JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2006**

Vol. 5, No. 6, July - August 2006

On Assuring Software Quality and Curbing Software Development Cost

Won Kim, Samsung Electronics, Suwon, S. Korea

Abstract

Software quality and software development productivity have been topics of major interest and concern for the past three decades. The recent rapid growth in the size and complexity of software, and the cost of developing software, has given a greater sense of urgency to finding ways to assure quality of software and bring the cost of developing software under some semblance of control. In this article, I will outline ways of assuring software quality and curbing software development cost.

1 INTRODUCTION

To assure software quality, a few things have to be done right. The development team has to be properly staffed and organized. A development process has to be in place and followed by members of a development team. To curb the cost of developing software, a few things have to be done. Beyond possibly outsourcing development to where the cost is much lower, the in-house development team has to work as a cohesive team, and the caliber of each member of the development team needs to be continually and appropriately upgraded. Reuse of software assets has to be maximized. In the remainder of this article, I expand on the two issues of assuring software quality and curbing software development cost.

2 ASSURING SOFTWARE QUALITY

Software quality is determined simply by people and process. I will discuss the process aspect first.

The Process Aspect

A software development process is the specification of a collection of steps involved in developing software, an ordering of the steps, and a collection of deliverables in the course of development. Various software development processes have been proposed and

Cite this column as follows: Won Kim: "On Assuring Software Quality and Curbing Software Development Cost", in *Journal of Object Technology*, vol. 5, no. 6, July-August 2006, pp. 35-42 http://www.jot.fm/issues/issue_2006_07/column5

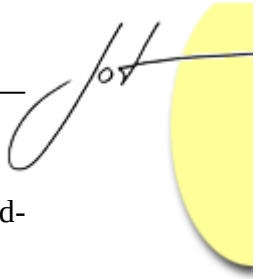
adopted over the years, including the waterfall process, iterative process, Rational unified process, etc. Some process is required for a software development project, even if it is a one-person project. There is no such thing as "the best" process for the development of all software. What is an appropriate process depends on a number of factors, including the nature of software to be developed, the competitive pressure that forces the development schedule, the size and quality of human resources that can be committed to the development efforts, whether the software will be maintained and reused, whether human lives or very expensive consequences are at stake, etc.

Any software development process always includes the following steps, regardless of how well each step may be executed or whether some of the steps are well-formalized: project planning, software requirements analysis, architecture design, detailed design, test planning, coding, testing, and release. The first five steps constitute the "upstream" of the process, while the latter three steps the "downstream". If the upstream is not done properly, the downstream efforts may suffer greatly. A change in the upstream, while the downstream efforts are underway, forces changes in the downstream. For example, if a requirement is added or changed while coding is well underway, the design has to change, test plan has to be updated, and coding has to be changed. Frequent or significant changes in the upstream can wreak havoc on the downstream, resulting in software that is poor in quality, poor in reusability, and high in post-release maintenance cost.

The deliverables are the outputs corresponding to each of the steps: a project plan, a software requirements specification, an architecture design specification, a detailed design specification, a test plan, source code with block comments, test suites, and release notes and manuals (user manuals, administrator manuals, tutorials). When outsourcing any or all of the steps of a development process, the deliverables are the tangible means of communication and evaluation of the results and quality of work. As such, the delivery of each deliverable represents the synchronization point between the in-house development team and the outsourcing partner.

The ordering of the steps in the waterfall process would be the order in which they are listed above. In general, however, all the steps except release need to be iterated. For example, a project plan is subject to change based on a detailed requirements analysis; requirements analysis in turn may not really be completed without an architecture design and detailed design, as the design efforts may reveal that some of the requirements cannot be met; detailed design may need to be changed after coding reveals mistakes in the design, etc.

Below I will discuss each of the steps in a software development process, except project planning and test planning. The requirements analysis step typically starts with a requirements specification that the marketing team has produced. Leaders of the development team analyze the requirements specification, and make changes to it. Changes typically reflect considerations of architecture quality attributes and constraints related to development efforts. Architectural quality attributes include performance, scalability, reliability, extensibility, availability, security, portability, etc. Constraints related to development efforts include estimated schedule, availability of human



resources, development environment, availability of development-related tools and third-party software, etc.

In the architecture design step, architects design and document the architecture. Software architecture consists of elements that comprise the software, and relationships and interactions among the elements. An element may be a module, process, or thread. Relationships between elements include aggregation, specialization, depends, etc. Interactions between elements include data passing, control passing, communication, etc. All relevant architecture quality attributes must be considered in the architecture design. A design may be documented in large part in textual form. It may (and should) be augmented with visual notations, such as informal boxes and lines (a box representing a software element, and a line representing a relationship or interaction between elements), the increasingly popular UML (united modeling language), or some combination of notations.

In the detailed design step, architects or developers map the architecture design to function/class level detail, including the input and output parameters of each function, or the attributes and methods of each class, and interactions among the functions/classes. Techniques, algorithms and data structures relevant for the implementation of all required functions and all architecture quality attributes must be included in a detailed design specification.

There are two aspects of the testing step that need to be emphasized. First, contrary to the false impression that the waterfall process may imply, the testing step should not follow the coding step; instead, coding and testing must be started simultaneously and performed simultaneously. In other words, the coding and testing steps should really be merged into a single step. Only in this way, defects may be detected early, isolated within a coding unit (a function or a class), and fixed by the developer who wrote it. If testing is done after many coding units are assembled, pinpointing the location of defects becomes much harder and time-consuming. Further, the same type of defect may have been repeated in other code units, requiring all instances of the same defect to be pinpointed and fixed.

Second, the testing step itself is not a single simple step. Instead, testing consists of seven sub-steps or elements, including code self-inspection, unit/module testing, code review, integration testing, system testing, field testing, and acceptance testing. Below I discuss each of the sub-steps or elements.

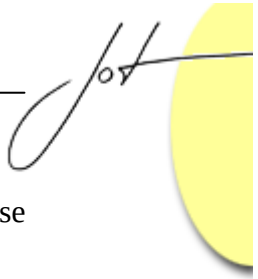
Code self-inspection is the inspection of a code unit to be done by the developer who wrote it. It is to be done when a code unit has been written, and done before or after compilation but ideally before attempting to run the code. Code self-inspection is the least expensive (in terms of time) and most efficient (in terms of the number and types of defects to detect and fix) testing method. The most serious objective of code self-inspection is to discover errors in program logic. However, it may be used effectively to detect defects that may become very troublesome to detect at runtime. A minimal checklist should include potential defects related to memory reference (in C or its derivative languages -- incorrect pointer chasing, dangling pointer, indexing of memory outside an array or structure, memory leak), responding to errors (invalid input, error

return from a function call), missing conditionals (missing case, missing default case in a switch statement, incorrect if-else chain), and termination (incorrect loop count, incorrect exit from the loop, incorrect computation due to incorrect looping).

Unit/module testing is the creation and execution of a test program to test a code unit or a module (a small assembly of code units that collectively implements a particular required function). The test program makes calls to the code unit or module being tested with a combination of values for the input parameters, and compares the resulting outputs against expected correct outputs. Unit/module testing should consist of positive testing and negative testing. Positive testing uses only valid input values, while negative testing uses only invalid input values. Through negative testing, it must be established that the code unit will not crash or hang, but rather respond to an invalid input with appropriate error recovery, such as return with an error message. Both positive and negative test cases should account for "boundary" values, values at the boundary and values just outside the boundary of correct input. Unit/module testing may primarily be black box testing, and not bother much with white box testing. When a proper code self-inspection precedes unit/module testing, most of the defects that can be identified through white box testing will have been fixed.

Integration testing is the testing of a major module or subsystem that results from assembling a set of code units. It is difficult to draw the line between a module and a subsystem, as it depends on the nature of the required function that it implements. Whether it is a module or a subsystem, in order to be amenable to testing, there should be a user interface (application programming interface or graphical user interface). It is this user interface against which integrating testing exercises the module/subsystem. As with unit/module testing, integration testing consists of black box positive and negative testing. However, as a module involves code units, and there are in general required interaction orderings among code units, integration testing needs to be more sophisticated. Specifically, integration testing should include scenario-based test cases, both positive and negative. A scenario is a sequence of invocation of code units to implement certain required function. A positive scenario-based test case is one that invokes the code units in the required correct ordering, while a negative scenario-based test case is one that invokes them in incorrect ordering. Further, the test cases created for integration testing may be organized into a test suite, and may be rerun whenever some changes are made to the module/subsystem under test. This is regression testing whose purpose is to make sure that changes to existing code did not trigger defects elsewhere in the code.

Code review is the review of source code to both detect defects and program logic, and to identify undesirable coding practices. There are two types of code review, depending on whether the software is developed by the development team or by outsourcing partners. When software is developed by the in-house development team, the most effective way is for technical leaders of the development group to review the code of each member of the group, rather than for peers of comparable skill level to review each other's code. When software is received from outsourcing partners, the in-house development team takes it over and extends it and modifies it. In such a situation, the in-



house development team has to review all or selected modules of the code for the purpose of learning them.

System testing is essentially integration testing against full software when all code units have been implemented and assembled. As system testing is in effect the "final" testing, the scope of testing must be all-encompassing, including not only all required functions, but all applicable architectural quality attributes, and boundary conditions on system-wide parameters. System testing must stress the software on all relevant major parameters, such as the number of files, the number of records, the number of simultaneous users, the input rate, etc. The system test cases should be organized into a system test suite, managed and preserved as part of project deliverables.

There is one important gray line with system testing. The responsibility of system testing should be shared between the development group and the SQA/testing group. The development group should do at least the "initial" system testing, on all required functions and on all relevant architecture quality attributes, before turning it over to the SQA/testing group. The SQA/testing group should fully exercise the software by adding scenario-based test cases, test cases that stress the limits of the software on required functions and architecture quality attributes.

A subset of the system test suite that can be run automatically in about six hours may be selected and used for regression testing nightly by the development group. Such a nightly regression test suite may be run automatically after hours, immediately after an automatic daily build of software under development at the end of the day each day.

If the software is to be used in a geographical region that is different from where it is developed and tested, the SQA/testing group may take the field testing step. For example, in the case of embedded systems such as televisions, cell phones, printers, etc., different countries may use different signals and different formats. Often, only extensive field testing can expose certain defects, including some major ones.

The acceptance testing step is the final testing step. This should be the responsibility of a group outside the development team (that is, not the SQA/testing group) whose responsibility is to give a go or no-go decision on the release of software.

When software is released, often release notes accompany the software. Release notes include a list of non-critical bugs that are known but not fixed, and instructions for working around them. Manuals, including user manuals, references, tutorials, administrator manuals, go out with the software. All deliverables are stored for management and reuse for the next-release efforts.

The requirements analysis, architecture design, detailed design, coding, and testing require formal and informal reviews. The project plan, requirements specification, architecture design specification, detailed design specification, test plan, and system test suite must all be reviewed by leaders of the development group and the SQA/testing group. Source code must be reviewed within the development group as noted above.

The People Aspect

I will now discuss the people aspect in assuring software quality. The development team needs to be organized properly and duties of each group within the team, and each member within the group, must be clearly defined. A development team in general consists of people who play four roles: project management, development, SQA/testing, and maintenance. Duties of project management include project planning, schedule creation and management, people and resource management, process compliance management, interaction with management and marketing. Duties of the development group include requirements analysis, architecture design, detailed design, coding, and testing (that developers must perform). Duties of SQA/testing include test planning, test case generation, testing, defect management, software release, and process improvement. Development and SQA/testing have a shared duty of evaluating and adopting development and testing tools. Duties of the maintenance group include fixing defects discovered after release. A development team without a maintenance group gets to be hampered in its next release efforts by having to lose a part of its next-release development resources (often the most talented ones) to the unplanned task of tracking down and fixing defects in the previous releases. (As such, the need for the maintenance group is not really for assuring software quality.) (In the background, of course, there is another role, that of system administration. Duties of system administration are to maintain computers, and development and computing environment for the development team, so that the development team can perform their duties.)

Staffing the SQA/testing and maintenance groups is usually a serious challenge. Due partly to the seemingly unglamorous nature of the work involved, and partly to the impression of a relatively lower status associated with it, software engineers are loath (to the point of finding other employers) to being assigned to these groups. They much prefer to belong to the development group (and create lots of defects that the SQA/testing will uncover, and that the maintenance group will track and fix).

There are two practical ways to solve this problem. One is to require the development group to perform those elements of testing that are rightfully developers' responsibilities. These include code self-inspection, code review, unit/module testing, integration testing, nightly regression testing, and initial system testing. Full system testing then falls on the SQA/testing group. Another is to periodically move people. Some members of the development group may be moved to SQA/testing and maintenance, some from SQA/testing to development and maintenance, and some from maintenance to development and SQA/testing. The period would best be determined based on development cycle and career development needs for people involved. Leaders of the development group and the SQA/testing group should in principle be exempt from this role-rotation scheme, as these two groups need steady and firm leadership to function properly.

3 CURBING THE DEVELOPMENT COST

One currently fashionable way to curb software development cost is to outsource some steps in the development process to where the cost of skilled labor is much lower. Of course, the added cost of managing outsourcing partners and ensuring that miscommunications do not mitigate productivity has to be taken into account. Beyond the outsourcing route, broadly there are two ways to curb software development cost. One is to increase productivity of the in-house development team and each member of the development team. Another is to increase reuse of software assets. I will explore both below.

To increase productivity of a development team, the project schedule has to be understood and bought into by all members of the team. Further, as noted earlier, the development team has to be properly organized and staffed, every member of the team must understand his duties and work with other members of the team, and each group within the team must understand its duties and work with other groups. There should be a career development plan for each member of the development team, so that he may receive training on new technical skills, project management skills, technical communication skills, etc. to become a more valuable asset for the development team (and the organization that includes the development team) for the future. The need for the maintenance group within a development team is primarily to increase productivity of the development team in the next-release efforts. The moving of selected members of the development team from one group to another periodically, as mentioned earlier, serves the purpose of training them on different aspects of software development, besides facilitating the staffing of the SQA/testing group and the maintenance group. All the formal and informal reviews of the deliverables of the development process also help train members of the development team, besides helping them understand the requirements and design of the software being developed and tested.

Beyond enhancing the caliber of each member of the development team, each member should learn to use appropriate development and testing tools. Some are commercial offerings, while others are free open source tools. There are of course debuggers. There are also tools that analyze source code for various aspects of the structure of the code and potential problems; that automatically generate test cases; that launch test suites automatically and compare results against stored correct results; that help manage defects; that profile performance and system resource usage; that estimate test coverage; etc. There are simulators and emulators that allow testing without the necessary hardware being in place. There are tools, such as DoxyGen and JavaDoc, that help extract elements of the source code documentation for automatic inclusion in detailed design documentation. There are tools for managing versions and configurations of source code and other documents; for helping create requirements and design documents in visual notations. When used judiciously, these tools can help developers save a lot of time, and provide valuable insight into the structure and behavior of the

software they are developing. They are of course essential in managing and exchanging deliverables.

To increase reuse of software assets, the requirements analysis, architecture design, detailed design, and coding must be done properly. Reusable software assets include all deliverables of the development process, not just the undocumented source code or the binary code. If architecture design and detailed design do not properly reflect both the functional requirements and architectural quality attribute requirements, both the designs and, most likely, also the code resulting from the designs, will later have to be modified substantially for the next release. If coding is done poorly, it will tend to require substantial modifications in the next release to improve performance and reliability, to reduce footprint size, etc. Of particular importance, with respect to reuse, is to "predict" future modifications (adding functions), porting (different OS or chipset, different locale), extension (interfacing to different third-party software, different peripheral devices), and scaling (larger data set to manage, larger number of simultaneous users to support, larger number of devices to support).

About the Author

Won Kim is Senior Advisor at Samsung Electronics, Korea. He is Editor-in-Chief of ACM Transactions on Internet Technology (www.acm.org/toit). He is Global General Chair of the Human.Society@Internet International Conference. He is the recipient of the ACM 2001 Distinguished Services Award, and is an ACM Fellow.