

A Case Study on the Long-Term Effects of Software Redocumentation

Alexander J. Rostkowycz, Václav Rajlich, Andrian Marcus

Department of Computer Science

Wayne State University

Detroit, MI 48202

313 577.5423, 313 577.5408

vtr@cs.wayne.edu, amarcus@cs.wayne.edu

Abstract

This paper presents data from a long-term (four year) case study of the effects of incremental software redocumentation of data communications services software that is a part of a check processing system. It presents data on the relative cost and completeness of the incremental redocumentation and its impact on programmer productivity and cost of the software change. The break-even point occurred after 1.5 years of effort, which means that the extra investment in redocumentation started paying off after 1.5 years of redocumentation activity.

1 Introduction

This paper presents a long term case study of the effects of incremental software redocumentation on programmer productivity.

The software redocumentation reconstructs the software documentation, which either no longer exists or which has become obsolete. Incremental redocumentation rebuilds documentation in small incremental steps. Each step is taken at the end of the software change mini process and redocuments the part of source that is related to the change, and therefore was recently comprehended by the programmer.

Many software engineering practices have been proposed, but their real impact on the programmers in the trenches remains a matter of speculation. Only long term case studies shed light on their true successes or failures. This paper presents one such long term case study.

An important issue for software managers is the cost-benefit ratio for various software practices. While few would argue against good program documentation, it is the cost of this documentation that may make them to question its effectiveness. This paper presents data that answer the following questions: How much does incremental redocumentation cost? What are the benefits? When is the break-even point at which the extra cost of redocumentation pays off?

As a side product, the paper also contains additional data, like the percentage of the programmer's time that was spent in comprehension.

Section 2 of the paper presents an overview of Partitioned Annotations of Source (PAS), which was used in this case study. Section 3 presents the case study. Section 4 summarizes and discusses the case study data. Section 5 contains overview of the related work, and section 6 contains conclusions and future work.

2 Partitioned Annotations of Source (PAS)

This case study uses PAS (Partitioned Annotations of Source) [11], a hypertext-based software source code documentation system. The same hypertext technology used in constructing the World Wide Web – HTML, browsers, etc. - is used by PAS in documenting software. Though it could be generated during original software development, PAS lends itself particularly well to incremental redocumentation during the software maintenance phase.

Regardless of how the programmer arrives at his/her understanding of the software, whether it be partial or complete, tentative or confirmed, it needs to be recorded. And the partitions of PAS can be defined flexibly enough to accommodate whatever understanding is obtained.

PAS is structured so as to facilitate easy access to the information needed by the software maintainer without having to sift through irrelevant information. Using cognitive theories of software comprehension, and leveraging the advantages of hypertext, PAS documentation consists of a set of annotations assigned to each component (e.g. procedure, function, etc.) of a software project. Each annotation is partitioned into specific views or abstractions. The original software and partitions can then be browsed by standard hypertext browsers – Netscape, MS Internet Explorer, Mosaic, etc. – used in browsing the World Wide Web [11].

The resulting PAS structure is a matrix. One coordinate is the components of the code (e.g. functions) and the other is the partitions. Each entry is

an annotation for a particular component from a specific point of view. Every software component is annotated by the same set of partitions, and in turn, every partition applies to all components [11]. This structuring of hypertext into partitions was first presented in [10]. Among the partitions commonly suggested in the literature [10] are: domain, representation, and algorithm.

The domain partition provides a description of the component using only the language of the application domain. The assumption here is that concepts of the application domain are implemented in specific code components [8]. By explaining the mapping of domain concepts to the code, the components become more comprehensible to anyone who understands the domain. Furthermore, domain-based comprehension is important because change requests are often expressed in terms of domain concepts [2]. Programming-specific information should be excluded from this partition.

The representation partition presents descriptions of the individual parameters, constants, variables, and data structures connected with the software component represent [10].

The algorithm partition applies to function components and describes how the component (e.g. function) accomplishes its goal [10]. This is typically expressed in the form of pseudo code.

3 The Case Study

In the case study, the redocumentation was accomplished by the software developer/maintainer, without the help of specialized documentation writers. The developer followed a modified mini process of Software Change [9] that included a redocumentation phase, and collected data about this experience along the way. This case study covers a four year period, 1997-2000, of maintenance of a software subsystem, *Communication Services (Commsrvc)*, which provides data communications services in a distributed check processing system. *Commsrvc* is written in Pascal. When checks are cashed, they must undergo a clearing process which takes them on a journey from the bank at which the check is cashed back to the bank that holds the account from which the check is drawn. The software uses check imaging hardware to accomplish this task. It interfaces with its users via messages using inter-program communication mechanisms.

The software is organized into hierarchical layers each of which relies on services provided by the layer below it, and cooperates with its corresponding peer across a distributed system of mainframes and microprocessors. The topmost layer concerns itself with the application domain, while the supporting underlying layers fall within the systems software

domain. The implementation supports several operating systems, including Unix, Windows NT, OS/2, and iRMX (Intel). *Commsrvc* is used today in several banking applications.

3.1 The Evolution of *Commsrvc*

Commsrvc was initially developed in the early '90s using the waterfall software development life-cycle. The functional specifications provided an operating system independent description of the *Commsrvc*. The design was done with CASE tools that utilize structure charts. The initial implementation of *Commsrvc* was done by a single developer and co-author of this paper, Alexander Rostkowycz. Unit testing was done by several additional developers, and integration testing was done on-site by an engineering quality assurance group. A considerable body of documentation was originally generated for *Commsrvc* in the form of functional specifications and structure charts and provided an accurate documentation at the time of first product release. Since the time of the first release, none of the documentation was revised.

The original design did not change significantly over a decade of maintenance. Table 1 shows data that reflects the evolution of *Commsrvc* from its first release and increase in size over time.

Table 1: The evolution of *Commsrvc*: number of Pascal functions and procedures, lines of code, lines of comment, and blank lines.

Year	Func.	LOC	LOCom	Blank	Total
Release 1	147	5739	998	1500	8237
1997	248	10177	1199	3530	14906
2000	289	11932	1470	4346	17748

Commsrvc was developed in an environment where software engineers *own* specific pieces of the software. The programmer who originally wrote the code was the primary *owner*, and typically assumed maintenance responsibilities for the code. *Commsrvc* has one primary developer. Though the code was briefly maintained during the initial crunch by two other developers for about one year, it has since remained the responsibility of the original owner.

The maintenance cycle was initially done with a mainframe screen based editor, relying mostly on hard-copy compiler listings. As PC-based tools improved, a GUI-based IDE was used for source control and backup. In particular, the language-blind editing and search facilities of Microsoft Visual Studio were used to maintain *Commsrvc* Pascal source. The modified source was uploaded to the mainframe, where final build and test occurred. This was the source maintenance model that was in effect during the entire case study period (1997-2000).

Maintenance of *Commsrv* occurred in an environment of declining software engineering resources. The check processing software, of which *Commsrv* is a part, was originally developed by dozens of software engineers, but it was supported by only a handful of engineers at the time of the case study.

3.2 The Change Process

During the case study, 52 changes were performed. Each change followed a mini process that was based on the mini process in [9]. The mini process used in the case study consists of the following activities:

- Process Initiation
- Comprehension and Impact Analysis
- Design and Scheduling
- Implementation and Testing
- Redocumentation
- Configuration Management and Release

Process Initiation. The process is initiated for a variety of reasons, such as *problem report*, *preventive maintenance*, or *new feature request*.

Problem Report. The problem report indicates that the product behaves in an unstable or otherwise undesirable manner. It is usually received from a customer and sometimes originates internally from lab testing. Customer problem reports were always submitted in writing.

Original problem reports are typically rather vague and require more information in the form of logs, both system- or application-generated. If most of the software is stable, observers of the problem are asked questions intended to place the problem in a narrower context (e.g., “what was going on at the time?”, “is the problem reproducible?”, etc.). In the absence of trained observers, we had to rely more on automated logging to record occurrences of exceptional events (i.e., exception paths rarely taken through the code). If a problem was reproducible, we sometimes requested traces of traffic between specific units, which served to provide a context for the problem. The source of many problems was mis-configuration – incompatible release levels, or mistakes made in the setting of application parameters, in the preferences form, or in the configuration files. Problems resolved without code changes were not included in this case study.

Preventive Maintenance. Preventive maintenance was intended to enhance reliability and comprehensibility, to ease extendibility, etc. Preventive maintenance work was usually initiated by the developer. *Commsrv* runs 24 hours a day, and developers do not like getting emergency calls in the middle of the night. Clean, easily comprehensible software is needed for quick fixes. Managers rarely

initiated preventive maintenance requests. In fact, this work was commonly done ‘on the side’ or during ‘free time’, without management’s knowledge to avoid veto by management on grounds that “*if it’s not broken, don’t fix it*”. In reality, much of this work was carried out along with other, officially sanctioned, changes.

New Feature Request. New feature requests are requests for additional functionality of the product. They usually originate from the customers or marketing. They could also originate from management wish lists, anticipating customer needs.

The change process was initiated from a variety of sources: a *customer*, a *colleague* (a fellow software engineer, in-house test personnel, or field support personnel), the *software engineer* himself, or the *management*.

Most change processes in this case study initiated with a formal change request. The exceptions were some changes initiated by the software engineer himself or a colleague.

Formal change requests are assigned a priority and placed on the queue of the software engineer. The individual software engineers continuously monitor their own queue and commence the comprehension phase as their time becomes available. Hot problems cause instant reprioritization of the developer’s time.

Comprehension and Impact Analysis. During this phase we decide what needs to be done to the software to resolve the problem. Though this phase may not consume as much time as the others, it is usually the phase of highest anxiety – especially if there is a high priority change request (i.e., due to a program crash) whose cause is unknown. It is characterized by the following activities:

Concept Location. Here, we map the change request to software components.

More often than not, it is not obvious from the problem/feature description which software components are to be the subject of study. Technical leaders usually assist in this function when they assign the problem/feature to a particular software engineer. These leaders have enough knowledge of software structure to isolate the problem down to a specific program, and assign it to its primary owner. The problem may be reassigned if the initially identified software is found innocent. If the change request is actually a bug report, the maintenance software engineer carefully inspects the request for clues. Sometimes the author of the request is interviewed by the engineer to gather as much information as possible.

Study Code and Documentation. Source code and related documentation are typically studied as a part of the concept location. They are also studied in order to gain comprehension and a high level of confidence. In the case of *Commsrv*, this assumes *data*

communication domain expertise, and sometimes *application domain* expertise (consult customers or application developers).

It also involves *language/tools expertise* (consult reference material, a fellow engineer, text book, online reference, source code references, etc.). *Program expertise* (via browsing/comprehension) is also involved.

Identify/Weigh Alternative Solutions. When time allows, we try to maintain or improve the integrity and elegance of design. In the case of urgent fixes (an irate customer), we may be forced to identify quick and dirty fixes to be implemented in the short term, with the intent of doing it right when more time is available.

Design and Schedule. During this phase we refine the design for the modification. This might involve developing a formal schedule or writing a *design change paper*, especially if initiated by a new feature request.

A decision is made on an implementation strategy to carry out the design. This involves dividing large changes into several successive incremental *transformations*, each with its own implementation and testing activity.

Implementation and Testing. The implementation and testing activities carry out and validate the software changes decided on in the previous activity. In practice, it consisted of the following activities repeated for each successive transformation of the software:

Implementation. Refine the design decisions made during the design and schedule phase. Again, this typically involves more study of code and available documentation. Restructure the software, if necessary, to localize change, to adhere to architectural philosophy, and to maintain architectural clarity. This involves even more study of code and documentation. Do the incremental coding for the change.

Test. Unit testing is done by the developer. An automated Design Verification Test is used to provide regression test of basic functionality. Integration test is also done by running with a wider base of software (real applications vs. developers own unit test utilities).

Fix. Fix any problems encountered during testing. This may involve fixing problems introduced in a previous transformation of the software.

Refactoring. If time allows, take advantage of recent re-familiarization with code to clean it up a bit (improve its clarity, readability – e.g., rename variables and functions, reformat the code, especially in those areas that presented comprehension challenges); also could do some restructuring of the software in anticipation of upcoming changes.

Redocumentation. During this change process phase, we preserve recent comprehension by recording

it in PAS. A more detailed description of this process activity is presented in section 3.3.

Configuration Management and Release. Check-in of final version of working software occurs during this phase of the change process. If there is a need to formally issue a new/patched version of the software at the conclusion of the change process, the following activities are performed:

- Check-out latest version from Source Control
- Do Release build
- Regression test
- Write Release Notes
- Make release available to customer

Each of the 52 changes of the case study repeated this process. Table 2 shows the chronology of the changes over the four years of the case study. Entries opposite each month indicate which change (and transformation) was worked on during the month. A change number accompanied by a letter of the alphabet (e.g., 7a, 7b, 7c, 7d, 7e, 7f, 7g, 7h, 7i, and 7j for change #7) indicates that the change was subdivided into several transformations. Note also that the effort on some changes spanned several months.

3.3 The PAS Technique Used

The source code has two main structural parts: functions (and procedures) and non-functional components. Applying PAS philosophy, each function or procedure in the source has the following partitions: *Domain*, *Intent*, *Representation*, *Algorithm*, and *Annotation Notes*.

Non-functional components are sets of global declarations – CONSTs, TYPEs, and VARs. Each of these groups counts as a separate component. There are 3 such components in *Commsrv*. They have just one partition: representation.

The **Domain** partition describes the role of the component in terms of the Systems Software/Data Communication domain, understandable to anyone familiar with these domains.

The **Intent** partition is a brief narrative description of the purpose of the component.

The **Representation** partition, for functional components, describes the role of function parameters and local variables. For non-functional components, the intended role of the specific global CONST, TYPE, or VAR is described, along with a very brief mention of where it fits into the Systems Software/Data Communication domain.

The **Algorithm** partition contains a pseudo-code description of the component.

The **Annotation Notes** partition was partially used as a change history and also recorded comprehension

time and annotation writing time during the annotation visit.

Table 2: Change process timeline

1997									
Jan	1a								
Feb	1a	1b	1c						
Mar	1c	1d							
Apr	1d								
May	1d								
Jun	2								
Jul	2	3	4	5	6				
Aug	6								
Sep	7a								
Oct	7a	7b							
Nov	7b	7c	7d	7e	7f	7g	7h		
Dec	7h								
1998									
Jan	7h	7i	7j	8a	8b	9a			
Feb	9a	9b	10						
Mar	10								
Apr	11	12a							
May	12a	12b	13a	13b	13c	14	15		
Jun	15	16a	16b	17	18	19	20a	20b	
Jul	20b								
Aug	21a	21b							
Sep									
Oct	22	23							
Nov	23	24							
Dec	24								
1999									
Jan	25	26	27						
Feb	27	28	29a	31a					
Mar	29a	29b	29c	29d	30	31a			
Apr	31a	31b							
May	32	33							
Jun									
Jul	34	35							
Aug	35	36	37						
Sep	37	38							
Oct	39a	39b	39c	40	41	42			
Nov	42								
Dec									
2000									
Jan									
Feb	43								
Mar	43	44	45	46					
Apr	46								
May	46								
Jun	46	47	48a	48b	49a	49b			
Jul	49b	50a	50b						
Aug									
Sep									
Oct	51								
Nov									
Dec	52								

PAS documentation consists of the following set of HTML pages: architecture page, index page, annotations page, source page, symbol references page.

Navigation starts at the architecture page. *Commsrvc* functions are grouped into logical modules and linked by call dependencies into a call-graph. The index page is divided into sections, each corresponding to a logical module. Clicking on a module name brings the user to the module's entry in the index page.

Within each logical module section there is a list of *Commsrvc* components assigned to that module. Each component has links to its related documentation in the annotations page, the source page, and the symbol references page.

The annotations page contains the complete body of partitioned component annotations for *Commsrvc*.

The source page evolved over time. Its first incarnation was manually generated and could be used only as an HTML tree leaf. In order to ease navigation, it now contains links to the index page, accessible via a *Notes link* placed within a comment next to the definition site of each component. This allows the developer to reach the annotation directly from the source file.

The symbol references page did not make its appearance until a specialized program was written to automatically generate it, since generating it manually, though possible, would be prohibitively expensive. It contains links to the source page.

The inherent flexibility of PAS allowed this evolution of the architecture and index pages, while allowing the annotations pages to remain stable.

Since the source file itself is one of the PAS partitions, it had to be slightly modified so as to be viewable by a browser. Placing the HTML elements within comments allowed us to convert the source to an HTML document allowing it to be both browsed and compiled.

The redocumentation event is termed here as an *Annotation Session (AS)*. Decision to do an AS was based on a combination of the following criteria:

- Enough components were changed or added since the previous Annotation Session.
- Re-comprehension of enough components had occurred since the previous Annotation Session.
- The change process was complete, or an intermediate logical break was reached.

In any case, AS took place only after preliminary testing showed the accumulated modifications to be viable (error-free). This prevented/reduced the need to redo annotations due to logic changes arising out of bug fixes. Typically, a few days to several weeks passed from the time of a given component change until its redocumentation.

The AS consisted of the following activities:

1. Identify and list all components which had *changed* since the previous Annotation Session. To do this, a file comparison utility was used.
2. Recall and list all components (not previously annotated) which were studied in detail (re-comprehended) as a by-product of the current Mini Process, but which were not changed since the previous AS. The components in this list are classified as *study* visits in the collected data.
3. List all components that were added since the previous AS. This includes only functions. The components in this list are classified as *new function* visits in the collected data.
4. Record the time spent performing steps (1), (2), and (3) as Annotation Overhead (OH).
5. From the component list compiled in (1), (2), (3), or (9), select a component for annotation, locate its corresponding component annotation template, and open in a text editor ready for annotation. Add the time spent completing this step to the OH. **Note:** Approximate time needed to generate an annotation template was measured to be approximately 3 minutes.
6. Take any time necessary to refresh re-comprehension of the component selected in (5). Record this time as the component *comprehension* time. Any time taken to comprehend the component prior to the AS is not chargeable to the annotation, since it would have taken place anyway, regardless of redocumentation.
7. Write the annotation partitions for the component selected in (5) and re-comprehended in (6), in the following order: domain, intent, representation, and algorithm. Record the time spent doing this as the component annotation *write time*.
8. Repeat activities (5) thru (7) until the list compiled in (1), (2), (3), or (9) is exhausted.
9. While in the redocumentation mood, identify any additional components for redocumentation. These components, by definition, have not changed, nor were they purposely re-comprehended or added as part of the current change process. Rather, they are opportunistically selected components which the engineer feels comfortable redocumenting. The components in this list are classified as *annotation-only* visits in the collected data. Go to step (8).

3.4 Collected Data

The data was originally recorded in either a special document called Process Log of Figure 1, or in the Annotation Notes partition of the component being documented. There was one such entry for every change. Some quantitative data was recorded at the

time of each software transformation or AS, while other data was recorded post-mortem after the conclusion of the change process.

Though data was collected spanning the entire change process, detailed, high-resolution timing data was taken only during the redocumentation phase of the process (during the AS). The occurrence of the AS was recorded in the Process Log, and for each software component annotation visit, two times were recorded in the *Annotation Notes* partition: *comprehension time*, and *write time*. In addition to these annotation related times, data was also recorded for development time excluding redocumentation time. These are approximations of design (pre-code) and implement (post design) effort. No effort was made to isolate sub categories of activity, such as code comprehension time vs. problem re-creation time, vs. conceptual design, vs. detailed design, etc.

<p>Formal design doc or change paper written? Formal schedule drawn up? Type of change: - Adaptive, Corrective, Perfective, Preventive Change Summary: Include a copy of Change History comments contained in the <i>Commsrvc</i> source file For each Change Process transformation: Nominal date assigned to the transformation List of functions visited, identifying: Modified/Annotated during this visit Name of function visited Studied/Annotated during this visit Annotated for the first time during this visit Brief description of any change Functions deleted (Non-function component visit data is in the component's <i>Annotation Notes</i> partition) Date Span Calendar Workdays spent in total effort Calendar Workdays spent in Pre-code Study/Design effort Total Man-Hours per week of effort Total Man-Hours spent in Pre-code Study/Design effort Total Man-Hours spent in Non-pre-code effort Total Man-Hours (combined pre-code, non-pre-code) Annotation Session activity: Note if a session occurred at conclusion of transformation Note annotation session # Note annotation overhead time (determine what needs to be annotated) (Per component annotation data is kept in the component's <i>Annotation Notes</i> partition) Release activity Note if any Base/Patch release activity occurred at conclusion of transformation Note relative release # Testing Notes Annotation/Redocumentation Notes Miscellaneous Notes</p>

Figure 1: The Process Log outline

In addition to the self-explanatory items appearing in the Process Log, there were also categories such as *investigative activity notes* and *analysis summary* which

provided a place to record the events and difficulties specific to the change process.

The 52 changes generated 81 final transformations. The strict formatting and redundant labeling facilitated *grep*-like searches across all Process Log entries - used to study the information and locate desired data.

Besides the Process Log and Annotation Notes, other documents included trouble report responses, release notes, and change notes written at the time of the change.

4 Results and Discussion

Based on all this data, several aggregate measures were taken.

By tracking the number of components which were visited at least once, we can determine documentation coverage of the software. Table 3 provides yearly summaries of the redocumentation coverage on a cumulative and yearly basis. By the end of the 4th year, 241 of 289 (83%) components were redocumented.

Table 3: Column 1 and 2 – cumulative (C) and yearly (NC) % of components annotated.

Column 3 – yearly annotation time (AT) per component annotation visit (CAV).

Column 4 – change effort (design + implementation + testing) per component change (CC).

Year	C %CA	NC %CA	NC AT / CAV	NC Change Effort / CC
1997	25.95%	25.95%	18.1 min	315 min
1998	68.51%	42.56%	8.6 min	142.6 min
1999	76.12%	7.61%	4.4 min	121.4 min
2000	83.39%	7.27%	5.3 min	103.2 min

The total change effort is made up of non-annotation time (i.e., design, implementation, test phases) and annotation time. Table 3 provides a summary of the Design/Implement/Annotate time by change, on both a cumulative and yearly basis. Time spent in the incremental redocumentation effort is very small compared to the total development/maintenance effort, consuming a cumulative 5.48% of total effort across the 4 year case study period. On an annual basis, percentage annotation effort falls from 6.21% in 1997 to 4.94% in 2000.

Table 4: Breakdown of annotation effort – cumulative (C) and yearly (NC) comprehend (C), write (W), and overhead (OH)

Year	C.C	C.W	C.OH	NC.C	NC.W	NC.OH
1997	416	1903	184	416	1903	184
1998	1070	3604	529	654	1700	345
1999	1266	3963	783	196	328	254
2000	1422	4367	975	156	434	192

Looking at the annotation effort alone, we are concerned with the annotation *Comprehension Time* vs. *Write Time* vs. the *Overhead*. Table 4 provides a cumulative and yearly summary of this relationship. We can see that *Write Time* comprises the majority of this time when compared to incremental *Comprehension Time*. The probable reason for this is that at the time of annotation, we still possess the component comprehension gained or enhanced as a by-product of the preceding change. Very little, if any, incremental effort is needed to (re)comprehend the components undergoing redocumentation at the time of the *Annotation Session*. At the end of the case study, cumulative breakdown of the annotation effort stood as follows: comprehension time (21.03%), write time (64.55%), and overhead time (14.42%).

We also analyzed the total non-annotation component of the change effort. The time spent in the implementation/test phase is approximately twice that spent in the design phase. At the end of the case study, cumulative breakdown of the non-annotation effort stood as follows: Study / Impact Analysis / Design time (33.27%) and Implement/Test time (66.73%).

The first time a component is redocumented, we refer to it as a redocumentation *1st visit*. Succeeding visits are termed *revisits*. It comes as no surprise that the early phases of project redocumentation are dominated by 1st visit effort. As 1st visit effort wanes, revisit effort picks up, and levels out at a low maintenance-mode level. We also analyzed this data per year and per change. The average component redocumentation time drops from 18.1 minutes in 1997 to 5.3 minutes in 2000 (a 71% reduction in redocumentation effort over the 4 year period).

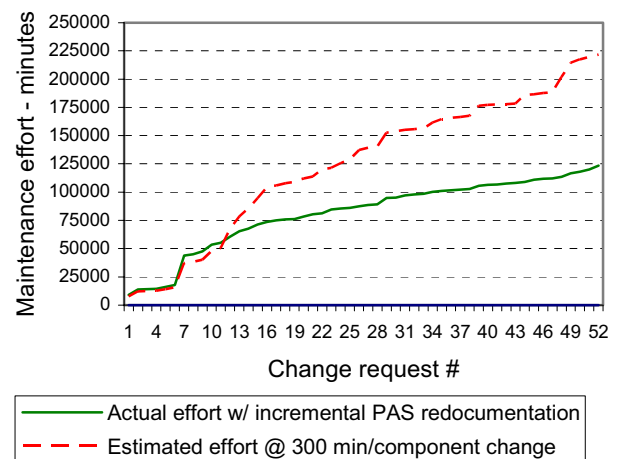


Figure 2: Estimated maintenance effort vs. actual maintenance effort using PAS

Perhaps the most significant numbers to be extracted from the collected data relate to the clear and rather dramatic decline in maintenance effort, as illustrated in Figure 2 and Table 3 (columns 3 and 4). To facilitate comparisons, in Table 3 we express maintenance effort in terms of minutes per *Commsrv* component changed. This table isolates minutes of maintenance effort per component change for the individual years of the case study. So as not to bias the numbers with the known decline in redocumentation effort, only the Study / Impact Analysis / Design and Implementation/Test phases of the maintenance effort are included. We note 315.0 minutes of effort per component change in 1997, and see it steadily decline to 103.2 minutes of effort per component change during 2000. The steepest decline came between the first and second year, where we see a 55% drop in effort (315 down to 142.6 minutes per component change). A further 15% drop is experienced between each of the succeeding years (142.6 down to 121.4 minutes from 1998 to 1999, and 121.4 down to 103.2 minutes from 1999 to 2000). The case study thus demonstrates a rather early pay-off for the PAS redocumentation effort. This finding came as a real surprise. Given the developer's intimate familiarity with *Commsrv*, not much of an impact was expected in this area.

We also assessed the cost of the annotations versus the benefit in saved maintenance effort. We wanted to establish whether the annotations paid off during the four years of the case study, and if so, to what degree, and when did the break-even point occur.

In summarizing the entire maintenance over redocumentation effort for the four year case study, the redocumentation effort represents only 5.48% of the total maintenance effort:

In estimating the amount of effort saved, we must first make some assumptions as to what the maintenance cost might have been without the benefits of redocumentation. For this purpose, let us use effort numbers from a period of time before the redocumentation benefits had a chance to really take effect. This would correspond to the first year of the case study (1997). In the first year (see Table 3), the effort cost on a per-component-changed basis, excluding redocumentation, was 315 minutes. This is based on cumulative non-annotation minutes and cumulative components changed by the end of 1997. This is our baseline for comparison. Figure 2 plots the total actual effort, inclusive of the redocumentation work (solid line) against this baseline (dashed line). The baseline for the first year (prior to and including change #6) plots actual non-annotation effort. The break-even point occurs where the lines intersect and diverge from each other. From Figure 2 we see that this happens at the time of change #12 (May 1998),

approximately 1.5 years into the case study. The savings in maintenance effort increases from this point forward, as the two curves diverge.

An important fact needs to be mentioned at this point. The primary motivation in selecting *Commsrv* for annotation was that it was lacking in component-level documentation. Since the original developer was also the sole maintainer, he already enjoyed a good level of comprehension of the program and its components when the case study started. Generating the annotations was seen primarily as a benefit to future maintainers of the program. This need has not yet materialized as of the time of this writing. Though the reliance on the annotations was not as heavy as might be the case with a new maintainer, they were still used.

Though the annotations were frequently consulted during the change process, the only annotation consultation data recorded involved those cases where a component's annotation was seriously studied in detail in an effort to assist in (re)comprehension. Out of 52 changes, fully 14 (27%) exhibit some serious study of previous annotations with no corresponding change of the component. A total of 81 such annotation *study-only* instances were recorded. Changes showing the highest previous annotation *study-only* rates are: change #29 (*excessive CPU utilization*), with 27 studied annotations, change #21 (*long delays experienced while streaming data*), with 12, and change #11 (*Commsrv crash at a customer site*), with 8. Given the critical nature of the problems associated with these changes, the availability of annotations was most appreciated.

Finally, we also ranked all the components of *Commsrv* based on the change frequency. The following components suffered the most changes during the four years: *Cm_Main_Init()* (35 changes), *Global_VARS()* (27 changes), *Global_CONSTs()* (23 changes). This type of data indicates that these components should be annotated first.

4.1 Eliminating Confounding Factors

The preceding discussion reveals a rather dramatic and steady decline in *per-unit* maintenance effort across the case study period. This decline correlates to an increase in annotations, but is the PAS redocumentation work the primary cause of the reduction?

One could argue that the reduction in effort could be due merely to an increase in experience level of the maintainer. However this is not so, since at the time the case study started, the maintainer was already quite intimately familiar with *Commsrv*. He was its primary designer and implementer, and was its sole maintainer for the eight year period preceding the case study.

Was there a change in tools or some technique (other than redocumentation) which may have

accounted for the change in maintenance effort? Except for the introduction of the PAS redocumentation element, the tools and techniques used remained remarkably stable across the four years. Microsoft Windows NT 4.0 was the operating system throughout the case study. Although newer versions of Web browsers and Microsoft Visual Studio (the primary tools used for *Commsrv* maintenance) were used as they became available, the features used remained constant.

There was also not much variability in the testing/debug techniques used. Regression testing was especially stable with respect to effort, consisting of a fixed series of automated tests.

The arrival rate and severity of the change requests remained as unpredictable and random as the real world events that caused them. The decline in per-unit maintenance effort cannot be attributed to increasingly trivial changes due to software maturity. To the contrary, the challenge of the changes tended to increase with the passage of time, as the circumstances which precipitated the changes became more difficult to recreate and diagnose.

5 Related Work

A summary of the field of software comprehension is found in [12]. A number of approaches to software comprehension have been identified. They deal with the strategies programmers use to understand code. Documentation facilitates and records this comprehension.

There are hundreds of documentation solutions in industry and research. One of the most commonly used java documentation tools is Javadoc [6]. Javadoc parses java source code and generates documentation from the comments found in the code. It generates one .html file for each .java file and each package. In addition, it produces a class hierarchy (tree.html). Javadoc parses special tags, similar to PAS partitions, that are recognized when they are embedded within a Javadoc comment.

Alexandria [1] is a CVS / Javadoc / Source code / Documentation management system meant for use within Open Source projects. The backend is implemented in XML. Definition of source code is done within XML and then transformed (XSLT) into various HTML files and a master build.xml file which is used to drive Ant. Ant downloads a CVS repository and then builds documentation with the use of Javadoc.

Another very successful documentation tool is Doxygen [5]. Doxygen can be used for documentation of Java, C, and C++ programs and is somewhat similar to Javadoc and PAS.

A well known hypertext documentation technique is Donald Knuth's Literate Programming [7]. The goal of Literate Programming was to have a program read like a "work of literature". Literate programs combine source code and documentation in the same file that is meant to be read by the programmer. The documentation is more than traditional comments; it can combine figures, formulas, footnotes, or whatever the word processor has to offer. Literate programming tools parse the file to produce either documentation or source code. There are many tools that have been developed for literate programming: WEB, CWEB, FWEB, NOWEB, and many more (see www.literateprogramming.com).

DocGen has been developed by The Software Improvement Group [3]. It automatically analyses software systems, presenting the results in a standard web browser. It generates information regarding the whole system, such as call dependency graphs, cross-references, and metrics.

There exist a number of documentation tools that are doclets and are used to document java code. One of those tools is DocWiz [4]. DocWiz provides a graphic user interface to enter Javadoc comments.

Work on document generators is also reviewed in [13].

6 Conclusions

In summarizing our case study, data collected on our use of PAS in incremental redocumentation indicates that:

- Software redocumentation was achieved at small cost.
- The redocumentation effort, as a percentage of the total maintenance effort, declined over time.
- As the redocumentation effort progressed, there was a steady decline in total maintenance effort per unit of software change, resulting in significant savings of total maintenance effort.
- Break-even point at which redocumentation investment started to pay off was approximately 1.5 years.

6.1 Disadvantages of the Technique

Based on our experience, let us note some disadvantages of the technique. First off, at the end of the case study gaps in documentation still persisted. Since the technique is driven by software change, components which don't change or haven't been studied will not be documented.

Though the redocumentation proved to be a relatively small effort, for those software engineers who do not like to write, redocumentation still is a burden.

6.2 Advantages of the Technique

We have found that incremental redocumentation, when done by a disciplined developer/maintainer, can be very cost-effective. In our case study, redocumentation consumed only 5.48% of the total maintenance effort and resulted in the redocumentation of 83% of the software components across the 4 years.

The technique reduces the mental drain of writing documentation. Its incremental nature inherently minimizes the number of components which must be redocumented at a sitting. The developer having a limited tolerance for doing documentation tends to be met with the need to do only a limited amount of redocumentation (the authors' own documentation tolerance is about 20 minutes before needing a break).

When consistently applying the technique, documentation is guaranteed to be up-to-date. And since PAS documentation is tied to a component, it is much easier to identify and locate affected documentation when a component changes. This makes it easier to verify that existing documentation corresponds to the current state of the software. Additionally, troublesome components are guaranteed to receive a lot of needed documentation, since they will be frequently revisited.

Perhaps the most significant of all results is the steady decline in maintenance effort per unit of software change, resulting in very significant savings in maintenance effort. The experience documented with this case study has demonstrated that software can be effectively and efficiently documented using the incremental PAS (or similar) technique. This effort can result in a significant reduction and savings in the maintenance effort.

6.3 Future Work

In the early phases of incremental annotation, very little documentation exists (because very little of it has been written). Trying to find the annotation for an arbitrary component often ends in failure. To address the resulting frustration, some kind of visual clue might be given early in the browsing path that no underlying documentation exists, perhaps by appropriate coloring or even elimination of the corresponding links.

It is important to note that the incremental redocumentation of this case study began significantly later than the original release of the product, and yet resulted in a significant percentage of the software being redocumented. This would support the argument

for not doing code component-level documentation at the time of initial development, but rather wait until commencement of the maintenance phase. Since some code components will certainly experience infant mortality, why waste time documenting them? Those components that survive will most likely experience some changes. Documenting these components when they change could thus become an effective way of documenting the entire software. The appropriate time to start redocumentation remains an open issue to be answered by future experiments.

7 References

- [1] Alexandria, "The Jakarta Alexandria Project", <http://jakarta.apache.org/alexandria>, 2002.
- [2] Chen, K. and Rajlich, V., "Case Study of Feature Location Using Dependency Graph", in Proceedings of International Workshop on Program Comprehension, 2000, pp. 241-249.
- [3] DocGen, "Software Improvement Group", Date Accessed: March, <http://www.software-improvers.com/docgen.html>, 2004.
- [4] DocWiz, "The DocWiz Project Web Site", Date Accessed: March, <http://docwiz.sourceforge.net/>, 2003.
- [5] Doxygen, "Doxygen web site", Date Accessed: 1/7/2003, <http://www.stack.nl/~dimitri/doxygen/>, 2004.
- [6] Javadoc, "Javadoc web site", Date Accessed: 1/7/2004, <http://java.sun.com/j2se/javadoc/>, 2004.
- [7] Knuth, D., "Literate Programming", *The Computer Journal*, vol. 27, no. 2, 1984, pp. 97-111.
- [8] Rajlich, V., "Incremental Redocumentation with Hypertext", in Proceedings of 1st Euromicro Working Conference on Software Maintenance and Reengineering (CSMR '97), Berlin, Germany, March 17-19 1997, pp. 68-73.
- [9] Rajlich, V., "A Methodology for Incremental Change", in *Extreme Programming Perspectives*, Marchesi, M., Succi, G., Wells, D., and Williams, L., Eds., Reading, MA Addison Wesley, 2002, pp. 201-214.
- [10] Rajlich, V., Gudla, R., and Doran, J., "Layered Explanations of Software: A Methodology for Program Comprehension", in Proceedings of IEEE International Workshop on Program Comprehension, 1994, pp. 46-52.
- [11] Rajlich, V. and Srikant, V., "Using Web for Software Annotations", *International Journal of Software Engineering and Knowledge Engineering*, vol. 9, no. 1, 1999, pp. 55-72.
- [12] Storey, M.-A. D., Fracchia, F. D., and Muller, H. A., "Cognitive Design Elements to Support the Construction of a Mental Model during Software Visualization", in Proceedings of 5th International Workshop on Program Comprehension, 1997.
- [13] van Deursen, A. and Kuipers, T., "Building Documentation Generators", in Proceedings of International Conference on Software Maintenance, 1999, pp. 40-49.