Implementation of quality control in software development

by FRANK TSUI and LEW PRIVEN
IBM
Gaithersburg, Maryland

ABSTRACT

In recent years, many tools have been developed in the areas of programming techniques, design methodology, development processes, and test strategies. All of these are aimed at producing a quality product with a minimum cost. Together with emergence of these tools, there is a recognition of the problem of how to effectively implement them into a total systems development environment. One of the factors to a successful development is the ability to exercise proper control over the various development processes.

We will first establish the objectives for software quality control. Then, descriptions of the terminologies defining the various development processes will be presented in order to provide a common background. A basic philosophy of the entire development procedure and the notion of Continuous Integration will be presented. The introduction of quality control into this procedure will be discussed. Finally, some justification for such an approach will be shown.

INTRODUCTION

A large amount of effort has been spent on developing programming and design techniques. The various tools and methodologies are aimed at different steps in the development processes. Basically, these tools may be classified into the following categories in terms of their intended applications:

(i) Design, Code and Documentation
(ii) Test and Verification
(iii) Control of the Development Processes

In order to attain a good quality product, it is essential to understand how and where these tools apply in the development activities. The subject of this paper comes under the third category, Control of the Development Processes.

In this paper we will present a unifying paradigm of the entire development procedure. Within this global system, several development cycles for different functions may be in progress simultaneously. The development cycle is similar for each function. That is, the activities within each development cycle are fairly standard and flow in a fixed sequence much like an assembly line. For example, if we examine how a programmer does his job, normally the following activities take place:

(i) Design/Document
(ii) Code
(iii) Test
(iv) Integrate
(v) Performance Measure
(vi) Release

This is a "natural" process; that is, if left alone, most programmers would follow this sequence in implementing a program, except, possibly for the documentation part. Since programming lends itself to a natural flow, the process or the development cycle should keep this intact.

In lieu of the yet experimental and basically primitive automatic program correctness verifiers, quality control is obtained via the introduction of inspections and reviews into the development cycle. Then two additional steps called Error Prone Analysis and Post Functional Test Analysis will be introduced to monitor the development cycle and the entire development procedure.

GOAL OF QUALITY CONTROL

Today's environment demands that programming products be of "high" quality. From the users' point of view this means: functionally complete, easily usable and installable, and low error rates. Additionally, to the developer it means that the schedule must be met at the lowest cost. This is certainly a tall order based on the past performance of the programming community, but it can be achieved with the proper management and quality control techniques.

The main objective here is to ensure that only a minimal number of software problems exist. This may not be as obvious as it appears at first glance, for it is not sufficient that just the final product contains a small amount of problems. It has been the experience that correcting a problem in the final product will
require much more effort than correcting that same problem in the early phase of the development. Thus it is essential that errors be detected and corrected as early as possible and only a minimal amount of problems be allowed to slip from one phase of the development to the next.

The secondary objective, although it may be just as important as the main objective, is to increase the ease of usage and future maintenance of a product. As the cost of producing a completely new system increases, the property of being able to enhance and build upon the existing system becomes significant.

**TERMINOLOGIES AND DEFINITIONS**

In this section some basic notions of Design, Code and Test will be explicitly defined in order to facilitate the introduction of the concept of inspection or review.

The Design phase of the development cycle is composed of two levels: High-Level Design and Detailed Design. The High-Level Design contains the following information.

(A) An over-all functional specification describing what is to be performed.

(i) For each major function, the modules' names should be specified; and within each module, the description is carried to the level where each statement represents approximately 20 lines of executable source code.

(ii) All control blocks with major fields and their purposes identified.

(iii) For each function, all the macros that need to be designed should be listed and described to the depth where each statement represents approximately 20 lines of executable code.

(iv) For each module, all data files and tables should be presented along with a brief description of their purpose and method of access.

(B) The dependencies for each major function are presented by modules, identifying both internal and external cases.

(i) Internal dependencies specify those macros and control blocks which are shared by several modules.

(ii) External dependencies indicate all the modules, control blocks, and macros needed but were not included in the overall functional specifications described above.

(C) The linkages between functions (by modules) are specified.

(i) Parameter list is identified.

(ii) All system Generation/Initialization requirements must be described.

(iii) All locks and authorization into data files are described.

(D) For each function, a control flow (in some flow-chart form—preferably with a predetermined Structured flow chart) describing the logic should be given by modules.

The Detailed Design is a refinement of the High-Level Design. Thus the information described above for High-Level Design serves as the input to the Detailed Design phase. The information contained in the Detailed Design is used as the input to coding. It contains the following:

(A) A complete “Structured” logic flow (by modules) should be given.

(i) Each box in the flowchart should represent approximately 20 lines of executable source code.

(ii) Reference to control blocks and data files should be by field names.

(iii) Macro invocations should specify all required parameters (by values).

(iv) Each predicate in the flowchart should have a corresponding “purpose” description.

(B) Data structure should be clearly specified. Each data file and table should be described down to each field identifying the following:

(i) intended usage

(ii) attributes

(iii) access method

(iv) value range (if any)

The coding phase utilizes the Detailed Design as its input. Here the Detailed Design is converted to source code and machine compiled or assembled. Each module should be at least compilation or assembly error free. There should be a module prologue preceding the actual code, stating the following about that module.

(a) major functions

(b) control blocks referenced (if any)

(c) macros used (if any)

(d) Entry-Exit conditions (if non-standard)

(e) linkages (if any)

(f) I/O parameters (if any)

(g) list of logic revisions made since last version (if any)

(h) version number and date

(i) module owner

An important point here is that the code must follow the Detailed Design. Structured flowchart with its corresponding Structured Programming Language would make this process much easier than otherwise.

The Testing phase depends heavily upon the documents generated during High-Level and Detailed Design. Generally, there are three categories of testing:
The functional test refers to the verification of each function utilizing a series of test cases within the intended environment. This may be the most time consuming test, depending on how detailed the test cases are. The component test involves the verification of inter-functional activities. The system test is performed to insure that no part of the system is ill-affected (including performance) and that the system as a whole does not crash even under unusual, non-intended environment. An extensive description of testing processes may be found in Reference 6.

IMPLEMENTATION OF QUALITY CONTROL

Before we can meaningfully discuss Quality Control, there has to be an underlying development philosophy. In this section, we will first present a paradigm of the complete development process. Within that paradigm the development cycle with the built-in controls will be discussed.

In order to insure low error rates, some quality control techniques must be utilized. This implies a defined procedure that can be instrumented to provide a) the necessary points for measurement and b) feedback path to correct defects. As stated earlier, the "natural" sequence of activities of Design, Code, Test, Integrate, Performance Measure, and Release should be kept as part of the process. Together with these activities, some checkpoint must be obtained before an activity is considered complete and another begun. The question is whether there is any one activity that plays an overriding role in defining the interaction between the other activities. Design seems to be the logical choice, but integration will be shown to be the key in defining the sequence of design and development activity.

Similar to building a house, the sequence that programs are put together is very important. In the case of a house, first the foundation, then the outside walls, then the roof, etc. are "put together" in that sequence. The parts may be built independently and simultaneously. Once the "basic" framework of the house is completed, the other components such as kitchen cabinets, toilets, etc. may be added separately. The same concept holds true for programs; there is a main functional path that must be completed first. Then the other functions can be added until the entire product is complete. This technique works best when functions are added individually. The technique is called Continuous Integration; it provides two significant attributes to the programming, development process. This is conceptually similar to Iterative Enhancement. First, by adding functions individually to an already working base, error isolation can generally be narrowed down to the newly added function or interface areas. This is critical as the system becomes more complex. Second, the development of each discrete function entails the same development cycle and can be tracked through the process by a standard set of quality control techniques.

The concept of Continuous Integration can be easily demonstrated as follows:

Modules A, B, C and D are already developed, and they form the working base. Now, three new functions need to be added.

Figure 1 shows the relationship between additional, new functions and the impact to the modules. The crosses represent "affected." With Continuous Integration the three functions would be developed independently; although they can be developed simultaneously. As any one function is completed, it is integrated into the working base and a new base is formed.

Consider the alternative. Had we developed them by modules, the delay of any one module would affect the completion of at least two functions. If module B is delayed, nothing will be completed. This can be disastrous. The testing and verification of functions are also difficult with the Modular approach. All affected modules must be completed at the same time before any one function can be tested. Suppose modules A and C were completed first. Functional verification can not be performed until the most "affected" module B is completed with all its new enhancement. If the modules were developed at physically different sites, then the problems are even more complex.

Continuous Integration utilizes a "blueprint" for putting the system together called the system build plan. This plan is used to determine in what sequence code must be developed and, thereby, the design sequence. Figure 2 represents a unifying paradigm within which Continuous Integration is applied.

To have the processes started, assuming that the product has been defined via the commitment process, a system build plan must be produced. As shown in Figure 3, the production of the build plan is an iterative process, requiring trade offs in resource, function, and schedule. Once this has been agreed to for the
initial product definition, the processes shown in Figure 2 can be used to control the program production and introduction of changes.

Another area that is significant to the total development and quality control is documentation. A project "workbook" that contains at least the design documentation for each function must be maintained in an up-to-date manner. This documentation is used as:

1. Source for user publication
2. Source for test case development
3. Control Information for Continuous Integration
4. Source for quality control activities.

It is important to remember that Figure 2 is not a time sequence chart for the whole system; however, it is for any one function. At any given point in time, some activity will be taking place in each area. The processes outlined have many other facets that will not be discussed in this paper, for they are not relevant to the main topic of quality control. Refer to Reference 14 for further details.

In summary, the main features of the paradigm that are significant to quality control are:

(a) Process definition that provides for discrete turnover points between activities (e.g., design to code).
(b) Control over the input to the system via commitment control.
(c) A blueprint for putting the product together (e.g., system build plan). The system build plan is also the mechanism used to review the status of each function as well as the integrated system. Using this mechanism, inputs from the quality control activities can be used to modify the blueprint and/or reallocate resources to assist areas in need.

(d) Work units (functions to be developed) that move through the processes in trackable units.
(e) Documentation of the design which provides independent source material for quality control checking.

Now, we will show precisely how items (a), (d) and (e) mentioned above can be achieved by inserting quality control checkpoints into the development cycle of any major function. Recall that several functions may be developed simultaneously but they all follow a similar development cycle.

First a set of inspections or reviews will be included into the cycle in order to attain the main objective of quality control, which was mentioned earlier. That is, with the inspections in place, the number of errors passed from one phase to the next will be greatly curtailed. Strict Exit Criteria will be imposed at each step such that the ensuing step may not start until the previous step is completed. In addition to the built-in controls of inspections and exit criteria, there should be explicit control steps in the development cycle. Namely, two new steps (a) Error-Prone Analysis and

![Figure 2—Programming processes paradigm](image-url)
Post Functional Test Analysis are included. There will also be a Test Planning process in the development cycle. Finally, we have the complete picture of a Development Cycle in Figure 4. All the terminologies (e.g., High-Level Design) follow the definitions provided in the previous section.

Each inspection involves the following 4 steps:

(a) Preparation
(b) Inspection itself
(c) Rework
(d) Follow-up

In the preparation step, all the appropriate and required outputs for that phase (e.g., Detailed Design) must be distributed to the inspectors. The inspectors are then given enough time to digest the material before the actual inspection. Typically, the inspectors are composed of a moderator, the designer of the function, the coder of the function, a testing representative, and a publications or documentation representative. Refer to Reference 7 for details on inspections. The objective of an inspection is to identify problems as opposed to offering solutions and better alternatives.

The Error-Prone Analysis step follows the coding phase. The content of the inspection results from High Level Design, Detailed Design, and Code is analyzed. Those areas that required continuous corrections and changes are identified. The reports generated during this step serve two purposes. They will allow the developers to decide whether certain areas should be reworked before entering the Test phase. The decision might require a change to the build plan mentioned earlier. The reports can also be used as additional guideline to the Testing phase in that the error prone areas may require more extensive test effort.

The Post Functional Test Analysis step, as the name suggests, immediately follows the Functional Test. The recorded test results are analyzed. Besides identifying areas which are functionally weak, the data may be used to estimate the number of errors that will occur when the system goes into production. This estimate is measured against a pre-established error tolerance level. Thus the results from Post Functional Test Analysis may be used to determine the quality of the final product.

Due to constraints of available space, we will not discuss each exit criteria. Only the Exit Criteria 1 will be described along with recommended tools for the High-Level Design phase.

The output from the High-Level Design is a design specification in the form of flow-charts and HIPO diagrams. The recommended tool here is the structured programming concept in that the flowcharts abide by the rules of structured programming. These will be used for the inspection. The Exit Criteria 1 contains the following:

1. High-Level Design inspections results logged into an inspection file.
2. All necessary rework signed-off by managers and logged on the inspection file as complete.
3. The updated version of the High-Level Design Specification is logged into a project workbook, signed-off by managers, and distributed to (i) project developers, (ii) testers, (iii) documentation and publications groups and (iv) product assurance group (if any).
4. Each module is assigned an owner.

Note that the required output, High-Level Design Specification, serves, not only as input to the next phase of the development, but also as input for (a) early test planning and (b) documentation.

The Development Cycle (Figure 4) may be viewed as the static component of development environment as every function is produced by going through the same cycle. Its dynamics are shown in the Programming Process Paradigm (Figure 2). Thus the Development Cycle, into which various quality controls are built.
is the static part of a dynamic Programming Process Paradigm.

SOME JUSTIFICATIONS

In the previous section, we have presented an implementation of quality control in software development. The Inspections, Error-Prone Analysis, and Post Functional Test Analysis are all aimed at attaining the main objective of low error rate. The Exit Criteria which ensures proper control and certain amount of documentation are aimed at achieving the secondary objective of easiness of enhancement and usage.

These objectives of quality control seem to be "user" oriented rather than the developers. We believe that if properly implemented, the developers will benefit just as much. That is schedules will be met, and total development cost will go down. This can be demonstrated by comparing the cost of quality control (in people hours) against the cost of having to correct the errors (in people hours) afterwards.

Figure 5 shows the estimated person hours to inspect code and design. Recall that inspections are conducted with a designer, coder, tester and a representative from publications. One of these may be the moderator. So inspections require four people. Figure 5 shows the cost of quality control in developing different sized products in terms of people-hours based on 10 hr/k for High-Level Design, 15 hr/k for Detailed Design, and 13 hr/k for Code inspections. (k=1,000 lines of executable code)

It is conjectured that on the average a programmer makes 7 errors per 1,000 lines of source code. Assuming that the normal testing eliminates 50 percent of the errors and that with the Inspections and other quality controls the test efficiency goes up to 60 percent, then Figure 6 shows the number of errors left after release. The range of error-elimination efficiency of Inspections is chosen to be 30 percent, 40 percent, 50 percent and 60 percent.

It is estimated that each of these remaining errors costs approximately 70 people hours to fix. This includes identify the problem, make the correction, compile or assemble, test, and check the test. In some cases where redesigning becomes necessary the cost can be much higher. So far, we have assumed unique errors. For each of the unique remaining errors there may be several ill effects, causing one to spend more time in verifying that these are all due to the same unique error. We will, however, use the conservative 70 people hours. The question is whether the cost of correcting these remaining errors to the level that the quality control techniques yield is more than the cost of introducing these techniques into the development. Figure 7 shows the cost of correcting these errors to the levels of different Inspection efficiencies. For example, it costs 105 people hours to bring 3.5 errors down to 2 errors.

A comparison between Figures 5 and 7 shows that not until we can conduct quality control to the point of 60% error elimination before testing does the introduction of these techniques become economically worthwhile. In reality, a lower level of quality control proficiency would probably suffice, for machine time cost was never introduced into the comparison. Note that the quality control techniques here do not require any machine time. Furthermore, we assumed a fixed 70 people hours for error correction. This resulted in a linear relationship between the size of the product and the error correction cost. A more reasonable relationship would show that the cost of fixing an error would be higher the larger the product as in Figure 8.

For small modules, the error correction percentage of quality control would have to be high before it is economically worthwhile.
CONCLUSION

In this paper we have introduced a development philosophy called Continuous Integration and a general software development paradigm which can facilitate several Development Cycles. Within any one Development Cycle, a set of quality control techniques may be implemented. With the implementation of quality control, it is shown that the following can be achieved:

1. lower error rate
2. ease of maintenance
3. lower total cost

The economical comparison, although still not precise, does provide a guideline to the management for evaluating quality control techniques.

REFERENCES
