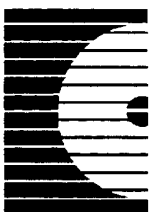


IEEE Communications Society REPRINT

**STRUCTURAL WAY OF THINKING AS APPLIED TO GOOD DESIGN
(PART 3, REDUCING PROGRAMMING ERRORS)**

**Zenya Kono
Homayoun Far Behrouz
Yasukiyo Yamasaki**

Reprinted from IEEE Communications Society
"IEEE Global Telecommunications Conference"
December 6–9, 1992



**IEEE Communications Society
345 East 47th Street
New York, NY 10017**



THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC

Structural Way of Thinking as Applied to Good Design (Part. 3, Reducing Programming Errors)

Zenya Koono, Homayoun Far Behrouz and Yasukiyo Yamasaki

Information and Computer Sciences, Saitama University,
Shimo-okubo, Urawa 338, Japan

Abstract This paper explains programming errors from the view point of human errors. Programming error problems may be analyzed by separating two factors "build-in" during design and various "check-out" following the design. Though separating error problems, key points for reducing programming errors may be reduced to how to make "build-in" rate small and how to make "perfect check-out". Error problems during design, testing and the progress of quality are explained. Various suggestions for reducing errors are given.

1. Introduction

This paper describes a structure of programming errors concentrating on human errors, rather than those frequently reported "technical" problems.

In classical Software Engineering, though many theories on design methods exist, rational approaches and theories on reducing errors have not yet been established. The difficulty comes from the fact that errors are the result of too many factors residing in and surrounding a development.

Software error problems have been rigorously studied in Total Quality Control (TQC) small group activities. TQC features the quantitative and rational approaches based on cause and effect relationship. They lead to various improvements in their developments, and the results are published extensively. The situation is like thousands of Software Engineering people study software development process. Through these reports, it becomes clear that programming errors are caused by human errors not by the difficulties of the problem or the resultant program.

2. Design error

2.1 Error characteristics in a coding process

An experiment on "coding" is explained as an example which was made during an actual project. This was conversions from a pictorial symbol to a line of code. The number of symbols to be converted was around six thousand.

Coding by a new rule is error prone. It is well known that checking after coding reduces errors. But these checks also suffer from errors. In hardware industry, it is a usual practice to divide an error problem to "building-in" and "checking-out" errors. In order to achieve a low error rate in this project, it was decided to do uniform controlled works observing error check-out. Workers coded carefully referring to a conversion rule book but finished with rough checks after each conversion. After completing all the conversion, they then made rigorous checks referring to the rule book. Six rounds of checks were necessary until the accumulated number of checked-out errors became saturated. (One error was found in the actual operation of the system later.) As is shown in Figure 2.1, the error "build-in" rate and error "check-out" rate were considerably larger than they were suppose to be. This experiment shows that coding or "building-in" errors and "checking-out" errors are different processes. Though both are results of human errors, different managements are necessary. Therefore, an error problem should be decomposed to an error build-in problem and a check-out error problem. They should be differently discussed.

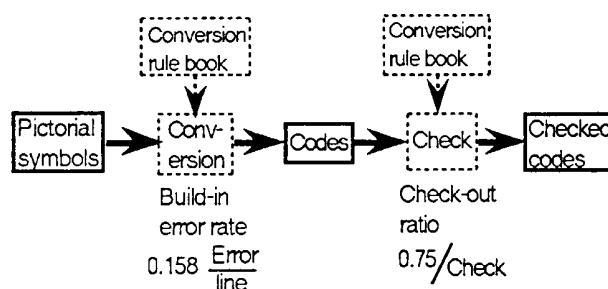


Figure 2.1 A coding experiment

The ideas of reducing errors are summarized as follows:

*Error rate is a result of a process. There are two kinds of errors; errors of "building-in" and errors of "checking-out" these errors.

*The former is measured by a "build-in error rate" (probability of building-in errors), while the latter is measured by an "error check-out rate" (in this case, probability of checking-out errors at one check).

*The former is built-in unknowingly and inevitably at once. A check also errs to check-out errors inevitably. Both rates depend upon processes in concern. But for checks, the final check-out rate is controllable.

*Repeated and rigorous checks are necessary to attain a final low error rate. In order to assure a low final error rate, oblige people to check-out, allowing man-hours for checks.

2.2 Error characteristics at program level

Actual error characteristics at program level are discussed next. Figure 2.2 shows an example of years of experience vs. normalized program error rate.[1, 2] The data is taken from an initial development (all participants had no previous advantages) of a system using a high level language for the first time (i.b.i.d.) at the end of the assembler age when desk checks were not obligatory. In Figure 2.2, black bars show the error rate checked-out at machine tests as is usually done. No meaningful trends are observed from this figure.

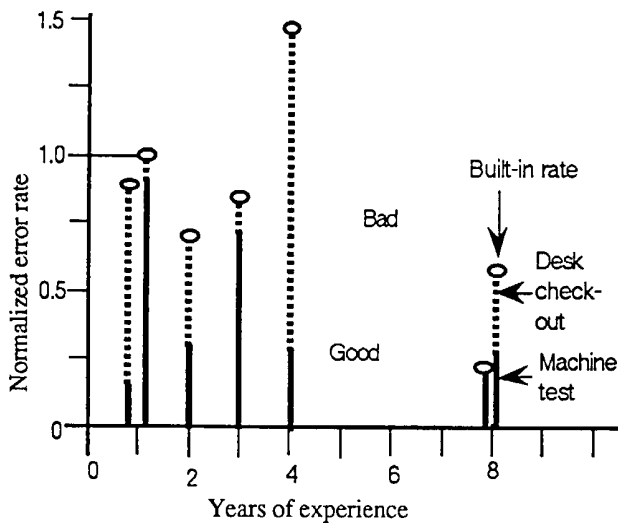


Figure 2.2 Years of experience vs. error rate

White circles in Figure 2.2, show the built-in error rate, which is the sum of program error rates found at machine test (black bars) and those of desk checked-out prior to machine test (dotted bars). The trend of white circles is designers' intrinsic build-in rates, and it reduces as designers gain experience. Plots agreed with the designers' evaluations (good and bad) as attached to the figure. The ratio of dotted bar/(black bar + dotted bar) is a check-out ratio. In this example, desk checks were not obligatory and the check-out ratio were at random resulting in that black bars variate wildly giving a meaningless result.

Apparent (built-in and checked-out) error rate is like a clinical thermometer. It is useful for practical surveillance, but does not tell what is wrong. If the problem is decomposed to error build-in and check-out, the cause is studied form measurements and the countermeasure may be obtained from cause and effect relationship.

Figure 2.2 depicts a rather ideal case. As this was the initial development, (the application and the language), all the participants showed their initial error build-in rate. Usually, experienced people show a lower build-in rate than those shown in Figure 2.2, due to their experience and familiarity with the system. The build-in rate for beginners is around 100 errors/kiloline due to their lack of experience and knowledge in both programming and the programming language, while this rate for experts depend upon the limit determined by the development process (people, methods, documents and environment) used. In actual developments, random error rates are observed. If causes is studied on build-in and check-out, the situation becomes clear. Through these analysis important know-how for project management is gained.

2.3 Improvements

Quality ("build-in" error rate and error "check-out" rate) is governed by the developing process which is performed by people, therefore quality is governed by the people concerned. The quality of a team when people get together for the first time is generally poor. Even if each person is highly skillful, it does not guarantee that the new project goes in a right way. It is improved when the people are united in the new process and are accustomed to the new way. Quality progress depends upon the people's progress. Thus the progress is governed by the human rate of progress which is generally slow. [7] The maturity of development organization [4] has a deep connection with people's maturity.

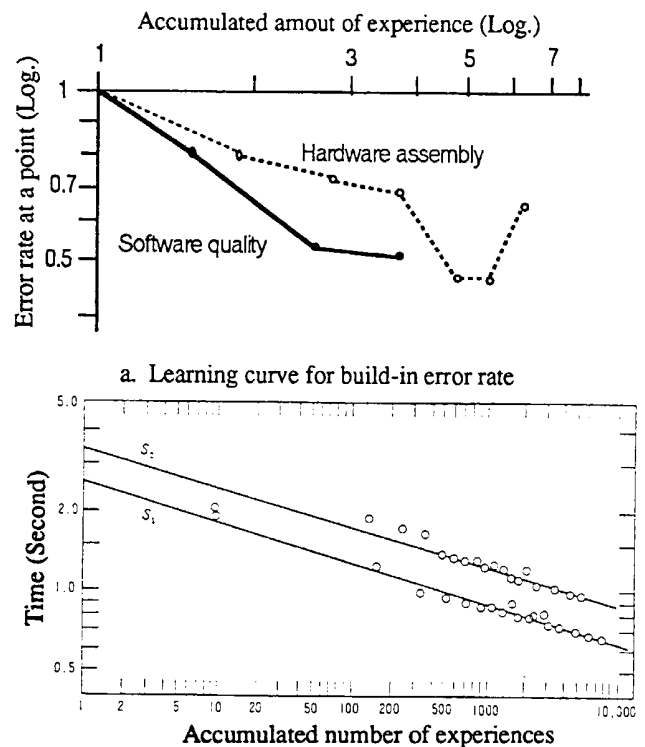


Figure 2.3 Learning curves for mental operations

Progress follows a learning effect as shown in Figure 2.3. [5] The horizontal axis shows the accumulated experience and the vertical axis shows the instantaneous quality ("error build-in" rate at a time point) both in logarithmic scale. (In linear scale, the improvement growth is first rapid and then gradually becomes slow.) When learning occurs, plots become linear. Not only build-in error rate but also check-out rate and total error rate show linear learning curve so far as people continue to improve processes. As the process is a human mental process, the curve shows a similar gradient to other human mental processes [7] and software productivity. [9] If processes are not improved, the quality is not improved. As the improvements are made from past experiences, the improvement is a feedback process. As the improvements requires improvement of people, the key point is how to motivate people to improve or change themselves together with environments, methods, rules, procedures and practices.

Usually, all the people in a development are confident that they are doing their best in the environment. In order to improve their process, it is necessary to make them see what are the problems, and motivate them to make efforts for improvement together with their bosses who exert influence upon people. Through recognition of problems and with proper motivation, they will improve their process, namely themselves. The effective way to do this is to recognize problems and to try to find out countermeasures. The systematic procedure has already been established in TQC.

The most important primary information in the process is where it was "built-in". By tracing the cause and effect chain, like

- *from a customer's claim to the erroneous program behavior,
- *the erroneous program behavior to the program (data) error,
- *the bug to the work process where the bug was built-in,
- *the erred process to the cause which brought the error,

and then the countermeasure so as not to repeat the same kind of error may be obtained. Although program errors are built-in during design, the cause might be the programmer's imperfect knowledge, poor documentation, insufficient tools, lack of man-hours, too tight a schedule or improper specification. Managements also has influence on quality. The same tracing and establishing the countermeasure must be made on "check-out". Documents play substantial roll in build-in and check-out. The correct cause must be sought by the cause study, and the corresponding countermeasure must be established and all the people (including bosses) must behave in the improved way in the next project. This feedback is essential for quality progress.

Thus following are keys for improving program quality:

- *Program quality, error "build-in" and "check-out", is a reflection of the processes concerned, or reflects the people concerned. It is not improved without improving process including people toward less error rate.
- *Improve development work processes by cause analysis and devising preventive means no to repeat. Standardize all the document rules as well as work processes. Teach and train people to keep standards and rules.
- *As principles of education teach us, the improvement is greater when the feedback is sooner, stronger, wider and lasting longer. The mechanization to a system is the best. The most of processes, however, remain human works. So strong discipline through continual education is necessary.

*The improvements are first rapidly growing and then the rate decreases. Continue to make efforts repeating improvements. The best way is to use "management by objects". Set a target which seems attainable with efforts, devise means to attain it, do works and evaluate them contradicting with what was planned and repeat the cycle.

*As improvements follow changes in people's work procedures, they accompany some pain. The most important problem is the motivation of people in improvements. The best way is to make them participate in improvements and realize them to become improved. Detailed practices are described in TQC small group activity and management books.

As participants gain success experiences, their skill in finding correct cause and effects based on quantitative data advances. As the result, people are advanced. The problem does not remain a technical problem.

3. Program design errors

Based upon the previous knowledge, how to reduce programming errors is discussed. Let us recall a situation when we were young. If high school students are asked to solve a difficult mathematical problem without errors, they will write diagram(s) to make the solution clear and will leave all traces of the solving (transformation of the problem) processes in detail, and will later check them carefully and repeatedly. The above example includes some of the key factors:

- *diagram(s) for recognizing the problem and the solution,
 - *small transformation steps for easier thinking as well as for checking,
 - *leaving all the traces of the transformation for checking, and
 - *rigorous and repeated checks
- are used aiming at lowering errors. These are rules established by years of human experience. As programming is another kind of human mental processing, the same applies to program design.

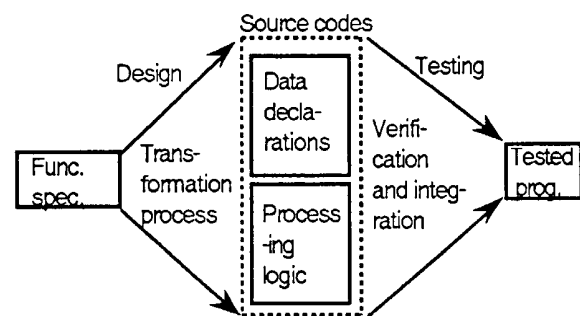


Figure 3.1 Design and test

Figure 3.1 shows the design and the test. The input is a functional specification, and the output of the design is much more complex source codes consisting of data declarations and processing logic statements. Programs are tested and integrated to a system. A development is a huge hammock consisting of numerous micro mental processing.[3]

During each micro mental processing, designers err probabilistically at a small rate. It is necessary to check each step of simple information transformation steps not only for logic but also for data.

The majority of program errors are built-in by carelessness and poor checking allows them to remain. The difficulty of a problem, being one of the causes of errors, does not necessarily result in more errors as is thought to be. It invites a complex solving process. If the design is made without paying enough precautions, more errors will be built-in and less errors will be checked-out. But when the process is divided to small enough and each steps are recorded and checked carefully, the situation can be made equal to designing easy programs. What governs quality primarily is the process in concern not the problem to be solved. Thus following are keys for reducing errors;

- *to leave each traces of design in documents, and
- *check small design steps by documents repeatedly.

Many people do not like or do not leave documents. There are two reasons, the first comes from the human psychology and the other is due to the productivity.

The psychological aspect is explained using learning how to shoot a golf ball. At the beginning, one will learn each elemental motion; like grip, step stance, take back swing, swing, finish and so on. At first, one will shoot balls rather unnaturally considering each motion. At this point, one can know what was wrong when a ball was shot in the wrong direction and correcting the motion by feedback. As exercises are accumulated, one's action becomes more smooth, and finally one begins to swing with a full motion automatically and continuously. This is called "automation (or skill)". After this point, one's consciousness of each motion element disappears.

- (1) After this, most people can not distinguish what was wrong when they shot a ball in the wrong direction.
- (2) Only a few exceptional people, who continue to be conscious of each elemental motion, can distinguish which action was the cause of the wrong shoot. Therefore only such people can correct their elemental motion and become a good player quickly, while the majority of people stay in poor ability.

The situation is quite the same in software design, as the human mechanism is the same. As a designer becomes accustomed to design, he begins to feel that he can see the final program at the instant he reads the specification. He can write the program without intermediate documents as experts do. He has become an expert. But at the same time, the following danger awaits him.

- (1) After programming has become a skill, a designer can see the resultant program directly. Now, it is tedious for him to record traces of the design. As he is accustomed to design without documents, his consciousness on intermediate design steps is lost, finally it disappears. Then, he can not write intermediate design steps even asked.
- (2) Suppose he made errors during design. When they are checked out at the machine test later, he can not recognize which part of his past process was wrong as he passed through automatically, and as the result he may not be able to correct the process. (Thus, only a few people show rapid progress, as is the case of playing golf.) If he had left his design traces (documents), checked out errors and corrected his process by studying the causes of these errors, not only his burden would have been decreased but also his skill would

have been improved. This has been noticed in TQC small group activities and also in education of programming.

In TQC, in order to achieve high quality, it is more important to establish a correct process rather than an efficient but low quality process. If a correct process is established, it is possible to rationalize it by some aids resulting in higher productivity. However, speedy but defective processes are not easily improved in their quality keeping its speediness. "Quality precedes productivity" or "quality first" is an important principle.

For achieving a low program error rate, it is necessary to improve processes. It is needed to locate poor quality process and improve them. The following are detailed key points taking human error build-in, check-out and the learning effect:

Before starting a development

*Precede quality rather than productivity. Checking precedes schedule. If schedule precedes quality, more man-hours will be lost at some later test. Allocate reasonable man-hours for checking.

*Fix work processes by fixing document (interfaces of each work process) and check-list standards resulting in fixing standardized procedure for the work. Oblige people to write documents and check them by check lists.

During the development

*Make checked-out error record by designers. Gather checked-out (by desk checks or machine tests) error reports. Tabulate these raw data to hierarchical structures of the system, the developing team, the work processes. These are important data for surveillance and provide very useful information for leaders. They can see a macro view from the hierarchical table, can get individual problems from error reports and can ask a designer focusing problems directly.

After the delivery

*Gather field detected faults until they saturate. (The field quality may be obtained from them.) They must be identified where built-in and where should be checked-out according to detailed intermediate documents left.

*Add field detected errors to the corresponding data during its development. Calculate the exact built-in rates and check-out rates of each program, each work processes, development group, team and people. Hierarchically tabulate them and study the list. Build-in rates as well as check-out rates reflect the quality of the design process (including check-out methods and check-lists), designers, team leaders and the manager.

*These data clearly show what are major weak points in building-in errors and checking-out errors. Erroneous processes will be found by build-in error rate classified by processes. Similarly, poor check-out processes will be found. By hierarchical data, poor quality team will be located, and the cause is something they share in common, such as work practices, the management or their environment. From personal data, individual skill may be evaluated quantitatively.

For improving quality of coming developments

*Find out respective cause of building-in or leaving errors and improve processes, otherwise quality is not improved. (Quality Control) It is necessary to let people review their own data and devise improvements by themselves. This gives an effective feedback to all. (Motivation by participation)

*Repeating this, designers grow up much rapidly as they face what is wrong with their products. On studying and accumulating the respective data, team leaders can estimate a designer's build-in rates at the start. Those who make such quantitative studies, acquire technical and managerial skill which ordinary leader can get after a long time. Their products' quality becomes better as they continue to improve their processes.

*Raise team's total check-out rate as the team target, and make each designer raise their improved build-in rate as their respective target. This must be made based on the quantitative studies of past experiences by themselves. Challenge a new project with the new target and the improved process. (Management by objects)

*After the completion, do the same analysis and find other ways to improve the process including people. Repeat these plan-do-check-action cycles.

These are the principle of Total Quality Control (TQC) cycle activities as applied to programming.

4. Testing

Many people believe in the effectiveness of tests too much. Few books refer to the **unreliability of tests**. Actually a test may not be so reliable. Figure 4.1 shows some data taken from old, assembler age to early compiler age. [3] The vertical axis shows residual program error rate (in Logarithmic scale) and the horizontal axis shows the accumulated number of test intensity. Residual program error rate means program errors found in later tests plus those found after the release. Accumulated test is the accumulated number of tests per source code size. A plot shows the quality resulting from the tests hitherto done. In the figure, dotted lines are environmental tests where the number of test was not counted. (Nominally one test but actually it may include many tests.)

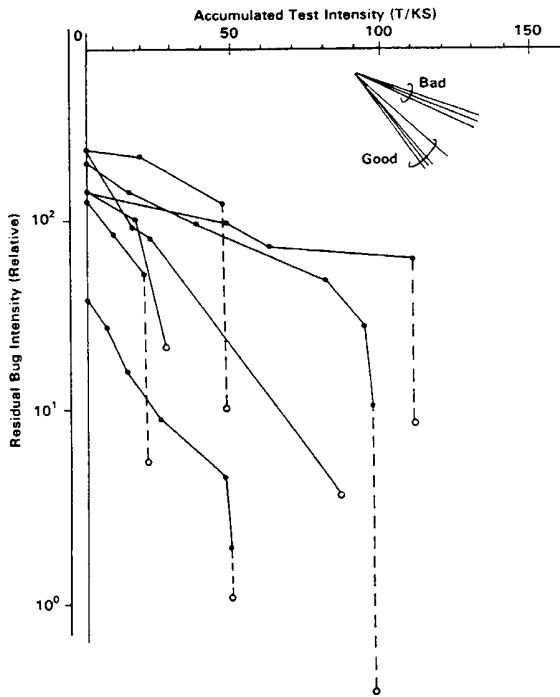


Figure 4.1 Decrease of residual error rate

From these plots, the following relations have been obtained;

*Circles show field ship-out quality.

*As the error rate decreases in negative exponentially against the test intensities, plots become linear.

*The gradient of plots shows the effectiveness of tests.

*The intersection of the plots and the vertical axis is a residual program error rate after desk checking before tests.

*From the preceding, the following relation is obtained:
 $S = R \exp(-ET)$

where, S :ship-out residual error rate; R :residual program error rate before tests; E :effectiveness of tests; and T :total test intensity.

In order to achieve an excellent quality or low S, R must be low and T must be high. A development group shows considerably stable R, E and T thus S, otherwise their processes are improved.

A test is modeled as shown in Figure 4.2. [6] As programs suffer from human errors, similarly all boxes in the figure also suffer from human errors. In the essence, a test is a comparison of a design result with the result of another design. Thus, a test is not so reliable. The situation becomes clear when both each error rate (and manhours used) in each elemental operations are quantitatively measured. These lead to quality therefore E improvement (rationalization) of tests.

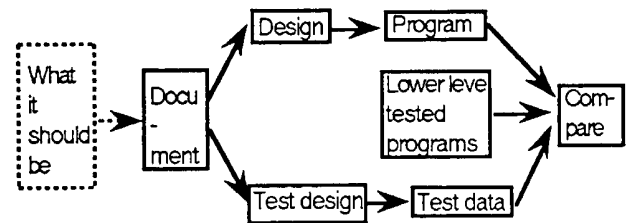


Figure 4.2 Structure of tests

The effectiveness of tests needs further study. Curves in Figure 4.1 are grouped into two categories. The steeper gradient group showed 30 times larger decrease of the residual error rate than the slower gradient group. A study of this low effectiveness group revealed that they showed a high error or cancellation rate of error corrections, and that the high cancellation rate was caused mainly by insufficient documents. The cancellation rate of preceding changes (to correct errors found in foregoing tests) is a good measure showing reliability of tests. Cause analysis on them show defects in development processes and leads to improvements for increasing effectiveness of tests.

Following are suggestions for decreasing error rates by tests:

*Take quantitative data (as described) on past developments and evaluate test process. There are no explicit common definitions of T. Use own way of counting them. It will be found Rs, Ts, Ss and Es for various developments are in respective range.

*Set new S for a target. Determine corresponding R, T and E.
 *Make R (residual error rate before tests) smaller as described before.

*Make T (total test intensity) larger. The test data must be highly reliable. The accumulated and updated proven test data is highly valuable. Automatic regression test is very useful. (Improve the way how to count tests.)

*Take various means to make E larger.

**Improve tests in two ways, the first is why errors were not checked-out before delivery based on quantitative studies on field detected errors and cancellation of changes. The second is how E is improved as described before. The most influential is documentation. Its discrepancies from "what it should be" invites very poor effectiveness. (Refer to Figure 4.2.) Documents must be detailed, clear and accurate.

**In bottom-up integration testing, lower level programs already tested and used for testing might contain residual errors. Care must be paid to detect errors not only caused at the level but also to detect errors caused by such lower level programs.

5. Actual development

Main factor for reducing errors have been discussed in preceding sections. In this section, some factors common to design and test are discussed.

5.1 Programming language

A software comes into an existence with some implementation language. Figure 5.1 shows our study on built-in errors classified according to the process of built-in. [10] Around half is built-in during coding and detailed design regardless of assembler language programs or compiler language programs. As higher level language programs need less number of coding lines for a function, the absolute number of errors of compiler language programs are smaller than that of assembler language programs. **The merits of high level language lie in decreasing expressions of designers thinking, not in complicating and enriching various functions.** As half of the errors arise during thinking in natural language, and another half arise during converting to programming language, **the future approach is to eliminate programming language.** Fourth generation language is toward this direction. For well standardized jobs, it is possible to use a natural language for programming resulting in a substantial decrease in programming errors.

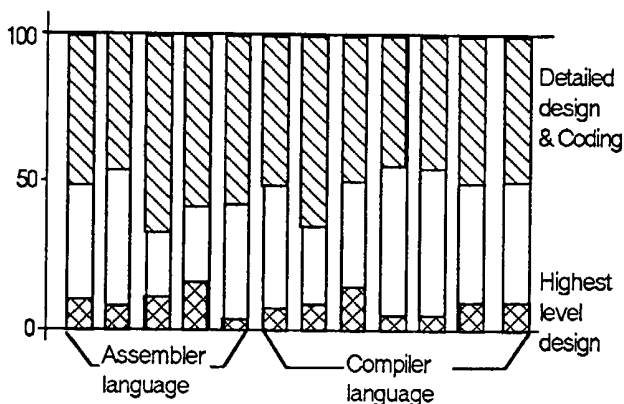


Figure 5.1 Build-in errors during design

5.2 People and organization

The present software industry is still a handcraft industry, as most of the work processes are done by humans. The workers' ability deeply affects quality. Ability not only in programming but also in the application is necessary. It is always important to **employ ability people**. As workers are with various past experience, most developers **standardize their work procedures by systems, standards, rules and practices**, which act as quality control of people. As ten years are necessary for a man to become an expert, ten years are needed for a team to become a very excellent team. In order to attract people to stay and grow in an organization, **an internal promotion career system based on long term employment policy must be taken**. Good management and a well organized career system are indispensable.

6. Conclusion

Thus far, various natures and relationships on programming errors from the view point of human errors have been explained, and based on them various suggestions have been described. Most of them may have been written in other papers or books. Various natures explained in the first half of this paper lead to ways for reducing programming errors. Quantitative relationships described lead to an error level diagram as shown in Figure 6.1 enabling an overview of quality for a development.

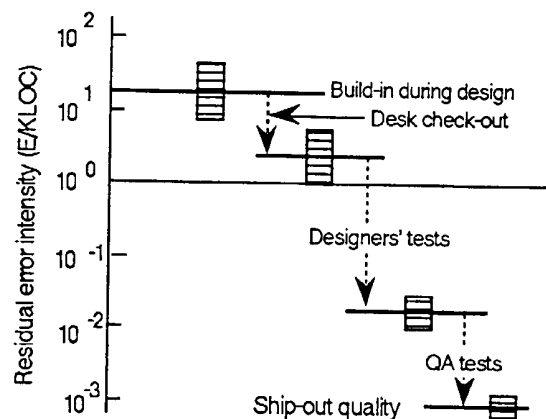


Figure 6.1 Error level diagram

*Programming errors are caused by the process in concern. The major part of processes are performed by human, errors are results of **inevitable human errors**.

*They may be studied by separating "build-in" and "check-out". Thus separating, details may be analyzed quantitatively by cause and effect.

*As errors are human errors, their nature follows "learning effect" as other human factors.

*The essential factor is the improvement of process by **reinforcements repeating experiences with feedback**.

*In order to improve quality, it is necessary to improve processes. The countermeasures are obtained from cause and effect relationship. The detailed procedures are given in TQC textbooks.

*Test is a comparison of a design with the results of another design, where all factors are prone to human errors.

*Quality after tests follows a relationship between quality prior to test, test intensity and effectiveness of tests.

7. Acknowledgements

The authors wish to express their deep gratitude to those who participated in the studies. This paper is an application of their studies and results of TQC practitioners in software fields made for many years.

References

1. Koono, Z., Yamato, Y. and Soga, M., "Structural Way of Thinking as Applied to High Quality Design", IEEE International Conference on Communications, 1988.
2. Koono, Z. and Soga, M., "Structural Way of Thinking as Applied to Quality Assurance Management", IEEE Journal on Selected Areas in Communications, Vol. 8, No. 2, pp. 291 - 300, Feb. 1990.
3. Koono, Z., Ashihara, K., and Soga, M., "Structural Way of Thinking as Applied to Development", IEEE/IEICE Global Telecommunications Conference 1987.
4. Humphrey, W., Kitson, D. and Kasse, T., "The State of Software Engineering Practice: A Preliminary Report", International Conference on Software Engineering 1989.
5. Koono, Z., Igawa, K. and Soga M., "Structural Way of Thinking as Applied to Improvement Process", IEEE Global Telecommunications Conference 1988.
6. Koono, Z., "Build-in and Check-out of Software Errors", The Third International Workshop on Software Quality Improvement 1991.
7. Kolers, P. A., "Reading a year later", Journal of Experimental Psychology: Human Learning and Memory, 1976, 2, 554-565.
8. 渡辺、緒方, "ソフトウェアの品質および生産性予測法の一事例について", 第2回ソフトウェア生産における品質管理シンポジウム (in Japanese)
Watanabe, K. and Ogata, H., "A case Report of Predicting Software Quality and Productivity", The Second Quality Control Symposium in Software Production.
9. Koono, Z., Tsuji, H. and Soga, M., "Structural Way of Thinking as Applied to Productivity", IEEE International Conference on Communications 1990.
10. 河野, "開発(設計)作業と誤り発生の構造", 第9回ソフトウェア信頼性シンポジウム, 1989. (in Japanese)
Koono, Z., "Structures of Development (Design) and Errors", The 9th Software Reliability Symposium, 1989.
11. Koono, Z. and Harada, E., "Structural Way of Thinking as Applied to Good Design; Part 2, Increasing Customer's Satisfaction", IEEE International Conference on Communications 1992.
12. 林喜男, "人間信頼性工学", 海文堂, 1984. (in Japanese)
Hayashi, Y., "Human Reliability Engineering", Kaibundo, 1984.