A Group of Design Principles and Testing Methods for Improving Software Quality

Zhang Ya_qi[1, a], Jiang Zhi_ping[1,b] and Lin Yan[1,c]

[1]Courtyard 10[#],Anxiang North Lane, ChaoYang District, Beijing 100101, China

[a]Wingwinglili@foxmail.com, [b]gfbddl@msn.cn, [c]15652215681@163.com

Keywords: Software Design, Software Quality, Software Test

Abstract. This paper raises a group of software principles, which have higher operability and can improve the robustness, reliability, maintainability, etc, of software, as well, the article gives specific advice on testing method for each principle.

Introduction

Software quality is a collection of characteristics which are relevant to the specified requirements or implied capabilities of the software, including validity, reliability, property, easy-using, security, compatibility, etc [1]. As a kind of invisible electronic product, software gets more influence from the environment and the user compared to the hardware, their qualities are more decided by the preliminary argumentation, the design and the development stage. Researches have shown that, it needs lesser cost in improving software quality in the early stage of software development, which brings a larger investment income, as shown in Fig 1 [2]. As the bridge between requirements and developments, software design has the most important impact on the quality of the software.

This article focus on the optimizing of software design itself, raising a group of principles which have a higher operability, so that the robustness, reliability, maintainability, etc, of the software can be improved. These principles could be practiced in software engineering, and can also be confirmed in software testing phase.

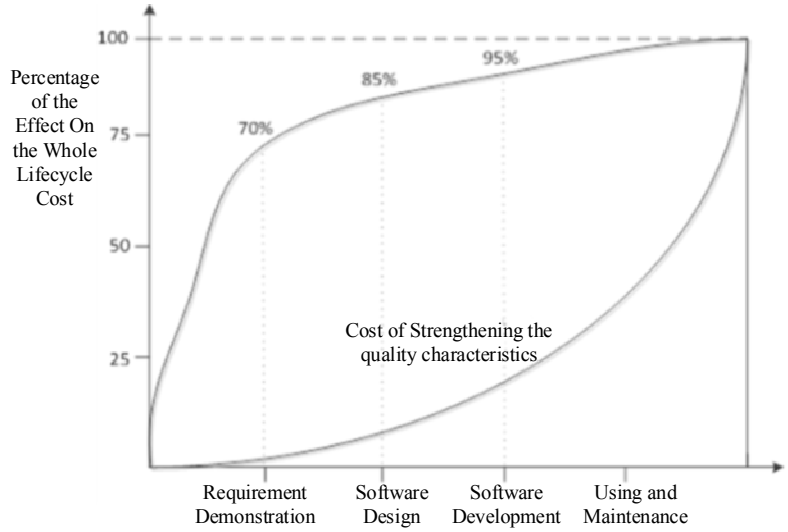

Fig 1 Contrast between the cost in quality and the effect on lifecycle cost

Source Of The Design Principles.

Since software has no producing link, most of qualitative problems of the software are caused by the defects of software design. Software Failure Mode Effect Analysis (SFMEA) is, through identifying the failure mode of the software, analyzing the cause of the failure and the serious consequences will be caused, so that the method of eliminating the harm can be searched. Applying SFMEA in the design stage of software developments, could prevent potential faults and put forward corrective actions in time.

With the development of internet, big data, etc, which are applied in all areas, software becomes more complicated and modularized, increasingly demanding on software normalization and reliability. In this background, the common failure modes are summarized from the practice: a) Correlation fault between software modules; b) A longtime running failure; c) Environmental adaptability problems; d) Operation fault.

Software design methods are reversely deduced from the failure mode to avoid the fault in the practice, and the design principles are summarized [3]. Other then, for a long time, there are no widely recognized effective methods in software quality testing, the developers pay more attention to the realization of the function and the performance index, but pay little attention to the software quality design. Therefore, the design principles putted forward in the article pay special attention to testability, and test methods are given for each design principle [4].

Design Principles

Loosely Coupled Association

Description. Association is the interaction between the two function entities (processes, thread, etc). This connection may be a kind of forced or loose dependence, existing at the start, or while running. A,B is the function entity, A depends on B, signed as A→B, which means part of A functions need assist in the completion of B, if B failed, A functions will not be fully available. Dependence is an objective existence, could not be eliminated, but there are also tight coupling and loose coupling which have real difference.

The tight coupling dependence performs as: a) If B does not start, then A starts failure; b) If error occurs in B running, A could not identify the error, and then part of A functions are failed; c) If B restarts, A could not recover the connections and affairs with B, and there is a fault.

The loose coupling dependence performs as: a) If B does not start, A still can start up ever, monitoring B according to certain rules, and can set up connection as soon as B starts, at the same time, A can give the correct feedback to the user's request which needs B's assistance; b) If there is a fault during B running, A can give a clear declaration. c) If there is a restart of B while running, A can also detect it through regularly monitoring or just at the visiting time, then recover the connections and affairs with B, to ensure the request being answered.

Absolutely, the loose coupling dependence blocks up the spreading of the fault, which makes the system become stronger and more reliable, easier to be maintained, and function entity can upgrade and evolution separately. This principle especially adapt to the systems which have complicated associations.

Testing Method. First, identify the function entities in the software, supposing the set of the function entities is G, $G = (G_1, G_2, G_3, \ldots, G_n)$. Pairwise these entities to get C_n^2 records, identifying whether there is an association between each two entities or not, table 1 is formed.

Table 1 Associations in the software

No.	Depending entity	Depended entity	Start dependence	Running dependence	Remark
1	G_1	G_n	N	N	$1 \leqslant m, p \leqslant n$
…	…	…	…	…	
m	G_m	G_p	Y	Y	
…	…	…	…	…	
n	G_n	G_1	N	Y	

Note: 'Y' means that the association does exists, 'N' means that the association does not exists.

Design testing case U_m^p: i) Keep G_p not start, then start G_m, if G_m fails to start, test does not pass; ii) If G_m starts successfully, run functions of G_m one by one, if the function is not associated with G_p, it can be normally used, otherwise would warn user that the function can not work well as a result of abnormal G_p.The above is not satisfied, the test does not pass; iii) Start G_p, then run the functions of G_m, if no error occurrs, test is through.

Service Has Daemon

Description. For the coral and key sevice of the software, once the service stops or loses response, it will cause almost all functions of the software to fail, processes to terminate, business to be paralysed and other major losses. To ensure continuous operation of such services, daemon is always designed to monitor and protect the service.

Assume the set of the daemon group entities is W, $W = \{(F_1, f_1),\ldots,(F_i, f_i),\ldots,(F_j, f_j),\ldots,(F_n, f_n)\}(1 \leqslant i, j \leqslant n)$, F_i represents the key services, performing normal business logic, f_n represents daemon, monitoring the state of F_i, not containing business logic. Once f_i monitors F_i stopping, it immediately restarts F_i or performs the repair program until F_i returns to normal. For a special key service F_j, a two-way guard strategy would be taken. In addition to the implementation of the normal business logic, F_j also monitors the state of f_j. If f_j ran abnormally, F_j would restart or repair f_j, making itself enter in the protected state once again.

With the establishment of daemon group, the possibility of disruption could be reduced, the duration time of service be prolonged, and the rubustness, reliability and security of the system be improved. This principle is especially adapted to those software and services who are most important, have serious influence, and need to run continually.

Testing Method. First, identify the coral and key sevices of the software with W_1, $W_1 = \{F_1, F_2,\ldots,F_p,\ldots,F_n\}(1 \leqslant p \leqslant n)$, pick out the special important services in W_2, $W_2 = \{F_i,\ldots,F_j\}(1 \leqslant i, j \leqslant n)$.

Design testing case U_p for each F_p in W_1: i) Stop F_p, perform the function of the software until error ocurrs, then close the software; ii) Stay and wait for about 1 minute or so, then restart the software and perform the function which has just failed, if error still appeared, test would not pass; iii) Continue to perform the function of software business, if no error occurs, test pass, while if there were errors, test would not pass.

Design testing case U_i for each F_i in W_2: i) Stop the damon of F_i, and then perform functions of the software, if error ocurrs, test does not pass; ii) Stay and wait for about 1 minute or so, then stop F_i, perform functions of the software until error ocurrs, then close the software; iii) Stay and wait for about 1 minute or so, then restart the software and perform the function which has just failed, if error still appear, test does not pass; iv) Continue to perform functions of software business, test could pass only if no error occurred.

Resources Have Allowance

Description. Resources refers to the objects of system which software may write or create in the process of starting and running, including memory, disk, handles, threads, database, application connections, etc. Resources have allowance means, the software should ensure that allowance of remained resource could satisfy the user‘s requirement before applying. Identify the collection of system resources with Z, $Z =(Z_1, Z_2,\ldots, Z_n)$, if the operation apply for Z_i $(1 \leqslant i \leqslant n)$,firstly the software should judge whether Zi can be accessed or written in, and the allowance of Z_i is enough or not.

This principle can effectively avoid the disk being written full, memory being overrun, space being insufficient, the number of connections beyond limit, and such failures as use out of resources, in this way to improve the system security and environmental adaptability. The principle is especially applicable to the software which has regular, continuous, large amount of data creating or writing.

Testing Method. Assume the collection of operations related to Z_i is C_i, $C_i = (C_{zi}^1,\ldots, C_{zi}^j,\ldots, C_{zi}^m)(1 \leqslant j \leqslant m)$. Design testing case U_i for C_i: i) Locking Z_i for it can not be used, perform the operations from C_{zi}^1 to C_{zi}^m, if abnormal errors occurrs, test does not pass; ii) Start another program, all occupying Z_i, perform the operations from C_{zi}^1 to C_{zi}^m, if errors occurrs, test does not pass.

Activity Is Recoverable

Description. Activity is the operation or task which makes the system state change. Activity is recoverable means that, if the software operation is suspended or illegally closed due to force majeure, when the software restarted, it could return to the state just before shutting off, parameters inputted could be reserved, while if not, give specification.

List all activities of the software in collection A, $A = (A_1, A_2,\ldots, A_i,\ldots, A_n)(1 \leqslant i \leqslant n)$, the activity diagram is shown as Fig 2. Software can record the states before and after the completion of A_i,

information user inputted can also be recorded. Once system interruptted while A_i operating, when restart, the software would judge whether A_i has been completed, if yes, it will recover to the state after A_i being completed, else then recover to the state before A_i beginning. Information user inputted will be restored at the same time, and if failed to save, prompt the user.

This principle is good to avoid business interruption, service suspending, data losing and so on, which are caused by system restart due to external factors, so that, the security, environmental adaptability and user friendiness of the software are improved. The principle is especially applicable to the service or software which have longer business process, more input information, larger amount of data processing, and have a higher requirement for continued running.

Testing Method. Design test case U_i for each activity A_i: i) Implement A_i, completing all operations of Ai, then restart the system, when the software starts-up again, judging whether it jumps to the state that Ai completed. If not, the test is not through; ii) Implement Ai, shut down the system in the process of inputting information, when the software starts-up again, judging whether it jumps to Ai and preserves the information inputted, or prompts user that preservation of the information is not successful, if not, test is not through.

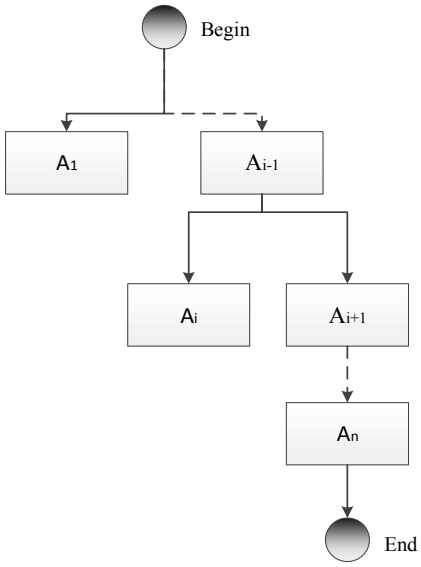

Fig.2 Activity Diagram of Software

Data Can Be Separated

Description. Data refers to which software need to load, input, store, output, including formatting data and unformatted data. Data can be separated means that, the installation, uninstallation and updates of software do not cause data loss or damage, data can be independently loaded, updated and deleted.

Recognize operations relating to data, including software installation, start-up, update and unloading, etc, forming operations set D, D = (D_1, D_2, D_3, D_4,..., D_n). Assume that D_1 represents software installation, D_2 represents the software start, D_3 represents software update, D_4 represents software uninstallation, D_i($5 \leqslant i \leqslant n$)represents other data related operations.

D_1 can be performed successfully without loading data; D_2 can be performed successfully with no data, prompting the user some function is not available without data; Implement D_3 after loading data, when the update is completed, loaded data can still be used; Delete data manually, perform D_i operations, prompting the user when the operation appears error because of data, but it will not cause the software to collapse; Perform D_4 operations, prompting the user whether to delete the related data, user can choose according to need.

This principle can reduce the software errors caused by data unloaded, data updated or data errors, it can also avoid data loss or irreversible phenomenon caused by the software errors, effectively improving the reliability, maintainability, supportability, testability, safety and environmental

adaptability of software. The principle is suitable for the software which need data or generate data during installation and operation, especially suitable for those who have large quantity and continuity of valuable data.

Testing Method. For the installation of the software (D_1), start (D_2), update (D_3) and unloading (D_4) and other data related operation (D_i), design test cases U_i: i) Perform D_1, if the software prompts the user must load data, else it is unable to complete the installation, test does not pass; ii) Perform D_2, if the software can't normally boot, test does not pass; iii) After loading data, implement D_3, if no data have been loaded after restart, test does not pass; iv) Delete data manually, perform D_i, if software appears mistakes, the test does not pass; Reload the data, performs Di, if software appears fault or is unable to read data, test does not pass; v) Implement D_4, if the software does not prompt user to choose retain data, test does not pass; Choose to keep the data, finding whether data file is in relevant path, if there is no file or the size of file is 0KB, not through the test.

Status Can Be Monitored

Description. Status is software running status, including the throughput of thread, the transaction efficiency, professional parameters, such as the error logs. The setting of monitor items is closely related to the software business logic. Status can be monitored means, status data of software running can be able to grabbed and gathered, and can be centralized in the visualized interface in real-time, when in case of congestion, service stopping, and other emergency, there will be alarming prompt for user according to certain strategy.

In this way of warning or rapidly detecting the dangers, large area faults are effectively avoided. This principle is applicable to those software who run continuously, have higher request for stability and reliability [5].

Testing Method. First, list all monitoring items of software in set J, $J = (J_1, J_2,..., J_i,..., J_n)(1 \leqslant i \leqslant n)$. For each J_i, design test case U_i: i) Run business functions corresponding to J_i frequently and vastly, increasing the amount of data exchanged, and then observe whether J_i perceives or hints. If there is no obvious change and prompt of J_i, test does not pass; ii) Stop service of J_i, observing monitoring item J_i, if there is no alarm prompt, test does not pass.

Postscript

The design principles proposed in this paper come from the software engineering developments practice, focusing on improving the quality of software. It provides new ideas for software design and testing of reliability. In practice, it should be appropriate chosen combined with the specific circumstances of the software, such as functional requirements, operating environment, importance of the software and so on, to deal with the balance between efficiency and resource consumption.

Summary

This paper raised a group of software principles, which have higher operability and can improve the robustness, reliability, maintainability, etc, of software, as well, the article give specific advices on testing methods for each principle.

References

[1] Zhou Jian: *Research on quantitative evaluation of software quality,* University of Electronic Science and Technology, (2007).

[2] Li Jian_hua, etc: *Research on software measurement system and implementation method based on process* (Proceedings of 16th National Youth Conference on Communication).

[3] Huang Chao: *Research on Software Reliability Modeling and prediction method based on Chaos Theory*, University of Science & Technology China, (2010).

[4] DoronA.Peled, Israel: Software Reliability Methods, China Machine Press (2012).

[5] Zhou Ming_hui,etc: *New thinking of software engineering based on big data*, CCCF (2014).