Quick Start to Quality – Five Important Test Support Practices

Louise Tamres
The Tamres Group, Inc.

Abstract

As testers, we understand the virtues of clear requirements, effective configuration management, software inspections and reviews, project planning, and project tracking. But how does the test group influence an immature organization to improve in these areas? Sometimes, the test group has to use short cuts, partial implementations, and even a clandestine approach to get things done. Practical strategies used at several software organizations have quickly improved product quality by addressing these five critical development practices.

1. Key Processes for Software Quality

Key processes for producing quality software include:

- Requirements
- Configuration Management
- Reviews and Inspections
- Project Planning and Project Tracking

These processes are vital for producing quality software: failure to perform any of these processes adequately is a leading cause of poor product quality. While it's true that testing will find lots of problems; testing by itself cannot improve the quality of the software.

These key processes are addressed by the Software Engineering's Institute Capability Maturity Model (CMM®) as well as other software engineering standards. Regardless of the development model or standard used, they all emphasize these key processes as necessary for successful software development.

Poorly executed key processes often divert a tester's activities into tackling non-testing tasks. Of course, the developer also suffers from poor processes and must often do extra work due to the resulting chaos. The relevance of poor processes on testers includes the following:

- Incomplete and deficient requirements increase the tester's burden of identifying missing and clarifying ambiguous information. The tester must then identify the proper authority who can answer the resulting questions. Failed tests and re-execution of tests often result from vague requirements.

- Poor configuration management often causes testers and developers to waste time tracking proper file versions and software builds, sometimes having to rebuild complex software products. Sometimes the tester cannot reproduce problems because of mismatches between the development and the test environment. Good configuration management helps ensure the reliability of a changing environment.

- Inadequate reviews allow defects to infiltrate the software. The tester eventually finds the problems during test execution, instead of having reviewers detect the same problems much earlier. The tradeoff is whether to find bugs early (in a review) or later (during system testing) when they are more expensive to fix. Finding bugs earlier reduces the amount of rework for developers and allows the tester to focus more on acceptance level testing.

- Poor project planning results in time crunches often due to incomplete or optimistic estimates. Often there's also a misunderstanding of what features are available for testing. Developers may be rushed to hand over their software. Sometimes features are cut from delivery and other features are not even tested due to lack of time. Slippage in the developers' schedule typically reduces the time allotted to testing.

This paper focuses on simple changes that a tester can do to reduce the problems caused by inadequate processes. A tester can easily modify his or her own work habits and can often influence those working in proximity. As opposed to a manager tasked with the job of instituting change, a tester may not always have the authority to enforce change.

The strategies presented below are quick and dirty. Often, applying shortcuts is a clandestine approach to improving existing processes. Naturally, these quick changes do not replace proper execution of key disciplines. They are, however, steps in the right direction for those organizations that want to improve their processes.

2. Process Shortcuts

By targeting each process' core intent, the tester can make some effective changes that can quickly improve product quality. Partial implementation of a key process only yields partial benefits, but this is an improvement for those organizations that perform these tasks minimally. Consequently, it is important to understand the pros and cons of shortcuts. There are times when they're warranted, and times when loss of rigor is unacceptable. In an immature organization, some improvement is better than nothing. For a mature organization, applying a shortcut is often a step backwards.

Shortcuts do offer some of the following advantages:

- Providing small, gradual steps with incremental successes
- Performing changes that often fall below a manager's radar, and thus requiring no management approval or intervention (Otherwise, some managers may declare that improving processes is not the tester's job)

- Teaching quality-related concepts to the rest of the development team
- Showing initiative on the part of the individual who instigates the changes
- Withholding the fact that changes are forthcoming will avoid resistance by those adverse to change
- Jumpstarting process improvement at an immature organization
- Making changes that facilitate the tester's job, as well as benefit the rest of the team

Whether shortcuts are suitable depends on the risks that an organization is willing to take. Reasons for avoiding shortcuts might include the following:

- An abbreviated paper trail may fail to satisfy stringent documentation requirements
- No metrics collected
- No signoffs by responsible persons
- Incomplete process
- Accreditation by an external standards organization may be withdrawn; an external auditor would regard shortcuts as a breach of accreditation.

Naturally, you lose the rigor and paper trail of a formal process, but you have to start somewhere and this provides a baseline for future improvement. Shortcuts provide incremental change resulting in suitable successes.

2.1 Requirements

When you strip requirements to the bare bones, the real purpose is to identify the system's intended behavior for each type of input situation. In reality, good requirements address more than simply stating conditions and results; they deal with such issues as scope, boundaries, environment, and performance. However, in order for the tester to achieve some quick-and-dirty benefit and reduce misunderstanding, we'll direct our attention to answering the question "What happens when ...".

Ideally, each unambiguous set of input conditions is mapped to a description of the corresponding system's behavior. Compare this to the definition of a test case: identify input states and then specify the expected outcome. In the process of developing system level test cases, the tester can easily identify poorly documented requirements – whether it's ambiguous input conditions, missing information, or unspecified expected outcome.

Prior to actually writing system test cases, the tester might want to use modeling tools that help organize thoughts, detect missing requirements, and formulate tests. Such modeling tools include outlines, tables, graphs, and use cases. For more information on how to use these tools for developing test cases, refer to [Tamres][1], whereas [Cockburn][2] and [Schneider][3] are excellent references on use cases.

By developing system test cases, the tester can pinpoint where requirements fail to describe the outcome. If the project is under extreme pressure, the tester should take

charge of completing the test descriptions by replacing missing information with reasonable assumptions – and indicating this as such. This can be of enormous assistance to the proper authority (often a manager or lead developer, depending on the project) who then merely has to approve the modified description rather than schedule resources to complete the product definition. Sharing the information with developers also previews what tests are planned, giving them the opportunity to bullet-proof their code if necessary. In this approach, the completed test case document provides additional information that clarifies the system's intended behavior.

Despite all the gain achieved by using test cases to clarify requirements, this shortcut does have its drawbacks. However, the shortcomings of this process should really be viewed as the opportunity for improvement in the next project cycle. Drawbacks to using test cases as a replacement for requirements include the following:

- Incomplete descriptions of system behavior
- System described as a set of inputs and outcomes
- No "big picture" or problem statement for describing the project's main purpose
- No requirements management
- Possibility of overlooking important boundary, performance, and configuration tests
- Possible addition of new requirements

2.2 Configuration Management

Virtually every company uses, at minimum, a commercial version control tool and most have a configuration management tool. Today's projects are too complex to control components manually. Despite the proliferation of tools, you must still use them and define the software component's architecture, naming conventions, and code structure.

Good configuration management practices can exist despite the lack of a formal configuration management process (although managers should assess risk and calculate productivity loss). The product build activity is one area where the project team might not employ the full benefits of configuration management. Different developers might use different conventions, rather than having uniform conventions for the entire project. The tester should work directly with developers and have them decide on necessary conventions. Most developers understand the configuration management headaches and know what needs to be done. Below are some tactics to help steer configuration management into a more usable state. Of course, not all steps are necessary based on the project's immediate needs.

- Focus first on version control and build control.
- Establish a simple structure for organizing files, including (if appropriate) the definition of symbolic variables.
- Create build scripts to guarantee a repeatable build process.
- Select a cutoff date after which all components must be placed under version control and must conform to the new methods. Avoid making this retroactive,

since it will only increase the amount of work for developers, and testers are only likely to validate newer versions of the software anyway.
- Document all decisions in an e-mail and send a copy to the entire development team.

The tester must also utilize and enforce good configuration management practices such as:
- Placing all test-related products (such as test cases, scripts, test documents, and tools for building the test environment) under version control.
- Accepting only software releases that were created by and that conform to the established configuration management conventions.
- Referring to every software build – as well as any document – by its version, including references in bug reports.

In some cases, an organization may not even have a version control tool. (Considering the proliferation of tools, this seems unlikely). Nevertheless, version control can be achieved despite the lack of a configuration management tool. The quickest approach is to use one folder for each different build version of the product. Every file – which may include source code, build scripts, and sometimes the operation system, link tools, and other build tools – needed to make a specific software environment resides in that particular folder. Of course, there will be much duplication among various folders. But at least, all the components necessary for each software build reside in one place.

Any improvement is a step in the right direction; however, these configuration management shortcuts do exhibit the following disadvantages:
- Focus on version control rather than full configuration management
- No company-wide procedures
- Possible reliance on manual actions

2.3 Reviews and Inspections

We have all had the misfortune of attending bad reviews. A group of people shows up for a meeting, where the majority sits down and opens up the document, reading it for the first time. When someone mentions the existence of a problem, several people in attendance discuss solutions, while the rest watch in silence. This is a waste of time. The number of problems found compared to the amount of time spent does not justify the so-called review meeting. When reviews are inadequate, problems that should have been detected by reviewers infiltrate the software until they're eventually discovered by the tester.

Lack of training and lack of management support are the main reasons that an organization has ineffective reviews. For organizations that do not understand the inspection process, I make no distinction between the terms *reviews* and *inspections*, because the benefits of neither are achieved. The real differences between these two concepts are in thoroughness, diligence, formality, and metrics.

Working at the grassroots level, the tester may be most effective by coaching one or more developer in the finer points of inspections. The goal is to teach people to think critically. Everyone wants to do good work and most understand the value of reviews, but not everyone has a critical eye or the attention to detail that makes for a good reviewer.

The key component of reviews and inspections is in detecting defects early. Desk-checking is the best way to achieve this. Thus, make the reviewers spend time actually reading through the material, such as requirements, designs, source code, user guides, test document, or any other artifact created in the course of product development. Establish these guidelines when mentoring a new reviewer:

- Provide a quiet place without interruptions for performing the desk check.
- Limit the amount of material to review, no more than 10 pages at a time.
- Limit the amount of time spent reviewing to one hour.
- Set a goal: challenge the reviewer to find 10 issues per page.
- Provide any reference documents that are necessary in cross-checking the material's contents.

For large documents, the tester should parcel the work into smaller, manageable pieces. Furthermore, to zero in on specific items, the tester can assign a particular role, examples of which could be to focus exclusively on cross-references, calculations, flow, or another specific point-of-view.

Depending on the organization's culture, the tester may want to seed some bugs. This consists on deliberately inserting known problems to determine whether reviewers are reading the material thoroughly. Also, if reviewers are forewarned about the seeding of bugs, they may be less intimidated to report on some issue that they might deem trivial or embarrassing.

Have the reviewer mark all comments and questions on the document itself. If time permits, gather all reviewers in a room and read off the issues they've found. The benefit is twofold: (1) groups tend to collectively find new problems due to synergy, and (2) participants learn more about conducting reviews effectively. This review shortcut uses no formal scribe: each reviewer hands over his marked up copy to the author. Any additional notes are the author's responsibility. Additionally, the author is responsible for any necessary follow-up. However, any person can designate a specific expert to check the changes. The tester, meanwhile, may wish to note some important issues to help create specific tests. At this point, the amount of information to record and the rigor of any follow-up should be decided on a case-by-case basis.

It is also vital that the tester lead by example. The first step is to conduct reviews – even informal ones – on all test artifacts (such as test plans, documents, test cases, scripts, etc.). In addition, when invited to a review (whether formal or not), the tester must prepare appropriately and show up at the review meeting with a detailed list of issues. That way, the rest in attendance can see how preparation helps the review process be effective.

The metrics data and formal documentation are a byproduct of a well-executed inspection process. It is best to focus first on defect detection than in recording metrics. The potential problem is that, in a poorly-performed inspection, managers could misinterpret the low number of defects found as false evidence that inspections are not worth the time. In this case, what you're probably recording is that the inspection process – but not the material under review – is faulty. Such erroneous statistics will not get inexperienced managers to support inspection process improvements. Another caveat in that some misinformed managers could use metrics against the developers and wrongly accuse them of creating defecting software.

This abbreviated review format does put extra burden on the tester. The hope is that the rest of the development team will continue to learn about and improve the inspection process. Although finding defects is the ultimate review goal, this falls short of implementing rigorous inspections due to the following:

- No formal issue log
- No metrics collected
- No follow-up procedures
- No signoff or approval
- No moderator or scribe
- No formal inspection training
- No management support

2.4 Project Planning and Tracking

Even when suitable project schedules exist, the level of breakdown may not necessarily give the tester enough information. Testers may want to use their own adaptable schedule to facilitating project planning and tracking. This customized schedule applies only to the testing activities, yet in a manner visible to the rest of the development team. By demonstrating a disciplined approach to project planning and tracking on a small scale, hopefully others will follow suit.

Borrowing some concepts from Extreme Programming, one can establish a strategy that supports highly changeable plans and promotes communication. The planning scheme makes use of index cards pinned onto a board. As an alternative, Post-It notes placed on a large sheet of paper are also easily moved around. A paper-based system is portable, cheap, easy to update, and easy to view.

The general approach for developing a testing schedule is as follows:

1. Create a matrix on a board (see Figure 1) that lists the names of the testers along the top and the dates of the next four weeks down the side. The last row refers to tasks that are not yet scheduled.
2. Write each testing task on a card and provide an estimate. Ideally, each task should be one week or less in duration. Try to break down the larger tasks into smaller components; otherwise, some cards will have to span several weeks.

3. Sort each task in order to be completed, whether by priority, highest risk, product delivery requirements, or whether the master project plan dictates the task's completion date.
4. Assign each prioritized task to a tester and post the tasks on a board in the anticipated completion order.
5. Encourage conversations between testers, developers, and managers with regards to testing priorities and identifying bottlenecks.
6. When a task depends upon information or delivery from a developer, discuss this need with the developer and have him commit to a date. Write this expectation on a new card and add it to the board.
7. Adjust the plan every week (or every two weeks). This may involve checking off completed tasks, reordering tasks, or adjusting estimates. Some tasks may be delayed or moved around due to other new circumstances.

	Person 1	Person 2	Person 3	Person 4	Person 5
Week 1					
Week 2					
Week 3					
Week 4					
Tasks not yet scheduled					

Figure 1: Sample Planning Matrix

The continual cycle of planning something, assessing status, and then readjusting the plan, blurs the line between project planning and project tracking; the communication of status is what's important. Regardless of the planning methodology used, the *plan* is not as important as the art of *planning*: the exchange of information among team members, including tradeoff analysis and compromises.

The entire team can visualize test progress simply by glancing at the board. By viewing the number of test items, the team – including developers and managers – can quickly assess the magnitude of testing activities. If the amount of testing exceeds the time allocated, the team must discuss tradeoffs and prioritize the remaining activities.

Once again, testers lead by example: they define, estimate, and prioritize their tasks. The hidden agenda is that by making the testing schedule visible and easily modifiable, the project manager might employ a similar approach on the project level.

Communication is the primary objective of this abbreviated planning scheme. When compared to a formal project plan, it exhibits the following drawbacks:

- No emphasis on dependencies or critical paths
- No management support
- No pretty picture and no formal chart
- No software tool to capture the information
- No correlation between tasks on board and official schedule

3. Next Steps

All strategies presented in this paper will help improve some processes. Even though this is a good start, much more work remains. The following ideas will help identify the next course of action:

- Get feedback from testers, developers, managers, and anyone else involved in the project.
- Collect a list of lessons learned.
- Announce the improvements achieved in the current project cycle.
- Provide a list of suggested changes for future projects.
- Decide whether to continue on the sly or whether to sell the case for process improvement to managers.

4. Summary

Focusing on the core intent of key processes provides some quick-and-dirty strategies to help improve software quality.

Requirements: Modeling tools that transform information from the requirements can help isolate missing information. Once input conditions are identified, each is mapped to an expected outcome. This forms the basis for system test cases. As the tester finds input conditions for which the outcome is undefined, the tester makes an educated guess and presents this proposed information to the proper authority for agreement.

Configuration Management: Almost all organizations use version control. Problems typically arise due to inconsistent tool usage or poorly defined procedures. By helping the developers establish a simple version control process and recording these decisions, the project team can build software in a consistent manner. Testers must refer to every software version by its unique version and accept only software produced through the established build procedures.

Reviews and Inspections: The most important part of a review is for each reviewer to spend quality time, undisturbed, focusing on a small portion of material with the expressed goal of finding defects. The resulting information is shared with the author, who then follows-up as appropriate, and the tester, who decides what information to apply in testing. Go through this process several times to make reviewers focus on defect detection, before bringing in more discipline, such as conducting a logging meeting, recording issues found, and collecting metrics.

Project Planning and Tracking: Project planning consists of (1) identifying tasks, (2) estimating the duration for completing each task, (3) deciding when to do each task, and then (4) communicating this information with the rest of the project team. Discussing the testing tasks with the developers will help create mutual consensus on how their input and deliverables affect the tester's activities. Evaluate progress on a weekly basis and adjust estimates and task order as necessary.

Even with minimal management support or lack of authority, the tester can assist in improving some processes. While this is good news, it is a far cry from a fully mature organization with well-define and effective processes. In the meantime, small incremental changes will eventually guide the organization down a path of future improvements.

5. References

To learn more about modeling requirements and defining test cases:

[1] Tamres, Louise. *Introducing Software Testing.* London, UK: Addison-Wesley, 2002.

To learn more about writing use cases:

[2] Cockburn, Alistair. *Writing Effective Use Cases*. Boston, MA: Addison-Wesley, 2001.

[3] Schneider, Geri and Jason P. Winters. *Applying Use Cases, A Practical Guide (2nd edition)*. Boston, MA: Addison-Wesley, 2001.

To learn more about defining requirements:

Robertson, Suzanne and James Robertson. *Mastering the Requirements Process*. Harlow, England: Addison-Wesley, 1999.

Wiegers, Karl E. *Software Requirements.* Microsoft Press, 1999.

To learn more about reviews and inspections:

Fagan, Michael E. "Advances in Software Inspections," *IEEE Transactions on Software Engineering*, Vol. SE-12, no. 7, July 1986.

Freedman, Daniel P., and Gerald M. Weinberg. *Handbook of Walkthroughs, Inspections, and Technical Reviews, Evaluating Programs, Projects, and Products*, 3rd edition. New York, NY: Dorset House Publishing, 1990.

Gilb, Tom, and Dorothy Graham. *Software Inspection.* Workingham, England: Addison-Wesley, 1993.

To learn more about configuration management

Buckley, Fletcher J. *Implementing Configuration Management: Hardware, Software, and Firmware*, 2nd Edition. Los Alamitos, CA: IEEE Computer Society Press, 1996.

Leon, Alexis. *A Guide to Software Configuration Management*. Boston, MA: Artech House, 2000.

Mikkelson, Tim, and Suzanne Pherigo. *Practical Software Configuration Management*. Upper Saddle River, NJ: Prentice Hall, 1997.

To learn more about planning in Extreme Programming:

Beck, Kent. *Extreme Programming Explained: Embrace Change*. Boston, MA: Addison-Wesley, 2000.

Beck, Kent and Martin Fowler. *Planning Extreme Programming*. Boston, MA: Addison-Wesley, 2001.