Software Quality Journal **6**, (1997) 337–348

A case study to demonstrate the impact of quality design principles when restructuring existing software

HOSSEIN SAIEDIAN[1], JAMES J URBAN[2]

[1]*University of Nebraska, Department of Computer Science, Omaha, Nebraska 68182, USA*
[2]*US Telecommunications, Omaha, Nebraska 68102, USA*

Received July 1995

The case study is about the System Monitor and Control Facility (SMCF) workstation product developed by a major telecommunications company that has been used to monitor MVS OS mainframe computer systems since 1983. In 1991, mainframe UNIX systems were added to the list of systems supported using software executing on the mainframe side. In 1994, an effort to develop a common interface using TCP/IP and Remote Procedure Calls (RPC) began with a product being developed in the C. The product, which was officially delivered in June of 1994, was coded using structured programming techniques. However, after the product had been in use for some time, maintaining and extending the code for additional functionality and portability was less than desirable. A decision was made by the programmers who support the host-side code to restructure (re-engineer) it such that certain software engineering principles be included into the product to make the product more maintainable and portable. This paper discusses the factors that led to the initial decisions of the designers and programmers, the evaluation of the existing code, and the resulting code with software engineering principles re-engineered into the existing code, and how the incorporation of these principles make maintenance simpler and how they may prevent or minimize defects in the future.

Keywords: design principles, re-engineering, re-structuring, quality software

1. Introduction

Changing active program code ‘to make it better’ has been described by some as reverse engineering, restructuring, or re-engineering. Reverse engineering, according to Chikofsky and Cross [1], is the process of analysing a subject system to identify the system’s components and their interrelationships and to create representations of the system in another form or at a higher level of abstraction. Glass [2] calls it the process of examining a completed software system to abstract out its design and underlying requirements. Choi and Scacchi [3] mention four properties that reverse engineering possesses: *structural, functional, dynamic* and *behavioural*. Restructuring is the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system’s external behaviour of functionality and semantics [1]. It is the act of taking a possibly unstructured program and adding structure to it [2].

Re-engineering is the examination and alteration of a subject to re-constitute it in a new form and the subsequent implementation of the new form [1]. Glass [2] adds to the definition by saying that re-engineering is the revising of a program semantically as well as syntactically to improve its operation. It may involve adding function, but most commonly, it is restricted to improving its maintainability.

These definitions are virtually the same when you consider that the intent of each of these activities is improving existing software quality. The question then becomes, when to start the activity? That depends entirely on the present state of the software and the amount of changes made to the software over its lifetime from its initial implementation. Some software that was well written to begin with will take more time to get into the state of un-maintainability while poorly written software may require immediate attention. The key is recognition that the software is in a state of un-maintainability in its life-cycle.

2. Background of the case study

To describe the decisions when improving a piece of software, it is necessary in this case study to describe what the product is supposed to accomplish so that the reader can more easily follow along when certain references are made. The product's name for the case study is called SMCF which stands for System Monitor and Control Facility. SMCF has a number of pieces. Some of which run on a stand-alone OS/2 workstation while other parts run on midrange and mainframe systems which the workstation monitors. For this case study, the software that is run on the host system will be analysed and in particular the UNIX based product.[1] The reason that this part of the SMCF code was selected was the large amount of maintenance performed thus far and the need to isolate and separate the client/server communication code from the application code for better portability to different hardware. SMCF uses RPC (remote procedure call) software to establish a connection to the workstation monitor. Messages are then passed between the host and workstation with the objective that the workstation will be used to alarm on situations that require operator intervention or for an operator to enter commands as a response to a host situation. RPC allows the programmer to write a generic interface routine and then generate the connection code 'stubs' with the `rpcgen`[2] code compiler. The programmer would have to write the application code associated with the activity on the host or the workstation and interface with the connection code 'stubs'. Even though a simplified view, there are three distinct pieces of code designed and written for the SMCF client/server model: code on the workstation, code on the host system, and communication code that the two other pieces of code must interface with to pass messages.

3. Original development decisions

The RPC version of this program had a predecessor that lacked reliability. Reliability is the sense of robustness, correctness, and consistency. The code was written using a combination of Bourne shell code and C code. It received its systems messages from the RS232C connection port on the workstation. In certain situations, the product either didn't work under abnormal conditions or didn't behave the same in relatively the same conditions. There was also the performance issue of the shell code and the number of processes necessary to be created and utilized in the UNIX architecture when the OS/2 workstation was initializing. The major flaw in the design, however, was that for a OS/2 workstation to monitor more than one UNIX system, a UNIX system (or focal point system) had to be used to filter messages from all other monitored systems. If the focal point system crashed or its communications failed, messages from other systems could be and at times did get

[1]SMCF can also monitor MVS mainframe systems with separately written software specifically for that operating system's architecture.
[2]`rpcgen` is a utility tool that generate client/server interface C code to implement an RPC protocol.

lost until the backup focal point system became operational. The rewrite of the code had to correct this critical exposure and eliminate the additional overhead incurred by the focal point domain in watching other UNIX systems.

Therefore, in early 1994 a design session was held, and the objective of the session was to come up with a common piece of software that could be used by the OS/2 workstation and run on a TCP/IP type network to monitor MVS mainframes, UNIX mainframes and midrange systems. The C language was the logical choice as the programming language for all environments.

DCE (Distributed Computing Environment) was considered as the communication interface but was immediately turned down as an option as most UNIX systems installed at the telecommunications company did not support the environment. RPC was the next piece of software that was considered. Since it appeared as though most UNIX systems were at a level of their operating system capable of supporting the software, the RPC software was to be the common communication protocol as it was also supported on the workstation in the OS/2 software and on the MVS mainframes.

The architectural design process included a discussion of security and the possible use of light-weight processes. It was decided that the software that was about to be written must incorporate security features employed on the most current release of UNIX and MVS. In addition, the software must check to verify that a workstation, group ID, and login ID were authorized to run the software. The use of light-weight processes would be incorporated if the programming environment allowed it. Only the OS/2 workstation and MVS environment could take full advantage of light-weight processes. Since most UNIX systems don't employ such a facility at this time, heavy-weight processes would have to be used (i.e. `fork` system call).

The detailed design took place over a period of approximately three weeks with coding of the design immediately afterwards. The first release of the code was ready for implementation by the end of May 1994.

4. Observations in the delivered design

The delivered product of the group's efforts was viewed as an immediate replacement for the old method of communication and monitoring. However, to determine the total quality of the delivered product from a software engineering point of view, a 'break-in' period was used to acquaint the users to the product. The 'break-in' period was informally set up after the product was system tested. The development staff kept a list of operational items for correction as suggested by the users to build on the existing quality of the product.

Quality factors of delivered products are determined to be documents, reliability, understandability, maintainability, usability, integrity, efficiency, testability, re-usability, portability, interoperability, correctness, flexibility, structuredness, module coupling, module cohesion and completeness [4–7]. Most of these quality factors were considered by the time the product was ready for official deployment. The users were patient as the production environment was systematically and in gradual stages converted to the new software. The users pointed out the problems as they occurred and turned them over to the development staff for maintenance.

This version of the code required five corrective fixes, one perfective fix, one preventive fix, and four adaptive fixes. The corrective fixes were mostly minor in nature requiring between three and fifteen lines of code with appropriate comments added to the module to explain the changes to the original deployed code. There was only one corrective fix that required changes to multiple modules. This particular fix was urgent as it caused monitoring of systems from the SMCF product to not take place when a distinct situation occurred.

Basic cleanup of the code was performed after the fix was implemented to allow a better understanding of the fix that was employed. The cleanup was necessary to make the code more understandable before the next major change needed to take place. Understandable programs according to Martin and McClure [7] are considered to be modular, consistency of style, avoid obscure code, have meaningful data and procedure names, structuredness, are concise, and complete.

Code for the perfective and preventive maintenance required adding as few as fifteen lines to an entire module for the incidence of debugging problems when they occur. The adaptive maintenance that was performed included the addition of code to support other hardware environments and their version of the UNIX operating system using the `#ifdef` statement.

The code that was delivered in May had now changed over a short period of time. There was no consideration given to the idea of long-term maintenance when the architectural design turned into detailed design and coding. The reason for the large amount of maintenance in a relatively short period of time was attributed to the immediate need to replace the prior version of SMCF code that resided on the UNIX systems.

According to Schneidewind [8], unless the programming staff design for maintenance, the maintenance staff will always be in trouble after a system goes into production. Fenton [5] makes the point that maintenance can degrade code by low-grade staff used for maintenance, fast response to make changes, or design documentation not used. Jarzabek [9] agrees by adding that after years of maintenance, program quality drops and it becomes more difficult to do trivial modification of code.

The demand for an immediate response to problems may have had some effect on the state of the product at the time. The amount of maintenance done at this time would definitely be considered as being nontrivial. To say that the code was less maintainable was probably true but it was not unmaintainable. Making this statement is subjective as the maintainable attribute cannot be measured [5]. A definition of maintainability of software would be the ease with which software can be understood, corrected, adapted and/or enhanced [5]. It should be testable, modifiable, portable, reliable, efficient, and usable [7]. The use of software metrics to measure internal attributes of the product could give a reasonable picture of the maintainability of the code. Software metrics is also the key to improving software quality [5].

5. Analysis of code prior to restructure

To make the maintainability of the product better, it was necessary to evaluate the software using objective measures. However, there is no single test or metric that has been developed to measure overall software quality since many individual quality characteristics are in conflict [7]. So the question remains in measuring internal attributes. Fenton [5] indicates that the complexity metric is used to capture the totality of all internal attributes. Even though considered by some as not being an accurate picture of length and complexity, LOC is still used today as is McCabe's metric and Halstead's metric [5–7, 10–12].

Fenton [5] indicates that complexity increases with size. Martin and McClure [7] go even farther by saying that program complexity increases disproportionately as a program's size increases. Therefore, size of a module will be measured using different software engineering metrics. The following is a list of the metrics used to describe the SMCF product:

- Lines of code (LOC) is measured in terms of total executable, and non-executable code. Some examples of executable lines of code (ELOC) are declarations, calculations, or string manipulations. Some examples of non-executable lines of code (NELOC) are comments

spanning the entire line up until the 'newline' character or blank lines. Optimum size, depending on which expert is speaking should be between 50 and 200 lines of code [5, 12].

- Cohesion is the extent to which the individual module components are needed to perform the same task and for each module measured with a number between zero and six. This value should be as high as possible with functional cohesion being the best (value of six) and coincidental cohesion being the worst (value of zero).
- Coupling is the degree of interdependence between modules and for each module measured with a number between zero and five. This value should be as low as possible with no coupling (value of zero) being the best and content coupling being the worst (value of five).
- McCabe's cyclomatic complexity measure is the number of linear independent paths through a program and should be as low as possible with a value no greater than 10 being ideal.
- McClure's complexity measure is the number of compares in a module and the number of control variables referenced in the module. This value should be as low as possible.
- Halstead's software science module length is the total number of occurrences of operators and the total number of occurrences of operands in a module. This value should be as low as possible.

Function points will not be measured because of the subjective nature of the values assigned to the various function points. An average module size and complexity will be computed for the entire program to determine as a whole if the exercise of code restructuring was successful. The terms restructure and re-engineering will be used interchangeably at times.

Table 1 shows various software engineering metrics applied to the product after the above mentioned initial maintenance was performed. The objective of this table is to establish a baseline by which future changes and evaluations can be measured against.

Table 1. Pre-evaluation of SMCF software

File	*Module*	*ELOC*	*NELOC*	*Total*	*Cohesion*	*Coupling*	*McCabe's complexity*	*McClure's complexity*	*Halstead's software science length*
hostmsg.c		77	31	108	–	–	–	–	–
	msgthread	1001	611	1612	3	3	273	317	6348
	add_to_list	12	13	25	6	2	13	16	220
	msgCleanup	5	3	8	5	3	19	20	225
	writepip	18	5	23	5	3	14	17	269
log_svc.c.		11	5	16	–	–	–	–	–
	main	55	37	92	4	3	13	15	333
	smcflog_1	40	9	49	6	0	8	14	250
log_xdr.c		2	4	6	–	–	–	–	–
	xdr_host_logon	19	7	26	6	0	5	6	117
	xdr_logon_result	10	4	14	6	0	2	3	49
msg_clnt.c		7	13	20	–	–	–	–	–
	sendmessage_1	13	5	18	6	0	3	5	97
	hostinitcomplete_1	33	9	42	6	0	10	13	172
	hostinitcompletewithtoken_1	33	9	42	6	0	10	13	172
	sendmessagewithextra_1	13	5	18	6	0	3	5	97
	senddata_1	13	5	18	6	0	3	5	97

Table 1. *(continued)* Pre-evaluation of SMCF software

File	*Module*	*ELOC*	*NELOC*	*Total*	*Co-hesion*	*Coup-ling*	*McCabe's comp-lexity*	*McClure's comp-lexity*	*Halstead's software science length*
msg_xdr.c		2	10	12	–	–	–	–	–
	xdr_host_singlemessage	13	5	18	6	0	3	4	71
	xdr_host_message	21	7	28	6	0	5	6	134
	xdr_host_messagewithextra	16	6	22	6	0	4	5	95
	xdr_host_dataelement	11	4	15	6	0	2	3	64
	xdr_host_data	18	6	24	6	0	4	5	117
	xdr_host_initcomplete	13	5	18	6	0	3	4	71
	xdr_message_result	10	4	14	6	0	2	3	49
	xdr_data_result	10	4	14	6	0	2	3	49
cmd-xdr.c		2	5	7	–	–	–	–	–
	xdr_host_command	13	5	18	6	0	3	4	71
	xdr_command_result	10	4	14	6	0	2	3	49
	xdr_checkstatus_result	10	4	14	6	0	2	3	49
hostcmd.c		68	15	83	–	–	–	–	–
	cmdthread	45	30	75	4	4	12	21	442
	IssueCmd	416	77	493	2	4	82	117	3217
	LogSMCF	284	34	318	5	3	62	80	2512
	DebugSMCF	255	14	269	5	0	58	70	2322
	cmdCleanup	8	3	11	6	4	16	19	222
hostlog.c		54	34	88	–	–	–	–	–
	requestlogon_1	334	75	409	3	3	92	111	1586
	AddNoteToList	48	6	54	5	3	19	28	387
	GetRPCWrksFromAddr	11	5	16	6	3	15	17	180
	DeleteRPCWrksFromAddr	78	9	87	5	3	25	29	594
	VerifyIPAddr	23	5	28	5	0	17	21	280
	CheckLogin	166	16	182	5	0	58	72	1073
	smcfcmd_1	54	12	66	6	0	23	30	431
	sendcommand_1	44	9	53	6	0	23	29	348
	requestcommandtoken_1	45	4	49	6	0	23	28	334
	checkstatus_1	44	5	49	6	0	23	28	323
	terminateconnect_1	57	13	70	5	0	24	29	337
	PrintList	26	5	31	6	3	17	18	300
	PrintNextFDs	11	4	15	6	0	13	14	213
	logCleanup	33	5	38	6	3	18	20	312
	logRestart	44	5	49	5	3	20	23	390
	set_f1	12	7	19	6	0	15	18	201
	procCleanup	71	13	84	5	3	32	38	411
server.h		101	34	135	–	–	–	–	–
smcfcmd.h		30	5	35	–	–	–	–	–
smcflog.h		28	4	32	–	–	–	–	–
smcfmsg.h		83	22	105	–	–	–	–	–
strstr.c		9	23	32	–	–	–	–	–
	strstr	9	5	14	6	0	3	4	73
	strdup	8	4	12	6	0	4	5	73
Average		73.6	23.7	97.4	5.47	1.1	22.96	28.35	538.02

6. Decision to restructure

Table 1, even though enlightening with respect to the SMCF software's module length and complexity, did not force the decision to restructure the code among the programming and maintenance staff entirely. If major changes needed to be made to the code, the programming and maintenance staff felt that the current structure would not easily support such changes. The demand on the programming staff to initially deliver the product in a relatively short amount of time was the chief reason behind this feeling. Decisions made at the beginning of the project did not support long-term maintainability. Future needs in the area of maintenance were not clear at that time. Another reason for the restructuring decision included the possibility that changes in the programming and maintenance staff could occur thereby making the current state of the code difficult to understand and maintainability less than desirable. It was apparent that the programming staff were new to this type of software (i.e. RPC) and the goal of replacing the current system was a high priority. The interest of correctness was placed over that of a total quality system with maintainability built in. According to Parikh [13], amateur DP shops are more interested in a correct computer report than in a quality system. Portability of the SMCF product to other hardware was a distinct possibility. There needed to be better organization in the code to account for these hardware differences. The upgrade of the operating system software needed to be accounted for when a different compiler or internal structure changed. The product needed to handle abnormal network state changes more robustly to keep the SMCF product functioning in all conditions. Even though common and global data passing are undesirable [12, 13], some global data was needed in the product but felt by the programming staff that the amount could be reduced. Documentation in each module was limited and lacked meeting company coding standards. All of these concerns needed to be addressed and remedied with the restructure of the code.

7. Decisions to improve maintainability

It was decided to follow a plan according to Schneidewind [8] to identify candidates for restructure. The number of entry points were looked at first. Secondly, modules were examined with respect to the degree of structure, the level of nesting, the degree of complexity, break-out of verb utilization, the analysis of potential failure modes, and the trace of control logic.

Glass [2] indicates that modules, named constants, data abstraction, table-driven code, file-driven code and documentation are good preventive maintenance tactics when looking at the restructuring of code. He also indicates that use of defensive programming tactics (exception handling, assertions, margins, and audit trails) limit unsafe programming and complexity. On a down side to restructuring, Jarzabek [9] indicates that re-engineering may effect system requirements by adding new functionality to the system. The programming staff was aware that a close examination of the code may lead to new ideas being generated. These new ideas may tempt the programming staff to the point of adding the additional feature during the time of restructure, which is supposed to be a time of minimal change to functionality. The programming staff was aware that the addition of new functionality has been implemented before during other code rewrites. In fact according to Chikofsky and Cross [1], it is rare that an application is re-engineered without additional functionality being added. The programming staff had to make sure that any new functionality added to the code did not place the state of the code back in to a state of poor maintainability.

The idea of *evolution* of the code was about to occur. Evolution (or more appropriately perfective

software evolution) means a continuous change from a lesser simpler or worse state to a higher or better state [1, 4, 8]. It is correcting defects, enhancing software functionality, improving the quality of existing software. The benefits of perfective software evolution are improved maintenance skill, reduced exposure to risk, reduced maintenance costs, more time for enhancement and new development, and enhanced system maintainability [4].

It was apparent that code evolution does not occur all at one time. The better thing to do for the sake of understandability is to evolve the code gradually over a longer period of time than to make giant leaps and lots of changes at any one time and come to the conclusion that the code has evolved to its highest point.

There was no immediate need to make changes to the code at this time at the request of the users so it was determined that this was a good time to look at re-engineering the code. According to Arthur [4], a complete restructure of code or partial change of the code is done best when there is minimal change.

To isolate modules further, the first major change to the code was to move modules to their own system file depending on where it was created. The displacement of modules isolates changes to a single physical and logical module thereby lessening the possibility of other modules getting changed accidentally resulting in additional debugging. Some of these modules were written by the application development team while other modules were generated by the *rpcgen* system utility that allows for the communication to the SMCF workstation. The separation between a communication module from an application module made logical sense because of the system dependent nature of these modules. This notion of separation is backed up by Jarzabek [9] who indicates that code should be isolated removing implementation dependent details.

Documentation was added to each module as specified by the telecommunication company C coding standards describing the characteristics of each module, the algorithms used, other modules called, modules that call each module, known defects, and a revision history. Comments were added to each variable declaration.

Common sections of code functionality were broken out to separate modules. An example of this idea of separation was the retrieval of the system clock time. To go even farther, some hardware manufacturers had enhanced their API (application program interface) to format the system time thereby reducing the number of lines of code needed to complete this function. It was decided to make the code more portable by using a common system clock call API. This common system call would then avoid code dependence upon hardware vendor's enhanced system clock call. The use of `#ifdef` was employed to reduce common sections of code that would describe what was to take place depending on the type of manufacturer's hardware the code was running on. Global variables, even though required by the application, were reduced from nine to six at this time. It was felt that the number of global variables reduced was an appropriate number in the evolution of the code at this stage. Global variables would be looked at again at a future time to see if they could be reduced further.

8. Analysis of code after restructure

Each restructured module was tested individually after the change to it occurred. Sometimes it was necessary that two modules be tested together because of the functionality that was broken out of a module thereby making two modules from a single module. It was found that if no new functionality was added to a module the testing was more likely to show fewer or no errors. Integration testing was completed at the same time as unit testing. Regression testing came up with few errors and were mostly due to changes made as a result of changes to or removal of global variables. These

types of errors as a result of changes to other modules are mentioned by Schneidewind [8] and Kafura and Reddy [10]. The phenomenon is called the *ripple effect*. Testing continued until it was felt that a module's actions were stable. Stability of a module is the inverse of the ripple effect.

Table 2 shows the results of the restructure. The average number of lines of executable code per module decreased significantly showing simplification of the modules. Non-executable lines of code per module increased mainly because of the addition of documentation added to describe the modules' characteristics. Total lines of code per module dropped slightly which was viewed as being insignificant. It was not expected that a great change would occur in the average cohesion and average coupling. Only a tenth of a point was gained for both average measurements. The reason for this prediction was that the code was very well constructed using structured techniques before the restructure began. The programming staff felt that to increase these numbers significantly, global variables needed to be reduced significantly or altogether. The greatest gain occurred in the reduction of complexity of each module. The metrics show complexity of each module reduced as much as 67% and length of each module reduced as much as 50%. There is no metric to describe understandability of a module. It was the intention of the maintenance staff that by adding more comments to each module that quick searches of the modules' descriptions would yield a better understanding among the programming staff. Only time will tell if detailed descriptions inside a module as an element of the restructure was a success.

Table 2. Post-evaluation of SMCF software

File	*Module*	*ELOC*	*NELOC*	*Total*	*Co-hesion*	*Coup-ling*	*McCabe's comp-lexity*	*McClure's comp-lexity*	*Halstead's software science length*
msgthread.c	msgthread	441	360	801	3	4	85	116	3062
addtolist.c	add_to_list	19	74	93	6	2	3	5	79
msgcleanup.c	msgCleanup	18	61	79	5	3	1	3	84
writepip.c	writepip	24	62	86	5	3	3	7	133
cmdthread.c	cmdthread	66	123	189	5	4	5	21	358
issuecmd.c	IssueCmd	179	156	335	2	4	27	56	1049
logsmcf.c	LogSMCF	46	96	142	5	3	6	14	275
debugsmcf.c	DebugSMCF	10	68	78	6	0	1	2	45
cmdcleanup.c	cmdCleanup	18	61	79	6	4	3	5	84
gettime.c	gettime	84	55	139	6	0	14	17	696
main.c	main	67	114	181	4	3	15	17	353
strstr.c	strstr	11	60	71	6	0	3	5	74
strdup.c	strdup	13	57	70	6	0	2	3	61
smcfcmd_svc.c	smcfcmd_1	63	9	72	6	0	11	12	330
smcfcmd-xdr.c		2	20	22	–	–	–	–	–
	xdr_host_command	13	0	13	6	0	3	4	71
	xdr_host_data_in	13	0	13	6	0	3	4	84
	xdr_command_result	10	0	10	6	0	2	3	49
	xdr_checkstatus_result	10	0	10	6	0	2	3	49
	xdr_senddatain_result	10	0	10	6	0	2	3	49
smcflog_svc.c	smcflog_1	42	5	47	6	0	7	8	223
smcflog_xdr		2	8	10	–	–	–	–	–
	xdr_host_logon	19	0	19	6	0	5	6	117
	xdr_logon_result	10	0	10	6	0	2	3	49
smcfmsg_clnt.c		4	11	15	–	–	–	–	–

Table 2. *(continued)* Post-evaluation of SMCF software

File	*Module*	*ELOC*	*NELOC*	*Total*	*Co-hesion*	*Coup-ling*	*McCabe's comp-lexity*	*McClure's comp-lexity*	*Halstead's software science length*
	sendmessage_1	12	1	13	6	0	2	3	91
	hostinitcomplete_1	12	1	13	6	0	2	3	95
	hostinitcomplete withtoken_1	12	1	13	6	0	2	3	95
	sendmessagewith extra_1	12	1	13	6	0	2	3	91
	senddata_1	12	1	13	6	0	2	3	91
smcfmsg-xdr.c		2	32	34	–	–	–	–	–
	xdr_host_single message	13	0	13	6	0	3	4	71
	xdr_host_message	19	0	19	6	0	5	6	134
	xdr_host_messagewith extra	16	0	16	6	0	4	5	95
	xdr_host_dataelement	10	0	10	6	0	2	3	64
	xdr_host_data	16	0	16	6	0	4	5	117
	xdr_host_initcomplete	13	0	13	6	0	3	4	71
	xdr_message_result	10	0	10	6	0	2	3	49
	xdr_data_result	10	0	10	6	0	2	3	49
addnodetol.c	AddNodeToList	60	94	154	5	3	7	17	296
checklogin.c	CheckLogin	115	98	213	5	0	25	37	738
checkstatu.c	checkstatus-1	34	62	96	6	0	5	9	170
deleterpcw.c	DeleteRPCWrksFromAddr	85	84	169	5	3	11	15	485
extract.c	extract	28	75	103	6	0	5	10	150
getrpcwrks.c	GetRPCWrksFromAddr	16	58	74	6	3	3	5	58
logcleanup.c	logCleanup	39	73	112	6	3	4	6	197
logrestart.c	logRestart	53	96	149	5	3	6	10	299
printlist.c	PrintList	42	70	112	6	3	4	5	261
printnextf.c	PrintNextFDs	15	68	83	6	0	1	2	85
proccleanu.c	procCleanup	55	80	135	5	3	13	18	313
requestcom.c	requestcommandtoken_1	36	70	106	6	0	6	10	184
requestlog.c	requestlogon_1	279	154	433	3	3	45	74	1679
sendcomman.c	sendcommand_1	43	69	112	6	0	6	10	231
senddatain.c	senddatain_1	15	57	72	6	0	2	4	54
set_fl.c	set_fl	16	71	87	6	0	3	5	72
sig_child.c	sig_child	18	55	73	6	0	2	4	92
terminatec.c.	terminateconnect_1	42	75	117	5	0	8	12	220
verifyipad.c.	VerifyIPAddr	31	85	116	5	0	5	9	176
hostcmd.h		16	36	52	–	–	–	–	–
hostmsg.h		15	39	54	–	–	–	–	–
server.h		98	93	191	–	–	–	–	–
smcfcmd.h		47	12	59	–	–	–	–	–
smcflog.h		29	6	35	–	–	–	–	–
smcfmsg.h		72	18	90	–	–	–	–	–
Average		43.9	52.9	96.9	5.57	1	7.43	11.61	263.83

Table 3 compares the analyses of Tables 1 and 2. It shows the number of increases, decreases and no changes to the modules under analysis with respect to the different software engineering metrics. The executable lines of code (ELOC) showed about a 50-50 split from the modules with lines added to those with lines taken away or unchanged. Non-executable lines of code (NELOC) showed that almost all modules received additional lines. The increase in non-executable lines of code was entirely due to additional documentation added to describe the module's characteristics as specified by the company's C coding standard. The additional comment lines also contributed to the overall total of lines for each module. As mentioned earlier, cohesion and coupling values had little change. The result of the metric analysis is reflected on this table. The complexity and length for all modules either decreased or stayed the same. The number of modules with decreasing complexity is the best indicator that the restructure of the code was meaningful and accomplished it's purpose at this stage of the product's evolution. The restructure yielded eight new modules that were broken away from the original modules for functionality.

Table 3. Changes from pre– to post–analysis

Metrics	*Increased*	*Decreased*	*No change*
ELOC	27	15	10
NELOC	49	3	0
Total lines	47	5	0
Cohesion	2	0	46
Coupling	1	0	47
McCabe's complexity	0	34	14
McClure's complexity	0	34	14
Halstead's software science length	1	34	13
Number of modules	8	–	–

9. Conclusion

On the whole, the development and maintenance staff felt that the restructure of the code was worth the time and effort spent not only working with the code but also using the software engineering metrics to characterize the code. The results in Table 2 will be used again when the code goes through the next stage of its evolution in order to determine its maturity.

Strictly looking at the recommendations made by some of the authors with regard to module length and complexity, the average value in Table 2 meets or exceeds the thresholds on the side of acceptability. However, there are a few modules that do not meet recommended length and complexity values. These modules will be looked at next with respect to further restructuring and evolution. In particular, when changes or additions need to be made to these modules, there will be a strong effort to reduce the size of the modules by dividing each of them up in to two or more smaller modules.

The telecommunications company standards, concerning coding modules of an application, did not mention software engineering practices directly but made reference to defensive programming and gave twelve rules to follow. These rules had more to do with proper C coding than with software engineering quality controls. On the plus side the standards did provide a good framework for commenting a module that took more time to initially code. The consensus among the maintenance staff indicates that the extra time applied now was worth the added effort. Until the view of

programmers and managers change with respect to the prioritization of results over long-term maintainability, this type of restructuring could occur again on future products. According to Glass [2], the developer should avoid getting into the situation of having to re-engineer.

Acknowledgement

Hossein Saiedian's research was partially supported by a summer fellowship grant from the University Committee on Research, University of Nebraska at Omaha

References

1. Chikofsky E.J. and Cross H. II Reverse engineering and design recovery: a taxonomy. *IEEE Software*, January (1990) 13–17.
2. Glass R.L. *Building Quality Software* (Prentice-Hall, Englewood Cliffs, NJ, 1992) pp. 180–195.
3. Choi S.C. and Scacchi W. Extracting and restructuring the design of large systems. *IEEE Software*, January (1990) 66–71.
4. Arthur L.J. *Improving Software Quality: An Insider's Guide to TQM* (John Wiley & Sons, New York, 1993) pp. 217–231, 247–259.
5. Fenton N.E. *Software Metrics: A Rigorous Approach* (Chapman & Hall, New York, 1991) Chapters 1–3, 5, 10, 11 and 14.
6. Lew K.S., Dillon T.S. and Forward K.E. (1988) Software complexity and its impact on software reliability. *IEEE Transactions on Software Engineering*, **14**, 1645–1655.
7. Martin J. and McClure C. *Software Maintenance: The Problem and Its Solutions*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983. Chapters 3, 4, & 17.
8. Schneidewind N.F. (1987) The state of software maintenance. *IEEE Transaction on Software Engineering*, **SE-13**, 303–310.
9. Jarzabek S. (1994) Life-cycle approach to strategic re-engineering of software. *Journal of Software Maintenance: Research and Practice*, **6**, 287–317.
10. Kafura D. and Reddy G.R. (1987) The use of software complexity metrics in software maintenance. *IEEE Transactions on Software Engineering*, **SE-13**, 335–343.
11. Kelly M. *Management and Measurement of Software Quality* (Ashgate Publishing, Brookfield, VT, 1993).
12. Shere K.D. *Software Engineering and Management* (Prentice-Hall, Englewood Cliffs, NJ, 1988) Chapters 2 and 3.
13. Parikh G. *Techniques of Program and System Maintenance* (Winthrop Publishers, Cambridge, MA, 1982) pp. 9–13, 25–35, 101–104, 201–202.