Writing Internal Documentation

Thomas Vestdam
Department of Computer Science
Aalborg University
Fredrik Bajers Vej 7E
DK-9220 Aalborg
E-mail: odin@cs.auc.dk

Introduction

Different kinds of documentation are produced during software development. For example, requirements specifications, design documentation, process documentation, documentation of tests, user documentation, interface documentation, and *internal documentation*, which is the topic of this paper.

Internal documentation addresses and maintains the program understanding and is intended for current and future developers. Current developers document thoughts and rationales behind a program, so future developers can gain understanding of the program, without resorting to reverse engineering (either manual or "automatic"). We know that programs that are documented in this way are likely to be better programs [5] because they have a tendency to be reflected upon more carefully during coding. In addition, maintenance is an important and time-consuming discipline during software development. Often a program lives for a long time, but during this lifetime, the program is subject to changes, such as bug fixes and enhancements. The developers that work on a program may vary during its lifespan. Future programmers such as newcomers to a software project and maintainers will have a hard time understanding the program if no documentation is available. As time passes the understanding of the program will disappear if this understanding is not maintained in some way – the developers will simply forget why things where done as they were. Even the original developers will forget the details of a program if no documentation has been written.

In our research programme we consider internal documentation as indispensable during program development [13]. The time spent on documentation is an investment in ease of program maintenance as well as ease of current and future development.

The production of internal documentation is not easy. Programmers often have a hard time figuring out what, how and when to document. In addition, the documentation should also be readable and usable to other programmers (e.g. maintainers, fellow programmers, and newcomers).

During our work we have observed recurring problems the programmers experience when they are writing internal documentation. In addition, we have observed recurring solutions to the problems. Consequently, a number of patterns presenting the problems and their solutions have been produced. Some of the background for these patterns is based on observations of use of a specific internal documentation paradigm, called Elucidative Programming [11]. Observations have been done in both industrial and educational contexts. Moreover, the patterns are also based on general observations and studies of internal documentation, as well as observations made by others (and presented in the literature). The observations have mainly been focused on internal documentation of object-oriented programs.

Others have made related patterns addressing for example component documentation [20], project documentation management [21], and reader-friendly media for documentation [22]. Such patterns can be combined with the patterns presented here.

A few distinguished meanings

In the patterns, we make use of a few special terms:

A *documentation reader* is a person studying documentation in order to gain understanding of an entire program or just parts of it. Potential documentation readers are, different programmers collaborating on the same program, future programmers taking over a project, maintainers, or the original programmer.

A *newcomer* is a programmer who enters a programming team during or after the development process (e.g. maintainer).

A *program entity* is in most situations a class, but can in general be a small or large logical part of a program. Large parts are for example, a class or a collection of classes, and small parts are for example a method, a block, or even a single statement.

A *programming task* can result in either a new part of the program or an alteration of an existing part (e.g. maintenance).

The pattern structure

The patterns presented in this paper are presented in a simple form, starting with a headline starting the pattern name, and which group the pattern belongs to. The patterns are grouped into:

Structural patterns:	What is important to explain about the program?
Temporal patterns:	Given a programming task, when is it time to write internal documentation?
Maintenance patterns:	How can existing internal documentation be maintained?
Stylistic patterns:	How can internal documentation be structured and presented?

Some of the pattern names stem from our research area, Elucidative Programming. These names may not be obvious to others, but because of lack of creativity, they have been kept. The pattern headline is followed by a section stating the problem the pattern solves, and sections describing forces, solution, and consequences. Finally, each pattern includes a section on examples demonstrating documentation strategies that are instances of the pattern. Some examples are actually anti-examples used as a contrast.

General forces

The following “forces” have a general influence on the patterns in this paper. These should be considered before applying the patterns.

Cost: Production of internal documentation often requires extra effort (i.e. time) of the programmers. This will raise the cost of program development. However, the payoff may be software that is of a higher quality and easier to maintain (e.g. has a longer lifespan). Hence, documentation will pay off in the long run.
Furthermore, the patterns presented here form one approach to doing documentation. Other approaches exist that are quite different, in terms of cost, and more appropriate in

specific software projects. For example, documentation is not really used in Extreme Programming [18].

Quality: If software is documented consistently during development the resulting product is likely to be of higher quality than if no documentation was written. This is because programmers tend to think and reflect more carefully about source code when they write documentation explaining the code. Programmers will, for example, reflect upon - and question - chosen solutions and code design.
However, the more time spent on writing documentation the higher the *cost* becomes.
Yet, internal documentation can be used as a problem indicator. Documentation that does not make sense or is inconsistent may point to an actual problem in source code.

Maintainability: A well-documented system is easy to maintain. The original programmers can use documentation to be reminded of details, whereas newcomers can gain enough understanding of the source code in order to perform maintenance tasks.
However, not all software project needs to be maintained or can easily be maintained by applying reverse engineering tools. Writing internal documentation in such cases is unnecessary (not worth the *cost*).

Difficulty: It is not easy to write documentation, especially for the inexperienced. Using these patterns adds even more work – some of the patterns are very concrete but others are more abstract. Nevertheless, the patterns are useful as source of inspiration. A programmer starting on documentation for the first time should be able to find inspiration in these patterns. Programmers already used to writing documentation (e.g. JavaDoc [3] and code comments) but wish to write even more thorough documentation, harvesting the benefits of maintainability and quality, may also find the inspiration in these patterns. And, experienced documentation writers may still find inspiration in these patterns, but will perhaps need to reshape some of the patterns before applying them.
However, as writing documentation is difficult, there is a risk that the resulting documentation becomes useless. Hence, if the quality of the documentation is not ensured we cannot expect to harvest possible benefits of *maintainability* and higher *quality*.

Dislike: Most programmers actually dislike writing documentation. Therefore, programmers often resort to reverse engineering tools, or interface documentation like JavaDoc [3]. Hence, the potential benefits of *quality* and *maintainability* may not be the reward. Education and attitude amongst software developers are common reasons for the aversion. Furthermore, it is even difficult to persuade programmers to just maintain existing documentation.

Pattern sequences

Instead of giving a graphical road map of the patterns, we will present three possible sequences of applying the patterns. Application of the patterns is iterative as writing documentation is an ongoing process following the programming process.

Sequence 1:
This sequence is useful for novice programmers or novice documentation writers. The ongoing process can be simplified by only using Document Structure Follows Program Structure.

* Decide upon and apply Documentation Templates.

* Separate and Interrelate Documentation and Program.
* Let Document Structure Follows Program Structure and repeatedly Intertwine Programming and Writing Documentation. Consider recording Program History, concepts (Conceptual Writing) or Transverse Issues.
* At fixed points in time, the process is stopped and the documentation is subjected to a Documentation Review. After each review perform Documentation Refactoring and resume the process of writing documentation. Resume writing documentation.

Sequence 2:
This sequence is useful for the more experience documentation writer. The focus is on capturing Transverse Issues. Consider using Document Structure Follows Program Structure if documentation of static program entries and structure is needed.

* Decide upon and apply Documentation Templates.
* Separate and Interrelate Documentation and Program.
* Repeatedly Intertwine Programming and Writing Documentation while recording Transverse Issues.
* During coding, either update the documentation directly or use Documentation Refactoring to maintain the documentation. During Documentation Refactoring, consider Extract Commonly Used Information and concepts (Conceptual Writing). Resume writing documentation.

Sequence 3:
This sequence is also useful for the more experience documentation writer, in need of comprehensive documentation. The focus is on capturing Transverse Issues, Program History, and concepts (Conceptual Writing). In order to make the documentation more flexible Extract Commonly Used Information can be applied during Documentation Refactoring (in step 5).

* Decide upon and apply Documentation Templates.
* Separate and Interrelate Documentation and Program.
* Repeatedly Intertwine Programming and Writing Documentation while recording Program History, concepts (Conceptual Writing) and Transverse Issues.
* During coding, either update the documentation directly or use Documentation Refactoring to maintain the documentation.
* At fixed points in time, the process is stopped and the documentation is subjected to a Documentation Review. After each review perform Documentation Refactoring and resume the process of writing documentation.

In general, when documentation is changed - for example, if Documentation Templates are introduced after production of documentation has begun - Documentation Refactoring can be used to alter/repair the documentation. If Extract Commonly Used Information and Conceptual Writing have been applied, Documentation Refactoring becomes easier. If Documentation Refactoring becomes too difficult to grasp, a Documentation Review can help relive the problem.

01. Separate and Interrelate Documentation and Program [Structural]

Problem: A pile of documentation in one hand and a pile of code in the other - how can one find the documentation relevant for a given part of the code?

Forces: Documentation is often in *physical proximity* of the program, for example in a literate program [5], as JavaDoc comments [3], or ordinary code-comments. This physical proximity makes it hard to ignore the documentation. When documentation is not in the proximity of the source code, it is often ignored and forgotten, and program and documentation become inconsistent.

However, the program often seems to disappear in the documentation [10]. For example, in literate programs the actual code is scattered throughout the documentation making it difficult for collaborating programmers to develop and maintain large programs. In addition, code comments also have a tendency to drown the program.

If documentation and program are separated, the program can be kept clean of "foreign" elements, such as long bodies of text or documentation specific elements and syntax. Furthermore, the program can be studied as it is and code visualisation-, debug-, or reverse-engineering tools can be used without having to extract the program from the documentation (or visa versa).

However, if documentation and program are separated good references between the two must be provided. Otherwise, documentation and program will become inconsistent.

Furthermore, as documentation is often fragmented into several documents a good reference mechanism is needed in order to keep these fragments connected (see also Extract Commonly Used Information and Document Structure Follows Program Structure).

Solution: *Separate documentation and program, and use typed links to provide selective and mutual navigation between documentation and program.*

Hypertext is a natural presentation media for both documentation and program. The documentation may consist of several documents, and typed hyperlinks can provide both selective navigation and *navigational proximity*:

- Provide typed hyperlinks from the documentation to the program.
- It should be possible to link to syntactical elements or source markers (i.e. marks placed in the code by a programmer) in the program.
- Provide hyperlinks from the anchored links in the program to the respective sources in the documentation.
- Use the names of the syntactical elements as anchor names instead of more arbitrary words. For example, a hyperlink like "the method is found here" may help the reader when reading the documentation, but the *origin* of a link becomes difficult to find when following a link from the program to the documentation.

– Finally, provide typed hyperlinks for links going from documentation to documentation.

As an exception, consider placing documentation such as interface documentation in relevant places in the program.

Consequences: Flexible navigation between documentation and program is provided, hence putting documentation and program in *navigational proximity*. The documentation can address specific program entities - simply by "pointing".

The types on the hyperlinks help the documentation reader to sort out which links are relevant in a given situation. Documentation Templates gives an example of link types, and can be used to standardise how links should be made.

However, physical proximity is lost. Furthermore, the documentation structure tends to become very fragmented and consequently difficult to maintain. It can even become difficult to get an overview of the documentation. These problems can be relived by applying Documentation Review and Documentation Refactoring.

Examples: Elucidative Programming [12] is a typical example of this pattern. Program and documentation are kept separate.

02. Document Structure Follows Program Structure [Structural]

Problem: Getting started on writing documentation as well as structuring the documentation is not easy, especially for the inexperienced documentation writer - how can one get started writing and structuring documentation?

Forces: A good starting point for a documentation reader is descriptions of the individual program entities and their static relations. This gives the reader something to hold on to as, ideally, every program entity can be associated with some kind of description in the documentation. Such *entity descriptions* can for example include explanations of the purpose of the entity, important methods, data structures, algorithms, or the services the entity provides or requires [9]. In addition, references can be provided between entity descriptions according to the static relationships of the program. It is for example, often necessary to understand a base class in order to understand its sub-classes.

Although the dynamic relationships are more difficult to deduct from the code than the static relationships, they are not of much use when trying to gain an overview of a program (see also Transverse Issues). In this situation, the documentation reader is more interested in getting an understanding of where entities are located in the program and their immediate purpose.

However, the documentation may turn out to be a mirror of the program. This may not be satisfying as program understanding sometimes is detached from the program structure. Documentation is intended for humans and not computers [5].

Solution: *Let the documentation structure follow the program structure when presenting static program structure and program entities.*

Create documents addressing the overall structure of the program, for example for each module or component. Give these specific names; such as “Overview of XXX” or place them in a specific part of the documentation.

For each “larger” program entity (e.g. classes), create a document and:

- Give an overall description of the entity. Consider including descriptions of important methods, data-structures, algorithms, interfaces, or even test descriptions.
- If the given entity is involved in inheritance, association, or aggregation relationships, add references to documentation of the involved entities.

During a design-phase diagrams are often created, these can be very useful in the documentation in order to communicate the program structure (see also [23]). This also includes use case- and architecture- diagrams.

Consequences: Good descriptions of the overall structure of the program can be a good starting point for a documentation reader. Such descriptions can also maintain the overall program structure during collaborative work. As such, this pattern can effectively be applied when programmers are unsure about how to structure the documentation.

When the documentation structure follows the program structure, and Separate and Interrelate Documentation and Program is applied, documentation of different entities becomes easy to locate, and the reader quickly starts forming a mental image of the static design of the program.

However, some documentation needs to be detached from the program structure in order to be comprehensible. Most often, this is Transverse Issues, or can simply be handled by applying Transverse Issues.

If the program structure changes, so must the documentation structure (see Documentation Refactoring).

Examples: Object oriented documentation is an example of how useful properties can be gained by structuring documentation in the same way as the program (i.e. an isomorphic structure) [15].

CASE tools such as Rational Rose are often used to produce and organise entity documentation that follows the program structure (e.g. design diagrams).

As a contrast, in Literate Programming [5] the documentation drives the programming as the program to follows the documentation structure.

03. Transverse Issues [Structural]

Problem: It is not feasible to document all aspects of a program, but which aspects should at least be considered?

Forces: It is often hard to understand the dynamic interaction between program entities when studying code (e.g. see [2]). For example, user interaction often involves several parts of the code by spawning actions, events, or calculations (i.e. method calls) that *transverse* the program structure. Figure 2 is a simple illustration of this. For example, the classes C_1, C_2, and C_3 stem from an interface component, a function component, and a model component respectively. When a user issues a command, C_1, C_2, and C_3, work together on solving a complex task that involves, retrieving information from a storage, doing calculations, and presenting a result. With a few sentences, for example explaining which methods are invoked when and where, the *transverse issue* T can bind C_1, C_2, and C_3 together. The transverse issue T simply describes how C1, C_2, and C_3 collaborate on solving the given task.

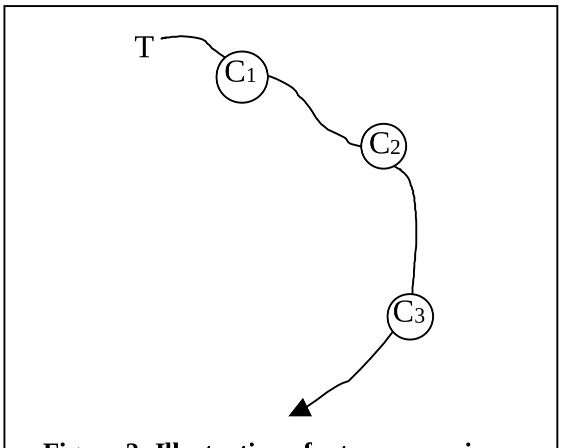

Figure 2: Illustration of a transverse issue

If the programmer has no understanding of relationships across a program, it becomes difficult to predict how a change somewhere in the program may affect other parts of the program [16]. Such information is difficult and time-consuming to dig out manually. For example, consider distributed systems where a few lines of code in one place may affect several other parts of the system.

However, it is difficult to predict which transverse issues will be useful to future documentation readers [2]. It is certainly not feasible to document all possible relationships across a program in order to *guide* documentation readers through the entire program.

Yet, some transverse issues are actually often documented. For example, design patterns [4] describing collaboration between groups of classes and methods, or descriptions of change that affect several parts of a program.

Solution: *Describe and explain program relationships such as collaborating program entities or code dependencies. Relate such transverse issues to relevant program fragments.*

Referring to Figure 2, the transverse issue T is of interest if:

– *C_1, C_2, and C_3 are collaborating.* For example, as a design pattern or are together performing a certain task. Only include descriptions of tasks that are essential to the program.
– *C_1, C_2, and C_3 depend on each others.* For example, the existence of one program entity is depended on the services of another program entity.
– *C_1, C_2, and C_3 are affected by a certain change.* For example, the respective program entities are extended to accommodate new functionality or conditions.

For each transverse issue create a document including descriptions of the transverse issue and references to the involved program entities (using their names, see Separate and Interrelate Documentation and Program). A transverse issue may also include references to other relevant documentation (see also Extract Commonly Used Information).

Consequences: A key to the understanding of a program is an understanding of the communication within the program. Descriptions of the communication enable the documentation reader to dig out rationales behind it, giving an understanding of why things work together as they do.

Requirements and tests can be seen as transverse issues and if described in the documentation, they become visible to the programmers collaborating on developing the program. Furthermore, if the requirements are related to affected program entities, maintainers (or others) can track the effect of the requirements.

However, *Non-transverse issues*, such as static relationships, are also important, but are not covered by Transverse Issues. Capture these with Document Structure Follows Program Structure.

Examples: DocSewer uses *documentation threads* [6], to guide a documentation reader to relevant program entities collaborating on some task. For example, the transverse issue T in Figure 2 could be a documentation thread.

In Literate Programming [5], on the other hand, descriptions of transverse issues are not commonly found as all documentation about a given program fragment is written in the context of that fragment.

04. Extract Commonly Used Information [Structural]

Problem: The documentation often becomes a mess because explanations of program entities become long and complex. It becomes difficult to find relevant explanations, and consequently difficult to update - how can the documentation become more flexible?

Forces: Often several *aspects* of a given program entity deserve attention in the documentation. For example, if using Document Structure Follows Program Structure aspects like overall description, explanation of important methods and algorithms are often added to the documentation. For example, class C in Figure 3 is a database handler, and A_1 is a description of how it wraps complex SQL statements by providing methods for these. A_2 is an overall description of the class and A_3 is a description of how the class has been tested.

A collective presentation of such aspects often leads to a large body of text, where the different aspects are entwined (see left side of Figure 3). Entwined aspects are difficult to separate making the text complex and difficult to read. In addition, this complex text must be reviewed when the given program entity is changed.

If instead aspects are extracted and explained separately (see right side of Figure 3), distinction becomes easier, and aspects become less *complicated* to read and maintain.

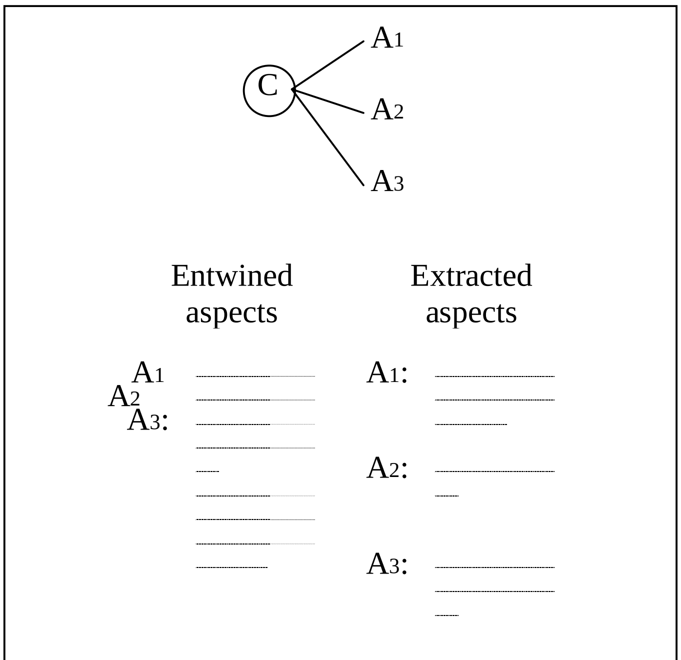

Figure 3: Two ways of presenting different aspects

However, this may lead to a very fragmented documentation structure, where all the aspects of a given program entity becomes difficult to isolate. Hence, the documentation becomes difficult to read and structure.

Yet, references between documentation parts often lead the reader to large bodies of text where the aspect referred to is difficult to isolate.

Solution: *Different aspects of a program entity should be explained separately, if these aspects are important in more than one context.*

Extract an aspect if it provides commonly useful information such as:

- General descriptions of program entities. Such as the purpose of an entity, rationale(s) behind an entity, or descriptions of important methods.
- Detailed descriptions of program entities. Such as descriptions of special algorithms or descriptions of tests performed on the entity.
- Overview descriptions. Such as descriptions of program structure and organization, or descriptions of specific protocols used.
- A description or explanation that a Transverse Issue refers to.

Give extracted aspects an identification mark, such as a title. Bring the aspect in *navigational proximity* (see Separate and Interrelate Documentation and Program) of both relevant program entities and other relevant documentation. References should for example be added if an aspect depends, mentions, uses, extends, or excludes another aspect.

If using Document Structure Follows Program Structure, consider collecting aspects of program entities as different sections in the respective "entity" documents.

Consequences: By applying this pattern the programmer provides easily identifiable descriptions of different aspects of the program. It is a matter of *separation of concerns* [1], each commonly used aspect of the program is described in exactly one place only and nothing else is described in that place. The documentation reader can search through these descriptions when looking for something specific.

Extraction of aspects may also lead to code refactoring. Complex documentation may point towards a need for simplification of the code. In other words, writing documentation often points out defects and problems in the code.

By describing the aspects of a program entity, the programmer may start thinking more about which aspects of a given program entity could be of interest for others. Furthermore, it is possible to create templates suggesting which aspects should be addressed for specific types of program entities (see Documentation Templates).

Examples: The documentation produced using Elucidative Programming in Java [12] is a fragmented hypertext structure making it possible to Extract Commonly Used Information.

DocSewer uses *documentation threads* [6] to sew different aspects of a program together, hence producing a linear presentation of the selected aspects. Documentation threads are a means for presenting a collection of aspects of use in a specific context (e.g. a transverse issue). It is therefore important that the different aspects are easy to extract.

In Literate Programming [5], all relevant aspects of a given program entity are written collectively in the context of the program entity. This has proven very useful when publishing programs (e.g. as a book).

05. Program History [Structural]

Problem: Often programmers inadvertently try solutions that have already been proven useless because they have forgotten previous encountered problems - how can this be avoided?

Forces: In order to understand the current state of a program it is also useful to know what has been tried, rejected, or changed as well as reasons for this. Such *program history* can help a documentation reader to gain in-depth understanding of why solutions where chosen and others were rejected. As time passes recollections of previous solutions will disappear, and future programmers risk trying solutions already proved useless. For example, if technicalities prevent usage of a preferred storage (e.g. database) this could be noted.

Future maintainers (or newcomers) will have many of the same questions as the original programmers about why a certain solution was selected and others rejected [1]. It is desirable if answers to such questions can be found in the documentation. Furthermore, if the forces working against a desired solution are documented, the solution can be re-tried if these forces are eliminated during program evolution. For example, technicalities preventing usage of a preferred storage may be eliminated by the release of new drivers.

However, recording program history is time consuming and it can be difficult to decide when an alternative is relevant. The result may simply be that *too* much useless program history is collected in the documentation.

Lack of time often causes programmers to choose “quick and dirty” solutions, but it is not worth the effort to actually document such solutions. However, programmers often know of a “better” alternative when a bad solution is being chosen.

Often programmers think they can cover program history through version control, but the change-descriptions stored in a version control system are not easy to trace back to the actual parts of the source code that were changed. The problem becomes even worse if a change involves several source files.

Solution: *Record the choices made during program development in the documentation. This includes relevant solutions that have been tried but proven useless. In addition, consider recording relevant alternatives such as solutions that are more flexible, optimal or even more aesthetic than the chosen solution.*

Describe selected solutions and explain the rationale behind the choice. For example, explain why choosing a specific architecture, storage, algorithm, or protocol.

Consider recording solutions that have been tried and proven useless or erroneous, if:

– The solution illustrates a conceptual problem - “This should work, but it doesn’t”
– It is a good solution and the error symptoms have a potential future remedy - “A better solution would be to do *this*, but then we need to get rid of *this* problem.”

Keep the code of tried solution separated from the “real” code. Describe it and explain why it was useless/erroneous, such as conditions, error symptoms or conceptual problems.

Finally, consider recording an alternative if it, presumably, can lead to a better solution of a problem at hand – depending on the nature of the problem and the chosen solution.

Remember to be selective about all records of program history; otherwise, too much documentation will be accumulated.

Consequences: Records of the history of a program can give a documentation reader a deeper insight on the program than insight obtainable from documentation addressing the current state of the program. This deeper insight includes explanations of how problems were solved, as well as why relevant alternatives were not used. This prevents future programmers from trying solutions that are known to wreck the program. Furthermore, recorded alternatives can provide suggestions on how the program can be extended (e.g. to become more general or comply to other requirements) [16].

In addition, documentation of the programming process itself is to some degree maintained by explaining how and why things are done.

During program evolution, history will disappear in the big sea of changes. Tried solutions will become obsolete, and some history may end up addressing non-existing program entities. Documentation Reviews can detect this, and Documentation Refactoring can weed out obsolete history (and *stored* code), as well as remedy a poor structure.

Examples: Configuration management systems often include tools for storing different versions and variants of the same program (or documentation). This enables programmers to subsequently retrieve and compare variants of a program.

06. Conceptual Writing [Structural]

Problem: Different programmers often use the same words for different things and different words for the same thing – how can we avoid that the documentation inherits such inconsistencies?

Forces: New terminology is often invented during the development of a program, and program understanding is often discussed using this terminology [1]. This special terminology is also often used in documentation. For example, in explanations of *concepts* such as the program itself, special system set-up, special names denoting specific parts of a program, or special functionality. The use of this special terminology makes communication between programmers prompt and precise.

However, this terminology may not be obvious to others. Documentation containing unknown terminology is hard to understand for a newcomer. Even the original authors, if they are available, may not be able to recall the meaning of old terms.

In addition, programmers invent special terminology about the development process. For example, terms are invented for special activities, program constructs, working methods and even special decisions such as requirements to the program.

Even though such terminology do not directly make documentation harder to understand a newcomer may find it hard to understand the practice of the development process within an existing programming team. For example, special code conventions that must be upheld in order to ensure the quality or robustness of the code can be an important issue during development. Such conventions can also be seen as a part of the program terminology.

Even though terminology is defined when used, the same terms may end up being defined repeatedly or in several variations. Furthermore, programmers collaborating on a program often use different terms for the same concepts, or the same term for similar yet distinct concepts. This introduces inconsistencies in the documentation and consequently makes the documentation harder to understand.

Solution: *Special terminology used about the program must be defined in a central part of the documentation.*

Create a list of *concepts* (a glossary), for example as a single document or a collection of documents in an appropriate place somewhere in the documentation structure. Fill the list with terms from the program terminology along with their definitions.

In general, use the defined terms when writing documentation, and provide references from usage of terms in the documentation to the list of terms.

Consider adding terminology about the development process, such as code conventions, special requirements and code constructs.

Consider defining terms before writing documentation (See also Intertwine Programming And Writing Documentation).

Consequences: This pattern helps programmers to use the same words for the same concepts. In addition, the documentation becomes more precise as the program can be discussed and explained in a well-defined abstract terminology. The documentation may even become more compact, as terms need not be explained repeatedly. Newcomers can study the list of concepts in order to gain a basic understanding of the program terminology.

In the object-oriented paradigm, important program terminology is often represented by class names. Placing these names in the list of terms along with their definition can provide a documentation reader understanding of the fundamental concepts of the program. In general, explanations of concepts that stem from an application domain can be very useful in the documentation.

Examples: Conceptual Writing is to some extend used in PAS (Partitioned Annotations of Software) [9]. When a component (e.g. a class) is documented using PAS, a partition (a section) is dedicated to descriptions of the concepts behind the component.

Design patterns are known to provide programmers with a common vocabulary. In addition, WikiWikiWeb is an example of how people/programmers can collaborate on defining concepts, and consequently use the same terms for the same concepts [8].

07. Intertwine Programming And Writing Documentation [Temporal]

Problem: Documentation must be written, but when is it time to write it?

Forces: Writing documentation explaining the “big picture” before the code is written helps the programmer gain insights into the actual code design. Such insight can help reduce complexity of a code design or simply make it clearer.

When designing code programmers often form a *plan*, mentally check the plan, revise the plan, and then write the actual code. If problems are encountered and the overall plan has been documented the programmer can easily go back and question the overall plan, even after a longer period.

However, it is not always possible to predict the design details at an early stage. When a code design meets reality, revisions are often needed. If code design is thoroughly documented beforehand, the documentation is subject to constant revision. In addition, some programming tasks can even be considered as trivial and are not expected to introduce new concepts (see also Conceptual Writing) or complexity to the program (e.g. a bug fix or a simple extension). Such tasks are simply best solved by doing, and programmers often just wish to get the programming task done.

Yet, if the program is written without prior documentation of the code design, documentation may never be written. Programmers often dislike writing documentation – and if the coding is completed, why bother about documentation?

When programmers have lived with a program for a long time, they will take decisions about - and rationales behind - the program for granted. Documentation written after or near the end of the development process therefore tends to only include immediate details of the program that the programmers think they are going to forget. Such documentation is mainly of use for people who already know the program well, whereas newcomers will find it useless [1].

As time passes, the program understanding will disappear. Even the original programmers will forget the rationales behind the program. If program understanding is not maintained somehow, the program itself may degrade during maintenance [16].

Solution: *Document the overall code design before coding, and document the design details after the coding. In addition, maintain relevant issues that arise during coding by alternating between writing code and documentation.*

Writing documentation should be considered a cycle starting before the code is written and is concluded after the code has been written. The cycle includes documentation written:

– *Before* coding: Describe the context of the programming task, and sketch the code design. In addition, record introduction of new concepts (see Conceptual Writing) and Transverse Issues.
– *During* coding: Keep both coding and documentation on track by recording relevant issues that arise during the programming process. Relevant issues can be: new program entities (see Document Structure Follows Program Structure), rationales behind chosen solutions, special conditions in the code, special requirements, ideas, Transverse Issues, new concepts (see Conceptual Writing), and alternatives (see Program History).
– *After* coding: Review new code systematically in order to recall details, and conclude the documentation by synchronising it with the final code. Make sure that the overall problem that has been solved is described and include explanations of how it was solved. Consider recording new concepts (See Conceptual Writing), Transverse Issues, and new program entities (see Document Structure Follows Program Structure).

Consequences: Documentation written beforehand helps ensuring careful and thorough thoughts behind selected solutions. The time invested in writing such documentation pays of during future development and maintenance - and may even affect performance and correctness of the program.

Documentation written during coding captures the current state of the program. If the programmer alternates between writing code and documentation, the programming process is likely to be reflected in the documentation. This is useful if programmers are to finish the work started by other programmers, or themselves during a long-term project.

When coding is completed, the programmer knows how a given problem was solved. Through *after rationale*, the programmer can write good explanations of the completed code. However, if bugs are first fixed and then documented, the documentation is more likely to degrade over time, than if documentation is written before the fix [17].

However, when program understanding is documented after coding, some of the program understanding disappears simply because the programmer has forgotten some of the details. Furthermore, the documentation may tend to be perfunctory.

Examples: Writing documentation beforehand is in general considered as good common sense, but is rarely used due to programmer's bad habits (or perhaps it is in their nature). This pattern can be (and often is) exercised using Literate Programming [5] or Elucidative Programming [7]. Furthermore, JavaDoc [3] is also used to specify interfaces before coding, as well as maintain these during coding [19].

Alternatively, in Extreme programming [18], documentation is sparsely used – if used at all. Instead, the programmers try, as an ideal, to keep the code simple through frequent refactoring (when changes are made to the program). This helps ensuring that the source code can be understood by simply reading it. Moreover, tests are written before code is written, and are used as "specifications" of what the code should do.

08. Documentation Refactoring [Maintenance]

Problem: The program evolves, and documentation gradually falls behind. Keeping the documentation up-to-date according to every change is simply not feasible – when is it time to update the documentation and how can this be done without constantly changing the entire documentation?

Forces: When a documentation reader studying documentation encounters documentation that addresses code that does not exist, faith in the documentation is lost. This is also the case if code is obviously different from the explanations in the documentation. Such inconsistencies will frequently occur if the documentation is not maintained.

However, as a program grows in size so will the documentation. When the program changes, the programmer may be forced to read larger portions of documentation in order to identify how and where the documentation must be changed accordingly. This is very time consuming, and it is difficult to be sure that all relevant places in the documentation have been found. Programmers therefore often find it tiresome as well as difficult to maintain documentation, as this requires constant restructuring and revision of the documentation – even when dealing with small program changes.

Instead, it is tempting to only describe the actual changes made to the program in relevant places in the documentation. For example, change descriptions can be appended to existing descriptions of a program entity when it is changed.

However, this will result in documentation that becomes difficult to read, as different change descriptions must be put together in order to understand a certain program entity or aspect (e.g. Transverse Issues). Furthermore, some descriptions of change will eventually address entities in the program that do not exist any more or will simply invalidate older descriptions.

Documentation is subject to many changes during development, and eventually it is likely to become unstructured and incoherent. For example, begin to contain much outdated, obsolete, and invalid information. Documentation Reviews can be used to detect such problems.

Solution: *Refactor the documentation on a regular basis either by 1) providing updates to the documentation based on a hot list of changes that need immediate propagation, or 2) revise the entire documentation.*

During coding, maintain a list of changes made to the program. This *hot list* should be appropriately placed in the documentation. Entries in the hot list can for example consist of a description of the change, ideas to improvements, results from test runs, and in general things to remember. Consider constructing templates for list entries.

If documentation that may be affected by a change is discovered during coding, this should be uniformly marked throughout the documentation.

When it is time to update the documentation, the entries in the hot list are dealt with one by one. For example, consider updating the documentation once a week or use the number of entries in the hot list as an indication of when it is time to update the documentation (say, when the hot list contains 10-20 entries).

When the documentation begins to contain much unstructured or outdated documentation then refactor the documentation by *revising* all the existing documentation. This can be done on a regular basis, or as requested by a Documentation Review. In addition, programmers that use the documentation actively during work will often discover problems with inconsistencies, outdated and unstructured documentation.

A revision should ensure that the documentation is comprehensible and consistent with the program. In addition, such refactoring can be used to improve navigation by bringing the *right* documentation in *navigational proximity* (see Separate and Interrelate Documentation and Program) of the *right* code. Hence, revision of the documentation includes:

- Deleting useless, duplicated, and outdated documentation.
- Revision of existing documentation - possibly resulting in rephrasing or addition of documentation - e.g. according to new templates (see Documentation Templates).
- Tying together incoherent documentation.
- Going through all affected documentation systematically to ensure that all references are correct.

Consequences: Updating documentation is time consuming, but the resulting documentation has a high degree of consistency with the program. Even though it is not always possible to update everything, tools can be created to assist the programmer in detecting parts of the documentation that are affected by a change in the program or detecting references that has become invalid.

When a wrecked documentation structure is refactored, it can be salvaged and cleansed of outdated, useless, and incomprehensible documentation.

Intuitively it seems to be easy to come up with descriptions of change. It is a matter of putting a few words to what have just been changed in the source code. It can therefore be easy to persuade programmers to write such documentation. However, updating the existing documentation may still be challenging, but if the activity is part of the development process the programmer is inclined to get it done.

If documentation of the program evolution is important, the changes made to the program must be recorded in order to maintain what actually happened during the implementation. Hence, the hot list can be used to maintain Program History.

Examples: In Literate Programming [5], the code is in physical proximity of the documentation and it is not difficult to identify where in the documentation changes should be made. Hence, the documentation is often changed immediately after the code has been changed.

In Elucidative Programming in Java [12], special documents containing change descriptions are created when the program is changed, or typed hyperlinks are used to find the parts of the documentation that need updates corresponding to a program change.

In the open source community, change-logs are often used to keep track of changes. In general, a hot list can be seen as a change-log.

In extreme programming refactoring is often done after a program change has been successfully implemented in order to keep the code simple (i.e. easier to understand) [18] - the same can apply to documentation.

09. Documentation Review [Maintenance]

Problem: Documentation varies from programmer to programmer in terms of quality and quantity - how can the overall quality of documentation be evaluated and ensured during coding?

Forces: Most programmers lack the experience in writing internal documentation, and only few programmers actually write documentation. This may result in inconsistent and prose documentation of no real help to maintainers or collaborating programmers.

Furthermore, different programmers write documentation differently. Even if Documentation Templates are used, the quality of the actual contents of the documentation may vary a lot from programmer to programmer.

To make things worse, some programmers are so ridden with bad habits that they will simply not get around to writing documentation.

In addition, documentation is often incomplete, for example because the programmer starts writing some documentation, but is later detained by other activities. Often the incomplete documentation is forgotten and never completed.

However, some documentation is only used by the original programmer and only needs to be comprehensible to that person. Moreover, if future programmers can communicate with the original programmer (e.g. is still in the same programming team) it is often not only easier but also better to do so, instead of reading through piles of documentation [16]. Hence, incomplete documentation becomes less of a problem.

Yet, if the original programmer is indisposed and never documented parts of a program, for example considered essential program entities/aspects as non-essential, then future programmers will not find much help in the documentation.

Furthermore, over time Documentation Refactoring is likely to result in a lot of changes and updates that eventually will wreck the documentation structure.

Solution: *The documentation should be reviewed regularly in order to ensure the adequacy of the documentation. Documentation reviews can for example be performed in connection with code reviews.*

Evaluate whether the:

- Documentation coverage is adequate for specific needs.
- Documentation is comprehensible.
- Documentation is properly structured.
- Documentation is consistent with the program.

The reviewers should also give pointers to the programmers, on how they can improve the documentation and their documentation “skills”, for example, in order to break bad habits.

Consequences: By performing Documentation Reviews, the quality of the documentation can be ensured. If the quality of the documentation is not high enough, the programmers are asked to improve the documentation.

Documentation reviews are a good way of providing programmers inexperienced in documentation writing with feedback and suggestion on how to improve the quality of the documentation.

However, documentation reviews are time consuming, and if the reviewers do not approve the documentation, programmers must spend even more time on writing the documentation.

Examples: Code reviews are generally used in order to ensure quality of code, and Documentation Reviews can essentially be structured and performed in the same manner.

10. Documentation Templates [Stylistic]

Problem: Different programmers have different styles when writing and structuring documentation. Depending on which programmer wrote what in the documentation, some aspects of the program are well explained whereas others are not - how can we ensure some kind of uniformity of the documentation?

Forces: When several programmers are collaborating on documentation (and program) standards are quite useful in order to keep the documentation uniform. Such standards can specify which topics should be covered in the documentation. Standards can also give documentation readers some knowledge of what to expect of the documentation and where to look for specific information.

In addition, documentation is written differently from programmer to programmer. Some programmers have their own idea of how to write the documentation, whereas others have a difficult time getting started – "which topics should I cover in the documentation"? Documentation standards in form of guidelines and checklists can help programmers getting started as well as ensure that important topics are covered.

However, guidelines and checklists take away the freedom to write and structure documentation as the individual programmer prefers.

Yet, there is often little room and use for "individual" documentation if programmers are working in collaboration on a program.

In addition, by having guidelines and checklists programmers can save effort, as they do not need figure out what to write about. However, programmers may just end up filling out forms without thought.

Solution: *Identify overall aspects that should (or can) be addressed in the documentation and create templates for these.*

Overall aspects include: rationales, concepts (see Conceptual Writing), bug reports, change descriptions, entity descriptions (see Document Structure Follows Program Structure), Transverse Issues, or requirements.

For each template, specify to which kind of aspect or part of a program it applies.

Relationships between templates should also be specified. For example, specify which kinds of templates can be related/linked to each other.

A template should specify topics to be covered. For example, a template for a concept can include sections describing: status (e.g. completed, in progress or new), keywords, author, creation date, last updated, title, context, description or even an abstract of the concept.

A template can also include examples of use or general guidelines. Examples of use can be used to ensure that documentation is written in a "uniform language". Alternatively, Documentation Reviews can be used to ensure use of uniform language.

Consider specifying how references should be created and used in documentation. For example, if using Separate and Interrelate Documentation and Program the different types of links and their application can be specified. For example, some links are created because a specific program entity is briefly *mentioned*, whereas others are created because the entity is *described* in detail.

Consequences: This pattern ensures uniformity of the topics that should be covered when addressing certain aspects or entities of a program. This can ensure documentation “completeness”. A consistent and logical documentation model - accommodating specific needs within a specific programming team (or company) - can be created.

Intuitively, if specific guidelines are given to programmers, they will find documentation much easier to write. The overhead of figuring out what to write is reduced, and chances of documentation actually being produced are improved. However, if a wide variety of templates exists selection of appropriate templates becomes very difficult. Furthermore, programmers may still find it difficult to figure out how to author the actual contents of a template [9]. Consider using Documentation Review to ensure right usage of templates and the quality of the actual writing.

Note that, there are contexts and situations, for which it is not easy to create good and general applicable templates, guidelines, or checklists.

Examples: Partitioned Annotations of Software (PAS) [9] are used to describe components (e.g. classes) from different points of view (aspects). PAS can be seen as predefined templates defining topics to be covered. PAS is based on the existing structure of a program.

Conclusion

Programmers accustomed to writing internal documentation will probably recognise some of the patterns presented in this paper. The patterns are a combination of the common sense, common practise, and ideals used by programmers writing and structuring internal documentation. The patterns are based on our own observations of programmers producing internal documentation, as well as observations made by others. However, the list of references, only includes a few of the literature resources used as basis for the patterns (if all were listed, the list would become very long).

Some of our observations stem from use of Elucidative Programming [11], which have pushed the patterns in a certain direction. This type of internal documentation is based on hypertext and separate documentation and program (see Separate and Interrelate Documentation and Program). Transverse Issues, Documentation Templates, Conceptual Writing, and Extract Commonly Used Information are patterns commonly used when writing documentation the elucidative way [7] [12] [14].

Based on discussions at EuroPLoP 2001 and recent work we have become aware of new possible patterns. One of these addresses how to allocate time for writing documentation. Other possible patterns address how to make code walkthroughs in libraries and frameworks, or how internal documentation can be linked to use cases (e.g. [18]).

Future work also includes introducing the patterns presented here in development projects and comparison of the resulting documentation with documentation produced in projects not using the patterns. This should help evaluating whether patterns make a difference when programmers produce internal documentation.

Acknowledgements

Most of the patterns presented in this paper have been discussed in the Software Quality through Documentation of Program Understanding research programme (DOPU), at Aalborg University [13]. The people who participated in these discussions are acknowledged for their help in identifying, reviewing, and formulating the patterns. In addition, thanks to Andreas Rüping for his comments during the shepherding of this paper – the patterns became a lot clearer and concrete on his account. And thanks to the all the reviewers at EuroPLoP 2001, for their comments.

Furthermore, students from Aalborg University have participated in the experiments (one example can be found in [14]) – their participation has been very rewarding. In addition, the graduate students Max R. Andersen and Claus N. Christensen are acknowledged for sharing their observations of both production and use of internal documentation in an industrial setting [7].

References

1. Parnas, David Lorge & Clements, Paul C.: A Rational design Process: How And Why To Fake It. IEEE Transactions On Software Engineering, 12(2), 1986, pp. 251-257.
2. Erdem, Ali & Johnson, W. Lewis & Marsella, Stacy: Task Oriented Software Understanding. In Proceedings of the 13th IEEE International Automated Software Engineering Conference, 1998, pp. 230-239.
3. Friendly, Lisa: The Design of Distributed Hyperlinked Programming Documentation, Proceedings of the International Workshop on Hypermedia Design (IWHD'95), Montpellier, France, Sylvain Fraïssé and Franca Garzotto and Tomás Isakowitz and Jocelyne Nanard and Marc Nanard (editors), 1995.
4. Gamma, Erich & Helm, Richard & Johnson, Ralph & Vlissides, John: Design Patterns: Elements of Reusable Object-Oriented software. Addison Wesley, October 1994.
5. Knuth, Donald E.: Literate Programming, The Computer Journal, vol. 27(2), May 1984, pp. 97-111.
6. Vestdam, Thomas: Documentation threads - presentation of fragmented documentation, Nordic Journal of Computing, Vol. 7(2), 2000, pp. 106-126.
7. Andersen, Max Rydahl & Christensen, Claus Nyhus: Evaluating Elucidative Programming in an Industrial Setting, Aalborg University, 2001 (submitted to ICSM 2001).
8. WikiWikiWeb, http://c2.com/cgi-bin/wiki.
9. Rajlich, V. & Varadarajan, S.: Using Web for Software Annotations, International Journal of Software Engineering and Knowledge Engineering, Vol. 9, 55-72, 1999.
10. Nørmark, Kurt: Requirements for an Elucidative Programming Environment, Eight International Workshop on Program Comprehension, June 2000.
11. Nørmark, Kurt: Elucidative Programming, Nordic Journal of Computing, vol. 7(2), 2000, pp. 87-105.
12. Nørmark, Kurt & Andersen, Max Rydahl & Christensen, Claus Nyhus & Sørensen, Kristian Lykkegaard: Elucidative Programming in Java, The eighteenth annual international conference on Computer documentation - IPCC/SIGDOC, September 2000, available from http://dopu.cs.auc.dk.
13. Software Quality through Documentation of Program Understanding Research Programme, Department of Computer Science, Aalborg University, http://dopu.cs.auc.dk/.

14. Vestdam, Thomas: Introducing Elucidative Programming in Student Projects, Aalborg University, 2001, available at http://dopu.cs.auc.dk/.
15. Sametinger, Johannes: Object-Oriented Documentation. Journal of Computer Documentation vol. 18, no. 1, pp. 3-14 January 1994.
16. Naur, Peter: Programming as Theory Building. Microprocessing and Microprogramming, vol. 15, 1985, pp. 253-261.
17. Visaggio, G.: Relationships between Documentation and Maintenance Activities. Proceedings of the IEEE International Workshop on Program Comprehension, Dearborn, Michigan, May 1997, pp. 4-16.
18. Beck, Kent: Extreme Programming Explained: Embrace Change, Addison Wesley Publishing Company, 1999.
19. Kramer, Douglas: API documentation from source code comments: a case study of javadoc, In Proceedings on the seventeenth annual international conference on Computer documentation, 1999, pp. 147-153.
20. Kotula, Jeffrey: Using patterns to create component documentation, IEEE Software, Vol. 15(2), March-April 1998, pp. 84 – 92.
21. Rüping, Andreas: Project Documentation Management, In Proceedings of the Fourth European Conference on Pattern Languages of Programming and Computing, 1999.
22. Rüping, Andreas: Reader-Friendly Media for the Documentation of Software Projects, In Proceedings of the Seventh Pattern Languages of Programs Conference, 2000.
23. Rüping, Andreas: The Structure and Layout of Technical Documents, In Proceedings of the Third European Conference on Pattern Languages of Programming and Computing, 1998.