Strengthening Software Quality Assurance

Mutsuhiko Asada

Pong Mang Yan

Increasing time-to-market pressures in recent years have resulted in a deterioration of the quality of software entering the system test phase. At HP's Kobe Instrument Division, the software quality assurance process was reengineered to ensure that released software is as defect-free as possible.

The Hewlett-Packard Kobe Instrument Division (KID) develops measurement instruments. Our main products are LCR meters and network, spectrum, and impedance analyzers. Most of our software is built into these instruments as firmware. Our usual development language is C. **Figure 1** shows our typical development process.

Given adequate development time, we are able to include sufficient software quality assurance activities (such as unit test, system test, and so on) to provide high-quality software to the marketplace. However, several years ago, time-to-market pressure began to increase and is now very strong. There is no longer enough development time for our conventional process. In this article, we describe our perceived problems, analyze the causes, describe countermeasures that we have adopted, and present the results of our changes.

Mutsuhiko Asada

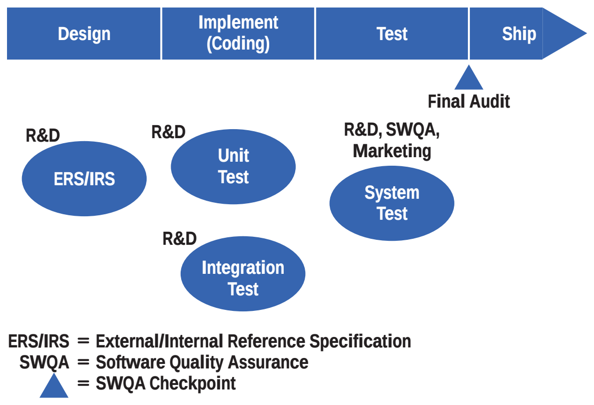
Mutsuhiko Asada is a software quality assurance engineer at HP's Kobe Instrument Division. He received a Master's degree in nuclear engineering from Tohoku University in 1986 and joined HP the same year. Born in Japan's Miyagi prefecture, he is married, has two children, and enjoys mountain climbing and photography.

Pong Mang Yan

Bryan Pong is an R&D engineer with HP's Kobe Instrument Division, working mainly on firmware. He received Master's degrees in electronics and computer engineering from Yokohama National University in 1996. He was born in Hong Kong and likes travel and swimming.

Figure 1

Hewlett-Packard Kobe Instrument Division software development process before improvement.



Existing Development Process

The software development process that we have had in place since 1986 includes the following elements:

- Improvement in the design phase. We use structured design methods such as modular decomposition, we use defined coding conventions, and we perform design reviews for each software module.
- Product series strategy. The concept of the product series is shown in **Figure 2**. First, we develop a platform product that consists of newly developed digital hardware and software. We prudently design the platform to facilitate efficient development of the next and succeeding products. We then develop extension products that reuse the digital hardware and software of the platform product. Increasing the reuse rate of the software in this way contributes to high software quality.
- Monitoring the defect curve. The defect curve is a plot of the cumulative number of defects versus testing time (**Figure 3**). We monitor this curve from the beginning of system test and make the decision to exit from the system test phase when the curve shows sufficient convergence.

As a result of the above activities, our products' defect density (the number of defects within one year after shipment per thousand noncomment source statements) had been decreasing. In one product, less than five defects were discovered in customer use.

Perceived Problems

Strong time-to-market pressure, mainly from consumers and competitors, has made our development period and the interval between projects shorter. As a result, we have recognized two significant problems in our products and process: a deterioration of software quality and an increase in maintenance and enhancement costs.

Figure 2

The product series concept increases the software reuse rate, thereby increasing software quality.

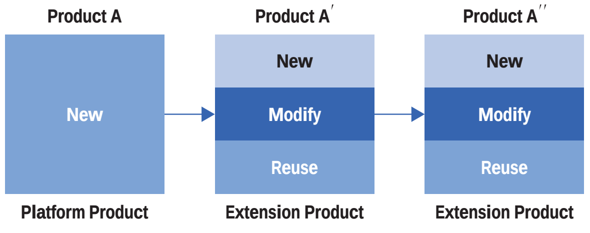
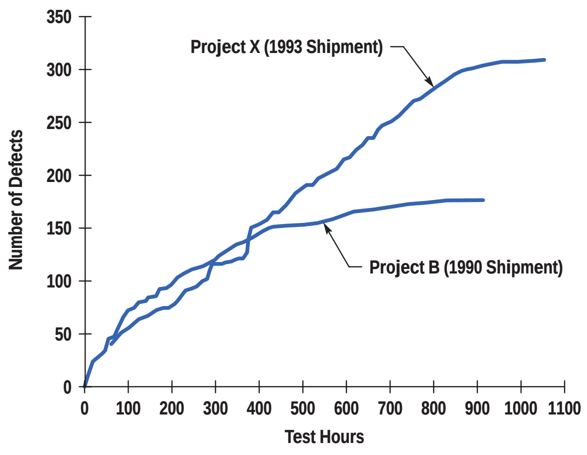


Figure 3

Typical defect curves.

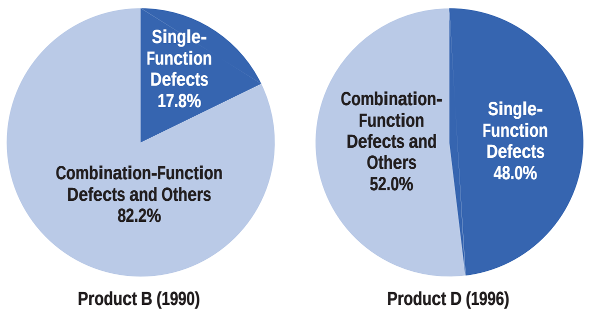


Deterioration of Software Quality. In recent years (1995 to 1997), software quality has apparently been deteriorating before the system test phase. In our analysis, this phenomenon is caused by a decrease in the coverage of unit and integration testing. In previous years, R&D engineers independently executed unit and integration testing of the functions that they implemented before the system test phase. At present, those tests are not executed sufficiently because of the shortness of the implementation phase under high time-to-market pressure. Because of the decrease in test coverage, many single-function defects (defects within the range of a function, as opposed to combination-function defects) remain in the software at the start of system test (**Figure 4**). Also, our system test periods are no longer as long. We nearly exhaust our testing time to detect single-function defects in shallow software areas, and we often don't reach the combination-function defects deep within the software. This makes it less likely that we will get convergence of the defect curve in the limited system test phase (**Figure 5**).

Increase of Maintenance and Enhancement Costs. For our measurement instruments, we need to enhance the functionality continuously to satisfy customers' requirements even after shipment. In recent products, the enhancement and maintenance cost is increasing (**Figure 6**). This cost consists of work for the addition of

Figure 4

Change in the proportion of single-function defects found in the system test phase.



new functions, the testing of new modified functions, and so on. In our analysis, this phenomenon occurs for the following reasons. First, we often begin to implement functions when the detailed specifications are still vague and the relationships of functions are still not clear. Second, specifications can change to satisfy customer needs even in the implementation phase. Thus, we may have to implement functions that are only slightly different from already existing functions, thereby increasing the number of functions and pushing the cost up. **Figure 7** shows that the number of functions increases from one product to another even though the two products are almost the same.

Often the internal software structure is not suitable for a particular enhancement. This can result from vague function definition in the design phase, which can make the software structure inconsistent and not strictly defined. In the case of our combination network and spectrum analyzers, we didn't always examine all the relationships among analyzer modes and the measurement and analyzer functions (e.g., different display formats for network and spectrum measurement modes).

Figure 5

Defect curves for post-1995 products.

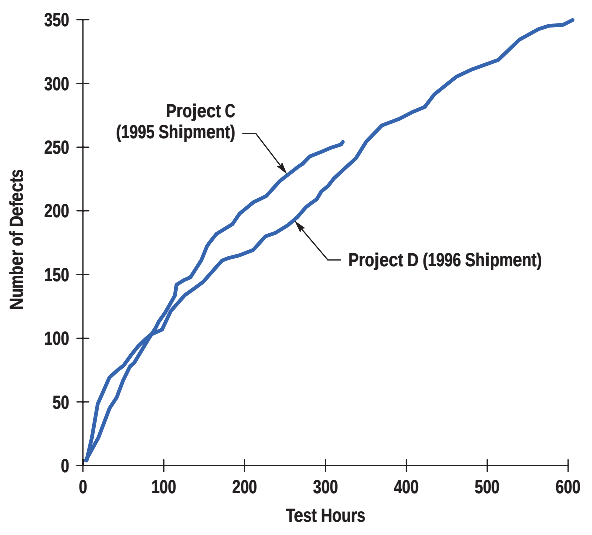


Figure 6

Increase in the cost per function of enhancement and maintenance. The first enhancements for Product B occurred in 1991.

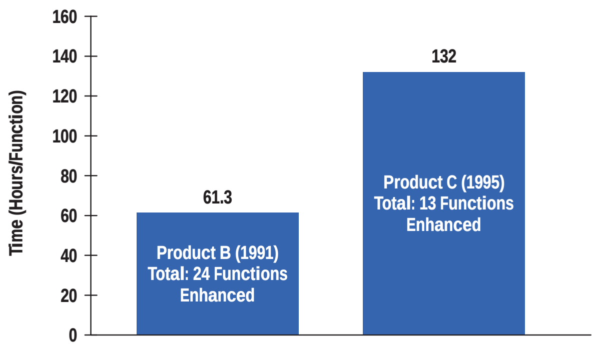
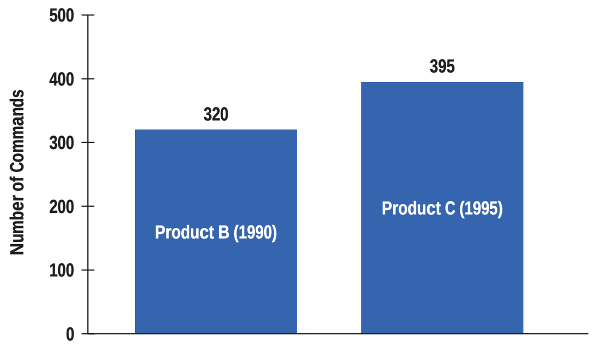


Figure 7

Increase in the number of commands in two similar analyzers as a result of changing customer needs.



Naturally, the enhancement process intensely disturbs software internal structures, which forces us to go through the same processes repeatedly and detect and fix many additional side-effect defects.

Countermeasures[1,2]

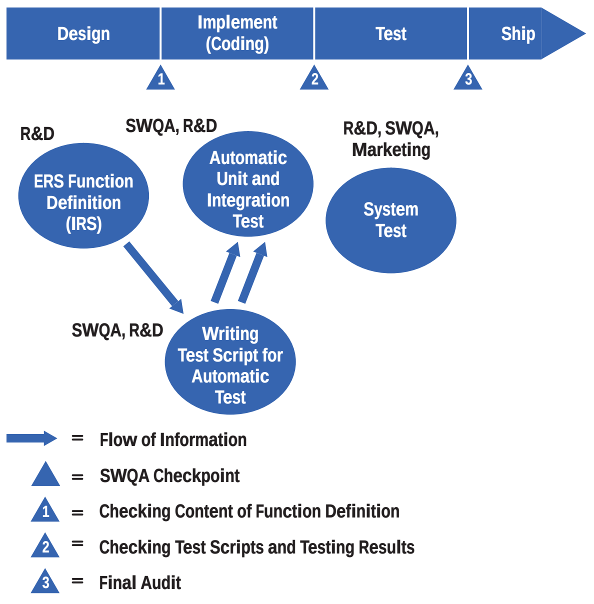
If we had enough development time, our problems would be solved. However, long development periods are no longer possible in our competitive marketplace. Therefore, we have improved the development process upstream to handle these problems. We have set up two new checkpoints in the development process schedule to make sure that improvement is steady (**Figure 8**). In this section we describe the improvements.

We plan to apply these improvement activities in actual projects over a three-year span. The software quality assurance department (SWQA) will appropriately revise this plan and improve it based on experience with actual projects.

Design Phase—Improvement of Function Definition. We have improved function definition to ensure sufficient investigation of functions and sufficient testing to remove single-function defects early in the development phase.

Figure 8

Improved software development process.


Design
Implement (Coding)
Test
Ship
1
2
3
R&D
ERS Function Definition (IRS)
SWQA, R&D
Automatic Unit and Integration Test
R&D, SWQA, Marketing
System Test
SWQA, R&D
Writing Test Script for Automatic Test
= Flow of Information
= SWQA Checkpoint
1 = Checking Content of Function Definition
2 = Checking Test Scripts and Testing Results
3 = Final Audit

We concisely describe each function's effects, range of parameters, minimum argument resolution, related functions, and so on in the function definition (**Figure 9**). Using this function definition, we can prevent duplicate or similar functions and design the relationships of the measurement modes and functions precisely. In addition, we can clearly define functions corresponding to the product specifications and clearly check the subordinate functions, so that we can design a simple and consistent internal software structure. We can also easily write the test scripts for the automatic tests, since all of the necessary information is in the function definitions.

SWQA, not R&D, has ownership of the template for function definition. SWQA manages and standardizes this template to prevent quality deterioration and ensure that improvements that have good effects are carried on to future projects.

Checkpoint at the End of the Design Phase. The first new checkpoint in the development process is at the end of the design phase. SWQA confirms that all necessary information is contained in the function definitions. SWQA approves the function definitions before the project goes on to the implementation phase.

Implementation Phase—Automatic Test Execution. In this phase, SWQA mainly writes test scripts based on the function definitions for automatic tests to detect single-function defects. We use equivalence partitioning and boundary value analysis to design test scripts. As for combination-function defects, since the number of combinations is almost infinite, we write test scripts based only on the content of the function definitions. When we implement the functions, we immediately execute the automatic tests by using the scripts corresponding to these functions. Thus, we confirm the quality of the software as soon as possible. For functions already tested, we re-execute the automatic tests periodically and check for side effects caused by new function implementations. As a result of these improvements, we obtain software with no single-function defects before the system test phase, thereby keeping the software quality high in spite of the short development period. The test scripts are also used in regression testing after shipment to confirm the quality of modified software in the enhancement process. In this way, we can reduce maintenance and enhancement costs.

Figure 9

An example of the improvement in function definition. (a) Before improvement. (b) After improvement.

```
[SENSe Subsystem]

SENSe
        :FREQuency
            :CENTer            <numeric>|DMARker|MARKer                  (Parameter changed
                STEP
                [:INCRement]   <numeric>|DMARker|MARKer                  (Parameter changed)
                  :AUTO        ON|OFF
            :MODE              FIXed|LIST|SWEep
            :SPAN              <numeric>|DMARker|MZAPerture              (Parameter changed)
                :FULL                                       [no query]
            :STARt             <numeric>|MARKer                          (Parameter changed)
            :STOP              <numeric>|MARKer                          (Parameter changed)
```

(a)

Module	Command	Range	Initial Value	Attribution						Description
Swp	POIN <val>	2 to 801 (int type)	201 (NA, ZA), 801 (SA)	S		2C	SL	4		In SWPT ≠ LIST, set MEAS POINT of sweep. (In SA, it's available when SPAN = 0.)
Swp	SPAN <val> (Hz)	0 to 510 M(Hz)	500 MHz	S	S	2C	SL	4	I	In SWPT ≠ LIST, set SPAN value of sweep parameter.
Swp	STAR <val> (Hz)	0 to Max Val (Hz)	0 Hz	S	S	2C	SL	4	I	In SWPT ≠ LIST, set START value of sweep parameter. Min and max value of START, STOP, CENTER depend of RBW.
Swp	STOP <val> (Hz)	Min Val to 510 M(Hz)	500 MHz	S	S	2C	SL	4	I	In SWPT ≠ LIST, set STOP value of sweep parameter. Min and max value of START, STOP, CENTER depend on RBW.
Swp	SPAN <val> (Hz/dBm)	0 to 510 M(Hz), 0 to 20 (dBm)	500 MHz, 20 dBm	S	NZ	2C	SL	4	I	In SWPT ≠ LIST, set SPAN value of sweep parameter.
Swp	STAR <val> (Hz/dBm)	0 to 510 M(Hz), – 50 to 15 (dBm)	0 Hz, –50 dBm	S	NZ	2C	SL	4	I	In SWPT ≠ LIST, set START value of sweep parameter.
Swp	STOP <val> (Hz/dBm)	0 to 510 M(Hz), – 50 to 15 dBm	500 MHz, 30 dBm	S	NZ	2C	SL	4	I	In SWPT ≠ LIST, set STOP value of sweep parameter.
Swp	SWET <val> (s)	0 (Min Meas Time) to 99:59:59 s	Min Meas Time	S		2C	SL	4		Turn sweep time auto setting off and set arbitrary value to sweep time. (In SA, query only.)
Swp	SWETAUTO <bool>	Off (0)/On (1)	On	S	NZ	2C	SL	4	D	Select Auto and Manual of Sweep Time (In SA, Auto only.)
Swp	SWPT <enum>	LINF/LOGF/LIST/POWE	LINF	S	NZ	2C	SL	6	H	Select Sweep Type.
Swp	SWPT <enum>	LINF/LIST	LINF	S	S	2C	SL	6	H	Select Sweep Type.

(b)

Checkpoint at the End of the Implementation Phase. At the second new checkpoint in the development process, SWQA confirms that the test scripts reflect all the content of the function definitions, and that there are no significant problems in the test results. The project cannot go on to the system test phase without this confirmation.

System Test Phase—Redefinition of System Testing. In an ideal testing process, we can finish system testing when we have executed all of the test items in the test cases we have written. However, if many single-function defects are left in the software at the start of system test, we will detect single-function and combination-function defects simultaneously, and the end of testing will become unclear. Therefore we use statistical methods, such as convergence of the defect curve, to decide when to end the system test phase.

In our improved process, we can start the system test phase with high-quality code that includes only a few single-function defects. Thus, we can redefine the testing method to get more efficiency in detecting the remaining defects. We divide the system test items into two test groups. The first group uses black box testing. We write these test cases based on the instrument characteristics as a system and on common failures that have already been detected in the preceding series products. The second group is measurement application testing, which is known as white box testing. The R&D designers, who clearly know the measurement sequence, test the measurement applications according to each instrument's specifications. We try to decide the end of system test based on the completion of test items in the test cases written by R&D and SWQA. We try not to depend on statistical methods.

Checkpoint at the End of the System Test Phase. We use this checkpoint as in the previous process, as an audit point to exit the system test phase. SWQA confirms the execution of all test items and results.

A Feasibility Study of Automatic Test

Before implementing the improved development process described above, we wanted to understand what kind of function is most likely to cause defects and which parts we can't test automatically. Therefore, we analyzed and summarized the defect reports from a previous product series (five products). We found that the front-panel keys, the HP-IB remote control functions, and the Instrument BASIC language are most likely to cause defects. We also observed that the front-panel keys and the display are difficult to test automatically. Based on this study, we knew which parts of the functions needed to be written clearly on the function definitions, and we edited the test items and checklist to make the system test more efficient.

Application of the Improvement Process

Project Y. Product Y is an extension and revision of Product X, a combination network, spectrum, and impedance analyzer. The main purpose of Project Y was to change the CRT display to a TFT (thin-film transistor) display and the HP-IB printer driver to a parallel printer driver. Most of the functions of the analyzer were not changed.

Since Product Y is a revision product, we didn't have to write new function definitions for the HP-IB commands. Instead, we used the function reference manual, which has the closest information to a function definition. The main purpose of the test script was to confirm that each command worked without fail. We also tested some combination cases (e.g., testing each command with different channels). The test script required seven weeks to write. The total number of lines is 20,141.

For the automatic tests, we analyzed the defect reports from five similar products and selected the ones related to the functions that are also available in Product Y (391 defect reports in the system phase). Then we identified the ones that could be tested automatically. The result was 140 reports, which is about 40% of the total. The whole process took three weeks to finish and the test script contains 1972 lines. The rest of the defect reports were checked manually after the end of system test. It took about seven hours to finish this check.

Both of the above test scripts were written for an in-house testing tool developed by the HP Santa Clara Division.[3] An external controller (workstation) transfers the command to the instrument in ASCII form, receives the response, and decides if the test result passes or fails.

Instrument BASIC (IBASIC), the internal instrument control language, has many different functions. It comes with a suite of 295 test programs, which we executed automatically using a workstation. The workstation downloaded each test program to the instrument, ran the program, and saved the result. When all the programs finished running, we checked if the result was pass or fail.

For all of the automatic testing, we used the UNIX® make command to manage the test scripts. The make command let each test program execute sequentially.

Using the test scripts, we needed only half a day to test all of the HP-IB commands and one day to test the IBASIC. Since Product Y is a revision product, we also used the test scripts to test its predecessor, Product X, to confirm that Product Y is compatible with Product X. The test items in the Product X checklist were easily modified to test Product Y.

Project Z. Product Z belongs to the same product series as Product Y (a combination network, spectrum, and impedance analyzer). The reuse rate of source code is 77% of Product Y.

One R&D engineer took one month to finish the first draft of the function definitions. To test the individual HP-IB commands, since the necessary function definition information existed, we easily modified the test script for Product Y to test Product Z. We employed a third-party engineer to write the test scripts. This took five weeks.

Since Product Z is in the same series as Product Y, we are reusing the test scripts for Product Y and adding the new test scripts corresponding to the new defects that were detected in Product Y to test Product Z.

The IBASIC is the same as Product Y's, so we use the same test program for Product Z. The automatic test environment is also the same as for Product Y.

Since Product Z is still under development, we don't have the final results yet. We use the test scripts to confirm the individual HP-IB commands periodically. This ensures that the quality of the instrument's software doesn't degrade as new functions are added. At this writing, we haven't started system test, but we plan to reuse the same product series checklist to test Product Z.

Results

Project Y. In this project, we found 22 mistakes in the manual, 66 defects in Product X while preparing the test scripts, and 53 defects in Product Y during system test. The following table lists the total time spent on testing and the numbers of defects that were detected in Product X in Project X and Project Y.

Table I
Defects found in Product X

	Project X	Project Y
Testing Time (hours)	1049	200
Number of Defects	309	88

According to this data, using the test scripts based on the function reference manual, we detected 88 defects in Product X during Project Y, even though we had already invested more than 1000 test hours in Project X and the defect curve had already converged (**Figure 3**). We conclude that testing the software with a test script increases the ability to detect defects. Also we see that a function definition is indispensable for writing a good test script.

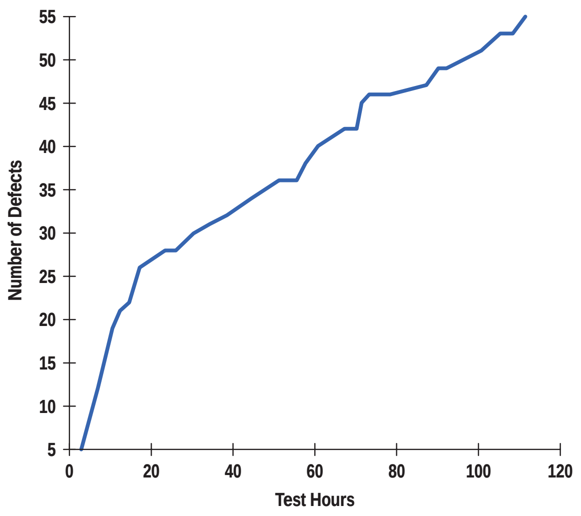
Since the automatic test is executed periodically during the implementation phase, we can assume that no single-function defects remained in Product Y's firmware before system test. Since Product Y is a revision product, there were only a few software modifications, and we could assume that the test items for the system testing covered all the modified cases. Therefore, we could make a decision to stop the system test when all the test items were completed, even though the defect curve had not converged (**Figure 10**). However, for a platform product or an extension product that has many software modifications and much new code, the test items of the system test are probably not complete enough to make this decision, and we will still have to use the convergence of the defect curve to decide the end of the system test. Nevertheless, it will always be our goal to make the test items of the system test complete enough that we can make decisions in the future as we did in Project Y.

The test script is being used for regression testing during enhancement of Product Y to prevent the side effects caused by software modifications.

In **Figure 11**, we compare the test time and the average defect detection time for these two projects. Because Product Y is an extension of Product X, the results are not exactly comparable, but using the test script appears to be better because it didn't take as much time to detect the average defect.

Figure 10

Defect curve for Project Y.



We needed time to write the test scripts, but the system test phase became shorter, so the total development time was shorter for Project Y. The enhancement cost will be lower because we can reuse the same test script for regression testing.

Project Z. We expect that the quality of Product Z will be high before system test because we test Product Z periodically in the implementation phase and confirm the result before entering system test.

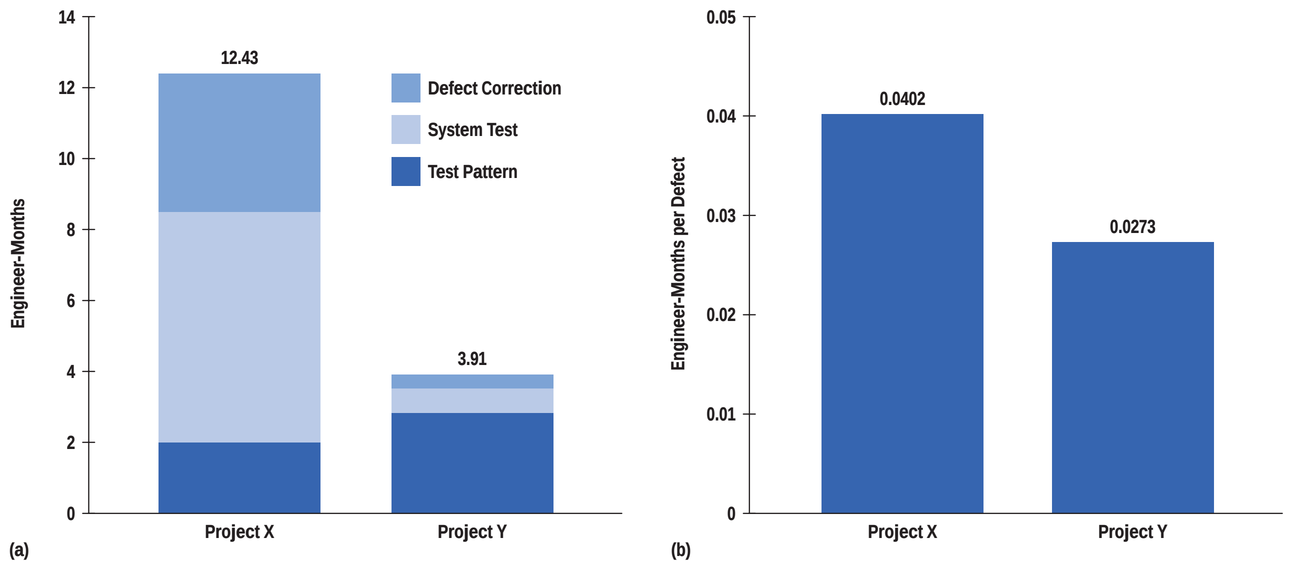
The additional work of the improvement process is to write formal function definitions and test scripts. Since this project is the first to require a formal function definition, it took the R&D engineer one month to finish the first draft. For the next project, we expect that the function definition can be mostly reused, so the time needed to write it will be shorter.

The test scripts are written during the implementation phase and do not affect the progress of the project. Therefore, we only need to wait about a month for writing the function definition before starting the implementation phase, and since the time needed for system test will be shorter, the whole development process will be faster.

Since we are reusing the test scripts of Product Y, the time for writing test scripts for Product Z is two weeks shorter than for Product Y. Thus, for a series product, we can reuse the test scripts to make the process faster. Also, making test scripts is not a complicated job, so a third-party engineer can do it properly.

Figure 11

Cost of software testing for Projects X and Y. (a) Engineer-months spent on software testing. (b) Engineer-months per defect.



Conclusion

We analyzed the software (firmware) development problems of the Hewlett-Packard Kobe Instrument Division and decided on an improvement process to solve these problems. This improvement process has been applied to two projects: Project Y and Project Z. The results show that we can expect the new process to keep the software quality high with a short development period. The main problems—deteriorating software quality and increasing enhancement cost—have been reduced.

This improvement process will be standardized and applied to other new projects. It will also make our software development process conform to the key process areas of CMM (Capability Maturity Model) level 2 and some part of level 3.[1,2]

Acknowledgments

We cannot overstate the importance of the work of Mitsuo Matsumoto on the automatic IBASIC tests. We would like to thank Akira Nukiyama and several of our colleagues for reviewing this paper and giving valuable comments.

References

1. M.C. Paulk, et al, *Capability Maturity Model for Software, Version 1.1*, Carnegie Mellon University, SEI-93-TR-024.

2. M.C. Paulk, et al, *Key Practices of the Capability Maturity Model, Version 1.1*, Carnegie Mellon University, SEI-93-TR-025.

3. B. Haines, *UNIX-Based HP-IB Test Tool (Ttool) Operation Manual*, 1991.

▶ Go to Next Article

▶ Go to Journal Home Page