Validation of Design Metrics on a Telecommunication Application

J. J. Li and E. Wong
Bellcore
445 South Street
Morristown NJ 07960-6438
jjli@bellcore.com
(201) 829-4753

D. Zage and W. Zage
Computer Science Dept.
Ball State University
Muncie, IN 47306-0450
wmz@cs.bsu.edu
(765) 285-8664

Abstract

A metrics approach for analyzing software designs that helps designers engineer quality into the design product has been developed in previous research by Zage and Zage[1]. Three of the design metrics developed are an external design metric D_e, which focuses on a module's *external* relationships to other modules in the software system, and an internal design metric D_i, which incorporates factors related to a module's *internal* structure, and a composite design metric *D(G)* that is the linear combination of D_e and D_i. Over a seven-year metrics evaluation and validation period, on study data consisting of university-based projects and large-scale industrial software, these design metrics consistently proved to be excellent predictors of error-prone modules. In this paper, we address the problem of identifying error-prone modules during the design phase of a telecommunications software system in order to detect and resolve design problems at an early stage. This paper validates the three metrics examined as effective predictors of error proneness.

1. Introduction to the Design Metrics D_e, D_i and D(G)

In the design metrics project, the research team at Ball State University began analyzing software systems to determine if identifiable traits of error modules could be uncovered during design. Our selection criteria were that the metrics capturing such traits must be objective and automatable. As presented in [1] and [2], the first three design metrics developed were called De, D_i and *D(G)*. The *external* design metric D_e is defined as

$$D_e = e_1(\text{inflows} * \text{outflows}) + e_2\,(\text{fan-in} * \text{fan-out})$$

where

inflows is the number of data entities passed to the module from superordinate or subordinate modules plus external entities,
outflows is the number of data entities passed from the module to superordinate or subordinate modules plus external entities,
fan-in and *fan-out* are the number of superordinate and subordinate modules, respectively, directly connected to the given module,

and e_1 and e_2 are weighting factors. The term *inflows* * *outflows* provides an indication of the amount of data flowing through the module. The term *fan-in* * *fan-out* captures the local architectural structure around a module since these factors are equivalent to the number of modules that are structurally above and below the given module. This product gives the number of invocation sequences through the module. D_e focuses on a module's *external* relationships to other modules in the software system.

The internal design metric D_i is defined as

$$D_i = i_1 (CC) + i_2 (DSM) + i_3 (I/O)$$

where

CC, the *Central Calls*, is the number of procedure or function invocations,
DSM, the *Data Structure Manipulations*, is the number of references to complex data types, which are data types that use indirect addressing,
I/O, the *Input/Output*, is the number of external device accesses,

and i_1, i_2 and i_3 are weighting factors. There were many candidates related to a module's *internal* structure to select from for incorporation into D_i. The aim was to choose a small set of measures that would be easy to collect and would highlight error-prone modules. During the analyses of errors for the projects in the database, the research team identified three areas as the locations where the majority of errors occurred. These were at the point of a procedure invocation, in statements including complex data structure manipulations, and in input/output statements. Thus three measures were selected, *CC*, *DSM* and *I/O* to form the components of D_i.

D(G) is a linear combination of the external design metric D_e and the internal design metric D_i and has the form

$$D(G) = D_e + D_i$$

The metrics D_e and D_i are designed to offer useful information during two different stages of software design. The calculation of D_e is based on information available during architectural design, whereas D_i is calculated when detailed design is completed.

2. The Study Data For This Analysis

The telecommunications system evaluated was the control component of a Private Branch Exchange (PBX) system consisting of nine processes. A PBX system is a switching system. Switching takes place in a time-division system by interchanging the information in two of the timeslots [3]. In our system, the process timer is used to implement the switching. To begin execution of the PBX system, the pbx process and the timer process are executed concurrently. All nine processes communicate with timer through interprocess communication. Figure 1 displays the relationships among the nine processes. The source code for the PBX system was divided into twelve C files and three header files. The system contains 99 modules and 2595 NCNB SLOC. (We use the term module generically to mean the primary design construct of a language. In C, this is a function.)

Figure 1: Structure Chart for Study Data

The metrics' validation includes four steps: 1) mapping of the architectural features of real-time systems in C to design metrics primitives, 2) metrics calculation and stress point identification, 3) testing the software to find the modules with errors, and 4) comparison of results from 2) and 3). Steps 1 and 2 were performed independently by the university design metrics research team. Step 3 and 4 were performed by Bellcore software engineers.

3. Results

Tables 1-3 contain the modules identified as stress points by D_e, D_i and $D(G)$, respectively. To be consistent with the previous research, the top 13% of the modules with respect to a given metric were considered as stress points. For this PBX system, the average D_e value is 211.9, and the average D_i value is 14.8.

De	Module	File
4650	process_CP_activation	callproc.c
1728	send_message	util.c
1659	ring_phones	callproc.c
1497	receive_digits	callproc.c
1072	receive_message	util.c
883	connect_TTRX	callproc.c
734	initialize	callproc.c
664	get_term_service_info	callproc.c
628	send_tone	tone.c
487	await_term_onhook	callproc.c
487	await_orig_onhook	callproc.c
420	main	timer.c
329	put_tone	callproc.c

Table 1: Modules Highlighted as Stress Points by D_e

Di	Module	F
59	process_CP_activation	callp
58	connect_net_path	netn
56	connect_TTRX	callp
49	receive_digits	callp
48	process_TTRX_request	ttrxn
47	process_retrieve	offi
44	scan_for_digits	ttrxs
43	get_term_service_info	callp
35	process_off_hook	callr
34	disconnect_net_path	netn
33	initialize_database	offi
32	process_TTRX_free	ttrxn
28	register_for_all_offhooks	callr

Table 2: Modules Highlighted as Stress Points by D_i

D(G)	Module	File
4709	process_CP_activation	callproc.c
1733	send_message	util.c
1685	ring_phones	callproc.c
1546	receive_digits	callproc.c
1085	receive_message	util.c
939	connect_TTRX	callproc.c
755	initialize	callproc.c
707	get_term_service_info	callproc.c
647	send_tone	tone.c
506	await_term_onhook	callproc.c
506	await_orig_onhook	callproc.c
434	main	timer.c
349	register_for_all_offhooks	callmgr.c

Table 3: Modules Highlighted as Stress Points by D(G)

A set of experiments was designed to test the PBX system. Unit testing was performed to localize the errors. Testing included three categories: basic behavioral testing, load testing, and performance testing. Behavioral testing checked for nine basic behaviors each of which consisted of twenty test cases. Load testing included five different scenarios each of which also consisted of twenty test cases. The performance testing determined whether the program was able to run continuously over a substantial length of time. The design metric stress point information was intentionally unavailable to the tester.

PBX passed all the twenty test cases of the first basic behavior. It failed in some of the second and the rest of the cases for the other behaviors. Major changes were made to *send_message* and *receive_message* for the program to pass the first seven behaviors. These two functions were identified as stress points by the external metric D_e. For the program to pass all 180 test cases, *process_TTRX_free* had to undergo major changes. This function was identified by the internal metric D_i to be a stress point.

Load and performance testing provided different results. PBX possessed a bottleneck that affected the program's capability of handling higher loads. The bottleneck was in the communication between processes of the system. The two functions responsible for failing the load test were *await_term_onhook* and *await_orig_onhook*. These two procedures were both predicted by D_e. The bottleneck also affected performance. The faulty function accountable for the poorer performance of the program also possessed a high communication volume. This function, *initialize,* was also identified as a stress point by D_e.

Besides the major changes, other functions may also have to be modified to accommodate the changes. For instance, to correct *send_message*, a parameter was added to define the size of the message sent between processes. Thus, all functions that invoke *send message* were modified to include the additional parameter. A summary of the errors found and changes made to the functions is given in Table 4.

Module	Errors	Changes	File	Module	Errors	Changes	File
allocate_net_path	1	24	netmgr.c	process_off_hook	2	25	callmgr.c
allocate_path	1	6	callproc.c	process_on_hook	3	33	callmgr.c
await_orig_onhook	4	40	callproc,c	process_query	1	16	linescan.c
await_term_onhook	4	40	callproc.c	process_retrieve	1	29	office.c
connect_path	1	12	callproc.c	process_TTRX_free	7	38	ttrxmgr.c
connect_TTRX	4	194	callproc.c	process_TTRX_request	3	33	ttrxmgr.c
deallocat_path	1	7	callproc.c	put_tone	3	60	callproc.c
disconnect_path	1	7	callproc.c	receive_digits	2	85	callproc,c
fatal_exit	1	17	util.c	receive_message	1	12	util.c
free_TTRX	2	14	callproc.c	register_for _all_off_hooks	2	39	callmgr.c
get_orig_service_info	1	32	callproc.c	register_hook_action	1	9	callproc.c
get_term_service_info	2	75	callproc.c	release_cp	4	75	callmgr.c
inform_tasks_offhook	1	22	linescan.c	release_CP	2	24	callproc.c
inform_tasks_onhook	1	22	linescan.c	scan_for_digits	2	58	ttrxscan.c
initialize(callproc)	3	37	callproc.c	send_message	2	9	util.c
main(timer)	3	108	timer.c	send_tone	1	34	tone.c
process_CP_activation	12	122	callproc.c	stop_tone	1	27	tone.c
Totals				**Totals**			

Table 4: Modules with errors

When the results of the testing and the stress point prediction were compared, the design metrics pointed to the error-prone modules. For example, when 13% of the modules were identified as stress points by the design metric D_e, 92% of the 13% highlighted modules had errors (12 out of 13). Moreover, these modules (13%) contained 51% of the detected errors. The metric D_e performed slightly better than D_i in predicting stress points in this system because this application involves large amounts of inter-process or inter-module communications. The structure within a module is relatively simple, whereas the external logic is relatively more difficult. The error proneness of a module is more related to the external logical complexity than to internal structural complexity in this application. When D_i highlighted 13% of the modules as stress points, 77% of these modules contained errors and the percentage of the errors identified was 47%. D_i is effective in predicting stress modules that have less communication with other modules. Similar results were obtained with the composite metric D(G). (See Table 5.) The last column in Table 5 indicates the number of modules highlighted by any of the design metrics. Note that the modules highlighted by at least one of the metrics contained 72% of the detected errors.

	De	Di	D(G)	All
Modules Highlighted	13%	13%	13%	22%
Highlighted Mods w/errors	92%	77%	92%	82%
Detected Errors in Highlighted Mods	51%	47%	49%	72%
Changes to Highlighted Error Mods	59%	53%	57%	75%

Table 5: Comparing Error Modules with Predicted Stress Points

5 . **4. Summary and Future Directions**

D_e is the most effective single metric for identifying stress points in the application studied. When reviewing the modules highlighted by at least one of the design metrics, the locations of 72% of the detected errors were identified. Therefore, these design metrics can also be used to assist telecommunications software development.

A design team would typically review the list of stress points to determine if a redesign of any module or file is in order. Even if the decision is made to leave the design intact, we now have detailed design information that can be used to determine where to place additional testing effort. Knowing the structural stress points is also useful if we later expect to add new functionality to this system.

This analysis was based on a standard design metrics model and was not fitted or tailored for this application. Not only did this evaluation support the early findings of the design metrics as highlighters of error-prone modules, but also the design metrics may be used to support performance models.

Another area worthy of investigation is the possible customization of these design metrics to various software architectures. The software architecture defines a system in terms of its computational components and the interactions among those components [4]. Some of the structural elements in software designs are composition of components, global control structures, communication between components, protocols for communication, synchronization, and data access. All of these elements can be mapped to the design metrics of inflows, outflows, fan-in, fan-out, CC, DSM and I/O. This mapping allows designers to compare architectural designs of all types and apply the design metrics technology to various software architectures, as we have done in this analysis. Explicitly recognizing and isolating particular patterns of a software architecture may provide added benefits in software development.

5. References

[1] Zage, W. and D. Zage, "Evaluating Design Metrics on Large-Scale Software", *IEEE Software Journal*, Vol. 10, No. 4, July 1993.

[2] Zage, W. and D. Zage, "Relating Design Metrics to Software Quality: Some Empirical Results", SERC-TR-74-P, May 1990.

[3] Bush, S.E. and C.R. Parsons, *Private Branch Exchange Systems and Applications*,

McGraw-Hill, 1994.

[4] Shaw, M. and D. Garlan, *Software Architecture, Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.

¤§ ¨ © ª