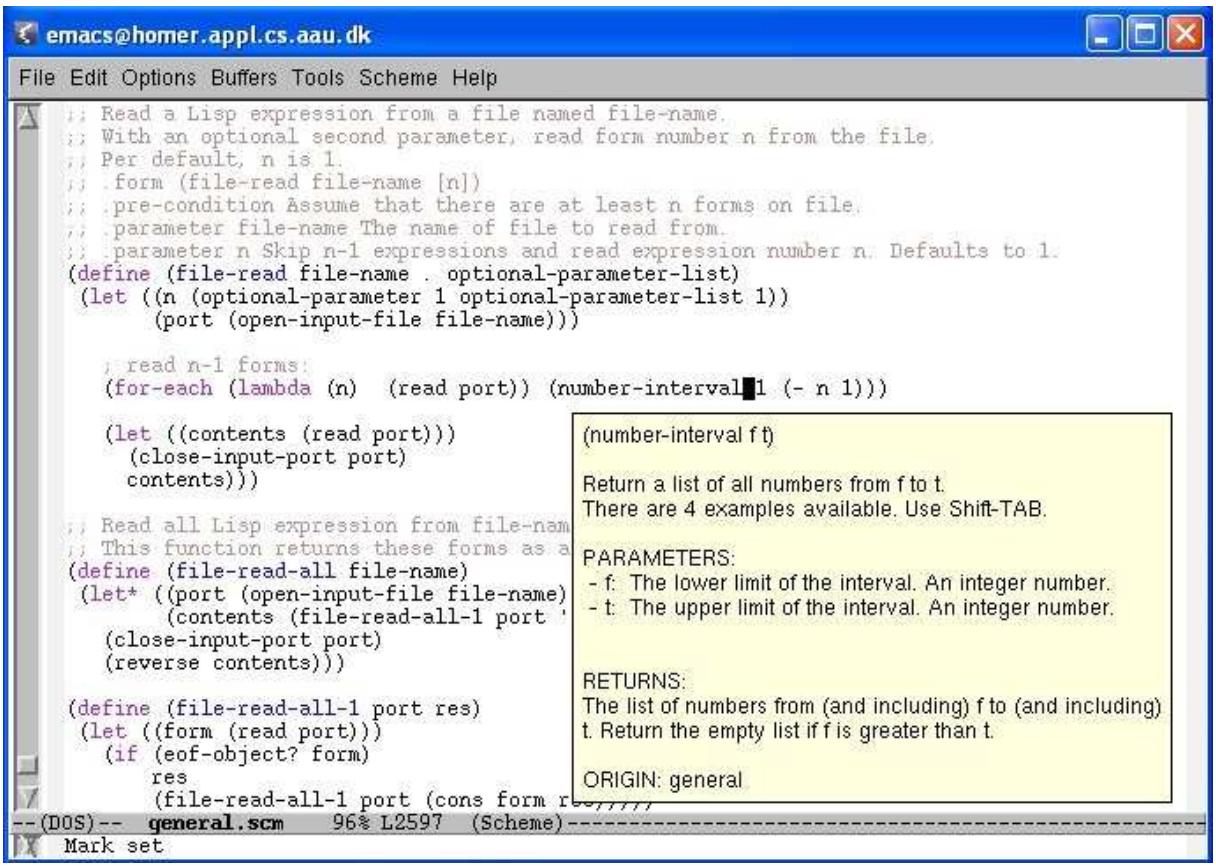

Figure 4. *Instantaneous access to interface documentation from the program editor.*

source program presentation (3) is a central delivery.

Scheme is a programming language which allows fine-grained evaluation of expressions in an interactive shell (also known as a read-eval-print loop). We have augmented the interactive shell with support for interactive unit testing. A separate paper [14] describes how this is done. With use of this facility it is flexible to capture test cases and to populate the test case repository (6). It is, in a relatively straightforward way, possible to document these test cases (9). More interesting, the test case repository (6) can be used as examples which illustrate the use of the abstraction documented by the interface documentation tool. Therefore SchemeDoc (A) accesses the test case repository (6). Each entry in the interface documentation presents relevant test cases from the repository.

Figure 3 shows a couple of interface documentation entries, each of which contains a collection of relevant examples which have been taken from the test case repository (6). As it appears from Figure 3, it is natural to render test cases of pure functions as examples in the interface documentation. Unfortunately, it is harder to come up with similar natural test case rendering of procedures from imperative programming. As can be seen in the figure, the examples augments the interface documentation in a very useful manner. It is often easier, and more tangible, to understand a program abstraction through a number of concrete examples as opposed to abstract explanations which use many words over several lines of text. Moreover, there is a high probability that the examples are correct and accurate at any point in time. The reason is that the examples stem from test cases which are verified automatically against the software via regression testing. The connection between the unit testing tool and the interface documentation tool is - to our knowledge - a novel aspect of the tool-set described in this paper. The closest similar work is Hoffman's and Strooper's FAQ approach [4] which connects testcases with API documentation, but without suggesting a presentation of the test cases as part of the interface documentation. A similar approach is taken by the Python `doctest` facility [2].

The program and documentation editor (D) is GNU

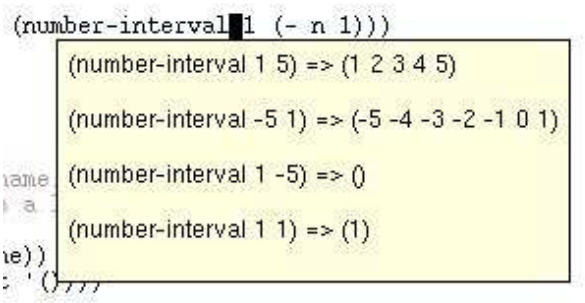

Figure 5. *Instantaneous access to examples from the program editor.*

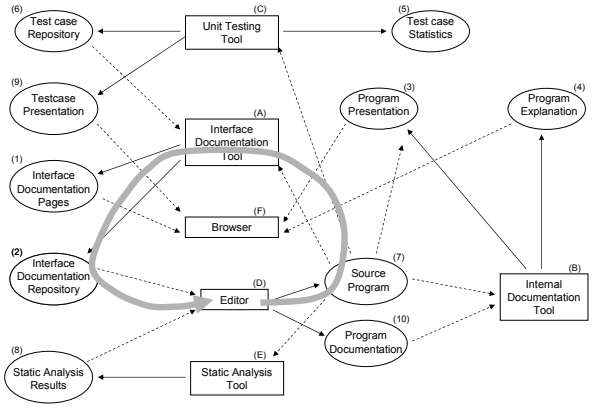

Figure 6. *Interface documentation feedback.*

Emacs - a powerful, classical, programmable text editor which is widely used for program development purposes on all platforms. The program editor is used for production of the source programs (7), which are the primary sources of information for all other tools (A, B, C, E, and F) discussed in this paper. It is possible to consult program documentation (1, 3, 4) in the browser (F) while using the editor (D). It is, however, attractive and productive if some of the program documentation can be brought to the programmer's attention from within the editor. Therefore, the program editor accesses the relevant parts of the interface documentation repository (2). The relevant parts are determined from the so-called major mode of the buffer and from a *dependency clause* located in the preamble of the source file (within a comment). On request, the information in the repository is used for name completion purposes, and for documentation purposes. Hitting the TAB key in the editor completes names if necessary, and if the name is already complete it pops up the relevant interface documentation entry. Figure 4 shows the documentation of the function `number-interval` (the same function used in the example of Figure 3) activated from GNU Emacs. As can be seen from the figure, it is stated that there exists four examples of the function `number-interval`. Hitting the SHIFT-TAB key reveals these four examples from within the editor, see Figure 5.

With access to interface documentation from the program editor, the use of certain names (the documented ones) becomes easy and more secure. Non-documented names in the same or nearby source files, as well as local names, do not enjoy the same support. Based on this observation, we have implemented a static analysis tool (E) within Emacs (D). The static analysis tool is able to extract the missing contextual information (8) and to join it with the information from the interface documentation repository (2).

5. Documentation Feedback

In the two previous sections we have described the connection between program documentation tools and related program development tools. The description has been been given both at a generalized level in Section 3 and at a concrete level - with a given language and given tools - in Section 4. The concrete description in Section 4 addresses an architecture based on separate tools, such as in a traditional Unix setup. The generalized description in Section 3 is a suitable basis for both an integrated development environment (with tightly integrated components) and for a modularized environment composed of separate tools.

We will now identify and discuss some documentation feedback relationships in the map of Figure 1. The purpose is to clarify the roles of program documentation in the flow between the tools. Also of interest, we will point out some non-existing relationships in the map which represent documentation flow problems or tool integration problems in existing environments.

Figure 6, which is a based on Figure 1, illustrates *interface documentation feedback*. As illustrated by the solid grey arrow in Figure 6, the interface documentation is extracted from the source program by the interface documentation tool and brought back to the program editor via the interface documentation repository. In the editor it should be possible to bring up interface documentation of a selected abstraction, based on the editing context. The value of the interface documentation rises when the documentation is fed back to the editor, as opposed to being available in the separate browsing tool. In addition, we hypothesize that the

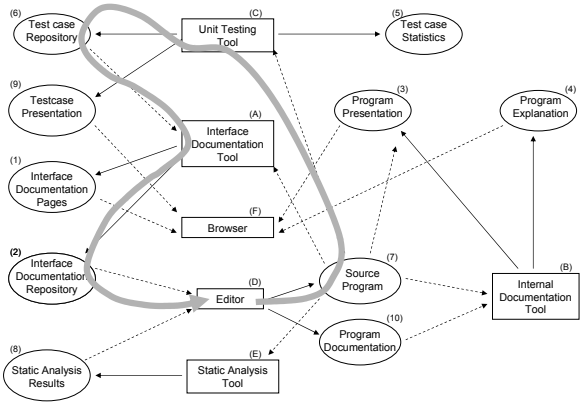

Figure 7. *Interface example feedback.*

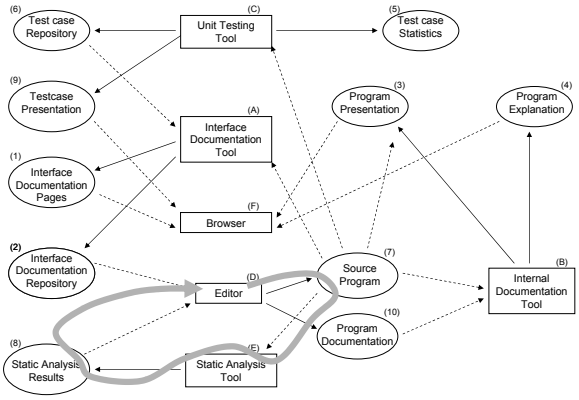

Figure 8. *Static analysis feedback.*

documentation willingness and eagerness of programmers increases as a function of experienced value of the documentation in the developing process.

The turnaround time of the circle shown in Figure 6 is typically relatively long, because the interface documentation repository is only inhabited when it makes sense to (re)process the involved source programs by the interface documentation tool. The crucial constituents of the feedback circle is the interface documentation repository together with the protocol which allows the editor to access this repository. As a future contribution to modularized environments it would be useful to standardize the format of the interface documentation repository and the protocol between the editor and the repository.

Figure 7 illustrates *interface example feedback*. The examples flow from the test case repository to the editor via the processing done by the interface documentation tool. The test case repository is inhabited by the unit testing tool. The *interface example feedback* cycle fuses examples into the editor, based on unit testing efforts. The examples can either be shown together with the interface documentation as such, or they can be presented in isolation from the interface documentation. In an advanced setup, it may be possible to request one or more examples that match the current editing context. In our current implementation of the tools, as described in Section 4, the examples are presented separate from other interface documentation. In other words, it is possible to request the full set of examples at any point in the program where the remaining interface documentation can be requested. In the current implementation of the program editor we do not attempt to select a subset of the examples based on the current editing context.

Figure 8 illustrates *static analysis feedback*. Based on a location in a single source program file *f*, the static analysis tools incrementally extracts knowledge from *f* about definitions which are relevant at the given location. As opposed to the cycle of interface documentation feedback, the turnaround time of the circle in Figure 8 is short. This reflects that - at least part of - the static analysis is carried out at the moment the programmer requests documentation from the program editor. The result of the static analysis is joined with the most recent interface documentation from the interface documentation repository. In that way, the programmer will experience comprehensive name completion and documentation from within the editor.

It is tempting to ask if there exist *internal documentation feedback*, along the lines of interface documentation feedback and interface example feedback. The possibility of internal documentation feedback is indicated in Figure 9. The question is if the internal documentation, as described in Section 2, will be available when needed in the program editor. If the internal documentation is located in close and physical proximity with the documented program abstraction, the problem is easy to deal with. If, on the other hand, the internal documentation is separated from the source program, it is profitable to be able to recall the internal documentation from the program editor.

Finally, in Figure 10, it is questioned if the programmer get sufficient feedback on the unit testing status while situated in the program editor. The feedback we have in mind is (1) the specific tests that involves a given abstraction, (2) the success or failures of these tests, and (3) the coverage of these tests relative to the given abstraction. Given the idea of various kinds of feedback to the editor, the testing feedback hinted in Figure 10 seems natural and attractive. We are not currently aware of any system which supports this kind

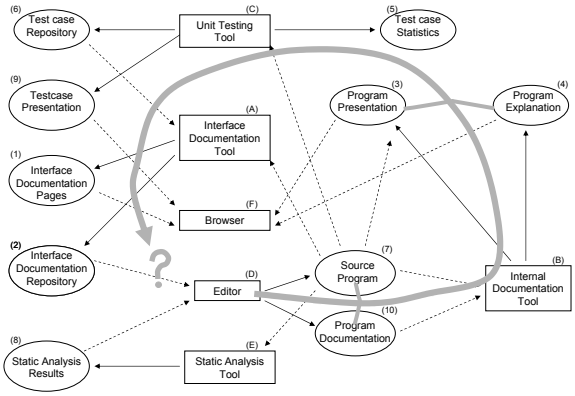

Figure 9. *Internal documentation feedback?*

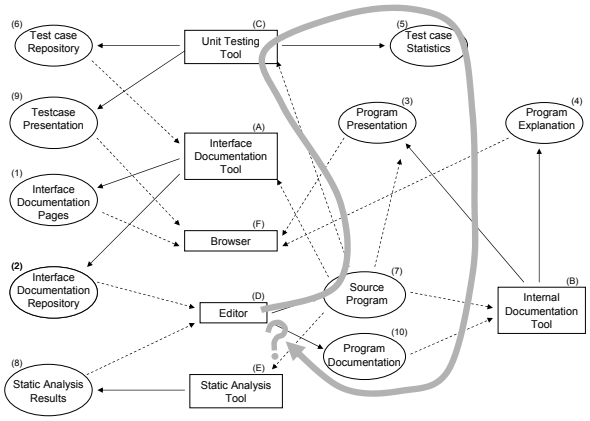

Figure 10. *Testing feedback?*

of feedback.

6. Conclusions

In this paper we have emphasized and mapped the flow of program documentation between tools in a modularized programming environment. *Documentation awareness* among designers of such an environment is necessary to ensure the best possible outcome and "profit" of a programmer's everyday documentation efforts. If a programmer experiences that program related documentation is available and helpful throughout the programming process, it is more likely that the programmer is willing to invest time and efforts in producing high-quality documentation.

In order to ensure an appropriate flow of program documentation in a modularized programming environment it is desirable to standardize both producer and consumer interfaces to documentation repositories. In this paper we have outlined an interface documentation repository and a test case repository. The interface documentation tool and the unit testing tool produce information to these repositories. In the paper we have pointed out that the program editor is a prominent consumer. In future modularized programming environments it will be important to offer standard protocols to documentation-related repositories. It will, in addition, make good sense to use a common XML language as the format of the program documentation repositories. Such a common XML language should be defined independent of the involved programming language of the source code.

The use of test cases as examples in interface documentation is an important and novel contribution of our work. It is straightforward to use and present test cases of pure functions as examples. The reason is that pure functions are fully characterized by the input (via parameters) and the output (via the returned value). It is harder to come up with good example-rendering of test cases in imperative programming. Test cases are ideal for harvesting of examples, because the test cases have proven themselves correct (syntactically and semantically) in the latest round of regression testing.

Interface documentation feedback as well as *static analysis feedback* is already realized between (the intangible) tools of integrated development environments, such as Eclipse and Visual Studio. *Interface example feedback* still seems to be missing in such environments. In modularized environments the various kinds of feedback is less well-developed. The reason for that is partly due to the lack of standard interfaces to documentation-related repositories.

References

[1] P. Briggs. Nuweb, A simple literate programming tool. Technical report, Rice University, Houston, TX, USA, 1993.

[2] P. Doctest. doctest. test interactive python examples, 2009. `http://docs.python.org/library/doctest.html`.

[3] L. Friendly. The design of distributed hyperlinked programming documentation. In S. Frass, F. Garzotto, T. Isakowitz, J. Nanard, and M. Nanard, editors, *Proceedings of the International Workshop on Hypermedia Design (IWHD'95), Montpellier, France*, 1995.

[4] D. Hoffman and P. Strooper. API documentation with executable examples. *The Journal of Systems and Software*, 66(2):143–156, 2003.

[5] A. L. Johnson and B. C. Johnson. Literate programming using `noweb`. *Linux Journal*, 42:64–69, October 1997.

[6] A. Kacofegitis and N. Churcher. Theme-based literate programming. In *Proceedings of the Ninth Asia-Pacific Software Engineering Conference*, pages 549–557. IEEE Computer Society, 2002.

[7] R. Kelsey, W. Clinger, and J. Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, August 1998.

[8] D. E. Knuth. Literate programming. *The Computer Journal*, May 1984.

[9] D. E. Knuth and S. Levy. *The CWEB System of Structured Documentation, Version 3.0*. Addison Wesley, 1993.

[10] D. M. Leslie. Using Javadoc and XML to produce API reference documentation. In *SIGDoc'02*, pages 104–109. ACM, October 2002.

[11] K. Nørmark. Elucidative Programming. *Nordic Journal of Computing*, 7(2):87–105, 2000.

[12] K. Nørmark. Scheme program documentation tools. In O. Shivers and O. Waddell, editors, *Proceedings of the Fifth Workshop on Scheme and Functional Programming*, pages 1–11. Department of Computer Science, Indiana University, September 2004. Technical Report 600.

[13] K. Nørmark. Web programming in Scheme with LAML. *Journal of Functional Programming*, 15(1):53–65, January 2005.

[14] K. Nørmark. Systematic unit testing in an read-eval-print loop. To appear in Journal of Universal Computer Science, 2010. http://www.cs.aau.dk/-~normark/laml/papers/unit-testing.pdf.

[15] N. Ramsey. Weaving a language-independent WEB. *Communications of the ACM*, 32(9):1051–1055, September 1989.

[16] V. Simonis and R. Weiss. Progdoc - a new program documentation system. In M. Broy and A. V. Zamulin, editors, *Perspectives of System Informatics*, volume LNCS 2890, pages 1101–1119, 2003.

[17] D. van Heesch. Doxygen, 2004. `http://www.-doxygen.org`.

[18] T. Vestdam and K. Nørmark. Maintaining program understanding - issues, tools, and future directions. *Nordic Journal of Computing*, 11(3):303–320, 2004. Extended version of [19].

[19] T. Vestdam and K. Nørmark. Maintaining program understanding - issues, tools, and future directions. Presented at 11th Nordic Workshop on Programming and Software Development Tools and Techniques - NWPER'2004, May 2004.

[20] R. Williams. FunnelWeb user's manual. Technical report, University of Adelaide, Adelaide, South Australia, Australia, 1992.