International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:3, No:10, 2009

Defect Prevention and Detection of DSP-software

Deng Shiwei

***Abstract*—**The users are now expecting higher level of DSP(Digital Signal Processing) software quality than ever before. Prevention and detection of defect are critical elements of software quality assurance. In this paper, principles and rules for prevention and detection of defect are suggested, which are not universal guidelines, but are useful for both novice and experienced DSP software developers.

***Keywords*—**defect detection, defect prevention, DSP-software, software development, software testing.

I. INTRODUCTION

As application area of DSP(Digital Signal Processing) is expanding fast, demand for DSP software grows rapidly, and higher percentage of total cost is expended on DSP-software development. While size and complexity of DSP software increase, we face serious challenge of building DSP software with high quality [1].

In order to achieve higher level of software quality, prevention and detection of defect must become the focus of attention [2]. Defect detection aims at finding faults in software by testing and then correcting them. On the other hand, defect prevention focuses resources on correcting flaws of development process, thereby preventing the defects from being creating in the first place as much as possible, so that less effort is needed to detect and fix them later.

To avoid common complications in prevention and detection of DSP-software defect, we should follow certain rules. The following principles and methodologies enable us to create reliable DSP-software and to establish an efficient development mechanism for a group of developers.

II. DEFECT PREVENTION IN DSP-SOFTWARE DEVELOPMENT

Prevention activities can be as simple as providing checklists, improving product document, and enhancing development tools. Defect prevention can improves DSP-software quality, provides continuous improvement of development process.

Defect prevention can result in significantly lowered field defect rates. A reduction of 50% in the defects that arise during development generally will result in a 50% reduction in the field defects as well. Moreover, the resources that were formerly spent correcting defects can be put toward developing additional function and reducing the overall development cycle time, both of which reflect higher software quality in the broader sense.

Defect prevention provides a continuous focus on process improvement. The development process is used here in the broad sense denoting all of the formal and informal stages and steps, methodologies, techniques, and tools that are used to develop software.

Defects occur because of flaws in the development process or difficulties in its execution. For example, defects can result from failures to thoroughly prepare or educate developers, failures to communicate changes, or failures to provide adequate time or proper tools for checking design closure. Preventive actions on the other hand can address each one of these shortcomings by improving or fixing the development process.

Defect prevention not only fine tunes an current development process and practices but also encourages identifying and implementing new processes, methods, and tools. For example, new design methods or tools might be introduced as the result of suggested action. Once the new process or method is introduces, further preventive actions can help refine and fine tune that development process.

The principles of defect prevention described here are in reference to a particular implementation within IBM Corporation known as the Defect Prevention Process [3]. The Defect Prevention Process provides a framework for achieving the objectives of preventing defects, and continuously improving processes. The Defect Prevention Process has proven effective in improving software quality at a reasonable cost for implementation.

There are four key elements in the defect prevention, as showed in Fig.1. In general, a successful implementation of the defect prevention process incorporates all these elements. However, a DSP-software development team may develop variations or adaptations of these key elements, depending on its particular needs or its development process. In general the activities of the Defect Prevention Process are repeated for each major development stage or step, for example high-level design, low-level design, coding, unit test, etc. If the development project is organized into teams of developers, the defect prevention activities are conducted at the team level.

A. Causal analysis of defects and problems

The regular analysis of defects and problems occurring in DSP-software development process is performed, and preventive actions are suggested. Causal analysis is done by the

Deng Shiwei is with the Beijing Institute of System Engineering, P.O. BOX 9702-19, Beijing 100101, China.

developers, that is, by the people who created the defects.

During each development stage, defects are detected through inspections, reviews, or testing. When a number of defects have been collected, a causal analysis meeting is held by the team. The team reviews the defects, determines their root causes and proposes actions to prevent similar defects in the future.

During the causal analysis meeting, the developers identify the causes of the defects that have occurred. Usually the developer who created the defect can best identify its cause but many times a discussion involving the entire team will help clarify contributing causes. The team tries to propose actions that will eliminate the causes and thus prevent recurrence of the defect. In addition, actions may be proposed that have no direct bearing on the specific causes or even on the defect in question, but which are good suggestions to be implemented for the team.

Suggested actions can address process improvements, tools improvements and enhancements, education offerings or improvements to existing education, improvements in communication procedures or project management practices, and changes to software itself. Suggested actions need to be described with specific details, not simply as vague suggestions for improvement, so that the action team can implement them readily.

B. An action team to implement preventive actions

The sufficient resources to implement the preventive action in a timely way should be provided.

Defect prevention must include a means of implementing the suggested actions that insures that they are implemented in a timely way. Timely action implementation requires that appropriate resources are allocated by management. The people who implement the actions should have the appropriate scope and authority within the team to effect changes to the team's processes and practices. And they should have the appropriate skills to be able to implement the actions. In addition the time that is devoted to action implementation should be protected from erosion by other development responsibilities the people may have.

The Defect Prevention Process calls for an action team to be established to implement the preventive actions. The action team consists of developers from the area who work part-time to implement actions.

Action team members are selected based on the skills needed. A software development team typically needs action team members to handle process changes, education offerings, and tool development and enhancement. The action team also needs representatives from the key technical areas in the development team, for example, design, development, test. Finally, a manager from the area is needed to handle suggestions for project management improvements and to assist in obtaining help for actions needing resources outside the action team.

C. Periodic, timely feedback to developers

The regular reviews of the details of the DSP-software development process are conducted with developers, and feedback on the process changes that have occurred from implemented actions is provided. This periodic feedback is usually done in a kickoff meeting at the beginning of each development stage or step.

Many actions result in information that is kept in on-line files that can be accessed by the developers. Such materials include, for example, development process documentation, product technical information, checklists, common error lists, development guidelines and conventions, educational materials, project management guidelines, and tools documentation. The materials are typically placed in on-line repositories with appropriate indexes and search capabilities for easy access. Such repositories preserve the area's technical process and software product knowledge.

Having on-line repositories, however, is not enough to ensure that the appropriate knowledge will be used by the developers during development. It has been found that periodic reviews of this information are needed to remind developers of the process details that are critical to their work in a particular stage. It is necessary to provide feedback in a timely way, at the appropriate points in the development process.

The form that this period feedback tacks is the stage kickoff meeting. Stage kickoffs are generally held by the team at the beginning of each development stage and are conducted by the team's technical leader. The team leader reviews the development process for that stage, focusing on areas that have been weak in the past, emphasizing new practices, and reviewing other enhancements to the team's process, methods, and tools.

It is important that stage kickoff meetings be integrated into the development process, at the beginning of each development stage, and that the developers be directly involved. It has been found that simply holding stage kickoffs can achieve significant reductions in defects for that stage

D. Tracking analysis of data from the prevention process

A data base of the preventive actions is provided for tracking, and data about the defect prevention process itself is provided for management control.

Data are collected from the Defect Prevention Process to provide both the tracking of action status by the action team and the measurement of the process as a whole for management. The forms of data collection and tracking include:

(1) A means of tracking actions as they are implemented, to ensure that the actions are implemented in a timely manner and are implemented correctly. The action team maintains a data base containing each action and uses reports from it to review and handle newly created actions, open actions being implemented, and recently closed actions. Among the data that are kept for each action is its priority, target data, estimated cost, and estimated effectiveness.

(2) A means of measuring the Defect Prevention Process to ensure that the required activities are being done and that the level of investment in prevention intended by management is being achieved. Management uses these data to maintain its focus on the process. The action data base can be used to produce monthly management reports on the level of

investment in the process, that is, the time spent in prevention-related meetings, for action implementation and on education on the process, and the work flow in the process, that is, the number of actions open at the beginning of the time period, actions created and closed during the period, and actions remaining at the end of the period. Management can use these data to monitor the team's adherence to the process.

(3) A means of validating the effectiveness of the improvements once they are implemented. Certain types of defects that occur frequently can be tracked to see whether they recur after preventive actions have been implemented. To accomplish this, the action team can monitor selected types of defects. If the defects recur, further preventive steps can be taken.

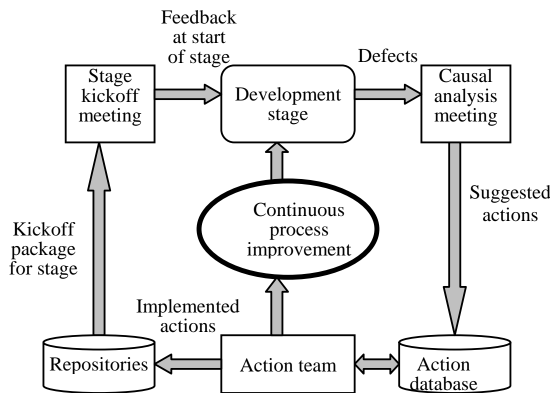

Fig. 1 Defect prevention process.

The key to a successful implementation of the defect prevention is the integration of prevention activities in the DSP-software development process. In particular, the causal analysis and feedback activities must become part of the basic practices of the development team, much as inspections have been integrated in many development teams. The prevention activities should not be a separate or isolated effort or one that conducted solely at the end of the development cycle. Rather, it should have a continuous focus throughout the DSP-software development cycle, involving all developers.

III. Strategic approach to DSP-software testing

From a procedural point of view, testing within the context of software engineering is actually a series of four steps that are implemented sequentially [4], as showed in Fig.2. Unit testing focuses on each module individually, assuring that it functions properly as a unit. Unit testing makes heavy use of white-box testing techniques, exercising specific paths in module's control structure to ensure complete coverage and maximum error detection. After modules are assembled to form the complete software package, integration testing addresses the verification of program construction. Black-box test case design techniques are the most prevalent during integration testing, although a limited amount of white-box testing may be used to ensure coverage of major control paths. After the software has been integrated, validation testing provides final assurance that software meets all functional and performance requirements. Black-box testing techniques are used exclusively during validation. Once validated, software must be combined with other system elements, e.g. hardware. System testing verifies that all elements mesh properly and that overall system function is achieved.

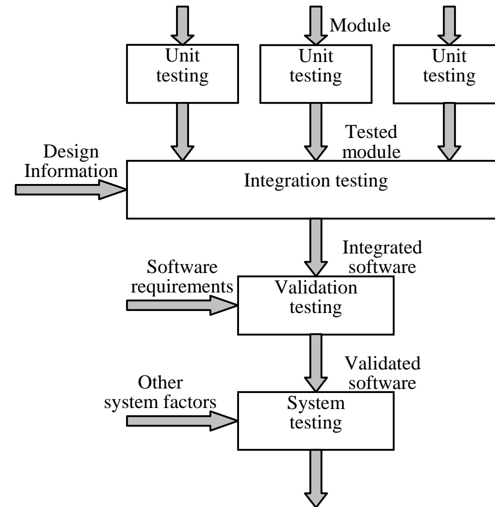

Fig. 2 Software testing steps.

For DSP-software development, the testing process includes following steps:

A. Unit testing

The core algorithms are developed and simulated using high-level tools and languages, such as C++, Matlab, MathCAD, LabView, and SystemView. These tools usually allow developers more conveniently to prove the algorithm's main idea and to roughly outline the smaller independent algorithms and submodules that you need to develop.

The C code for each small algorithm and submodule is develop, which is platform-independent, maintainable, and flexible code and able to be simulated on a PC before porting to a DSP.

Each small algorithm or submodule is tested independently on a PC. For each submodule, we can create a stand-alone test environment to test the submodule before integration with the whole system.

B. Integration testing

Integrating and simulating environment is developed to enable the integration of small algorithms and submodules. We should create such an environment for the whole system to simulate it on a PC. This environment usually provides a rich set of visualization tools that allow us to observe the behavior of the system as a whole and of each separate algorithm.

To enable the development and execution of different test cases with a set of different input test vectors, a test environment should also be created, which may be a part of the simulation environment. We should test this environment on its

own to ensure that it completely covers all possible algorithm states.

After small algorithms and submodules are integrated using integration and simulation environment, we should test these elements together as a whole system using the PC test environment before porting to a DSP microprocessor. The simulation should cover as many configurations and states of the system as possible. We should achieve all the required characteristics of the algorithm/system at this stage before porting to DSP hardware.

C. Validation Testing

Usually, converting the whole C source code into assembly code is unnecessary, which would make the code more difficult to be maintained and debugged, we convert only critical functions to assembly code of DSP, maintaining a C-like interface.

Using various input test vectors or test cases that cover all possible function states, we can create output test vectors for each function and the whole system on a PC and DSP simulation environment. Comparing these output test vectors bit by bit allows us to test the bit exactness of conversion.

Before testing bit exactness of the output vectors that assembly files generate, we must first test whether the C code gives a bit-exact result when we compile and run it under different compilers and platforms, such as 16- and 32-bit compilers and PC and DSP platforms.

D. System Testing

We should use the same C code that you used in the PC model (or, for converted functions, their bit-exact DSP assembly version) in the real-life system. The test configuration should be as close to real-life conditions as possible, and testing should include as many counterpart devices as possible, for example for interoperability. If any problems are found during real-time testing, we should return to the first step of this process to correct and the C code.

IV. Conclusion

When developing DSP applications, software developers can encounter a number of typical obstacles. A poorly organized development process can create many problems, some of which pop up immediately and some of which become apparent much later in the software life cycle. To avoid these common complications, we should follow certain rules and methodologies.

Using the principles and methodologies of defect prevention and detection, we can create reliable, high quality software. Following the framework of defect prevention also enables us to establish an efficient software development mechanism for a group of developers.

References

[1] N. Abkairov and A. Nazarov, “Tools of the trade: successful DSP-software development and testing,” *EDN magazine*, pp.71-74, Feb.21, 2002.

[2] J. Tian, *Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement*, Wiley-IEEE Computer Society Press, 2005, pp.27-40.

[3] R. G. Mays, "Applications of defect prevention in software development," *IEEE Journal on Selected Areas in Communications*, vol.8, no.2, pp.164-168, Feb.1990.

[4] R. S. Pressman, *Software Engineering: A Practitioner's Approach, Sixth Edition*. McGraw-Hill, 2004, pp.386-419.