Cem Kaner, Ph.D., J.D.
Attorney at Law
kaner@kaner.com

P.O Box 1200
Santa Clara, CA 95052
408-244-7000

NEGOTIATING TESTING RESOURCES:

A COLLABORATIVE APPROACH

Presented at Quality Week, 1996

In the session,

How to Save Time & Money in Testing

This session is about saving time and money. As in all aspects of product development, a time-honored way to waste time and money is to work without clear requirements, in a chaotic rush to meet unrealistic and unachievable deadlines. As testers, we often see the mess that this creates. But some of us seem unaware that, even if the overall project is a mess, we can go a long way toward establishing clear requirements and expectations in the testing sub-project itself. This paper applies what I think is largely common sense and common knowledge about project management and consensus-driven engineering. It describes an approach that I've used a few times – not always successfully – since 1983. The details will vary according to your company's circumstances and schedule, but the ideas, I think, have general application and merit.

I do most of my work with companies that develop consumer-oriented software for retail sale, or whose development processes look like consumer software companies. Some of the challenges in these organizations:

- Their schedules are often unrealistically optimistic.
- Their requirements aren't fully worked out -- perhaps because they're still building something that no one has ever built before, and they're still learning what they want to release. Or because they're not sufficiently skilled at requirements analysis. Note that there is no contracting customer, so there is no one external to the company who can sign off on a requirements document.
- Their specifications make the requirements documents look complete and up-to-date.

Some of these projects result in great products. Many yield mediocre products, despite heroic (if insufficiently organized) efforts by the staff. And too many yield worthless garbage, that might or might not be foisted off on unsuspecting customers.

There are many ways to improve the overall development processes, but global process decisions lie in the hands of the project managers and their management, rather than in the hands of testing groups. Therefore I won't speak to those global approaches today.

Even though we don't have sufficient authority to reorganize the overall development processes, there are things that we can do, as testers, to substantially improve our own processes. As a

byproduct, they help the other product development managers (the managers of the programmers, the writers, the designers, etc.) bring the rest of the project under control as the product comes into the testing group.

As always in my writing, I am focused on black box testing, which I think of as testing that is driven from the outside, from the customer's perspective, and is probably carried out without knowledge of the details of the underlying code.

OBJECTIVE

My goal is to facilitate a corporate consensus on testing tasks, priorities, and times. That is, I want to end up with:

- a bottom-up task list of every test-related task or area to be tested that requires a day or more of testing;
- corporate agreement on the priority of each task. I want to know the desired depth of testing;
- a realistic assessment of the amount of time required for each task, or area of testing (at the agreed level of depth);
- sufficient resources (time, money, people) to meet the testing requirements;
- a method for tracking performance against the assessment, and a means of updating the assessment when estimates turn out to be imperfect.

BACKGROUND THINKING

I start from the position that there are other stakeholders in the company who have as much of an interest and stake in the quality of the product as the testers. This includes the marketing, programming, customer service, and project management staff. They should have a big say in prioritizing the testing of areas of the program.

Many testers don't accept this position. Their experience is with marketing managers and project managers who seem to be focused on shipping products quickly, whether those products are any good or not. In these organizations, the tester fights like crazy to keep the company from shipping an unacceptable product. After a long series of battles, the product is eventually released. These testers "know" that in their companies, if they didn't fight for quality no one would.

This division of attitude and responsibility – and the naming of appraisal groups (such as testing) as Quality Control or Quality Assurance – traces back to the Taylor theory of management, developed in the early part of the century.[1] With the advent of mass-production, Taylor and his followers centralized the inspection process and made it an independent function in the factory. This approach might have been fine for the 1920's, but it is antiquated today.[2]

One of the most troublesome problems of this model is that it lets everyone else feel free to rely on the testers for quality-related advocacy. It leaves things up to "Quality Assurance" (the testing group), standing alone, to argue for adequate testing, for sufficient time and money for testing, and for facing the discoveries of errors that make the product unacceptable to release. I think this is pathological.

---

[1] See the discussion in J.M. Juran (1992), *Juran on Quality by Design*, Free Press, p. 363 *ff*.

[2] See Juran (note 1).

Also, see K. Ishikawa (1985) *What is Total Quality Control? The Japanese Way*, Prentice Hall, who describes the inadequacy of inspection-driven quality control in Japan (p. 15) and says (p. 25) that the Taylor "method was probably a viable method some fifty years ago, but it is certainly not applicable to today's Japan."

W.E. Deming (1982) *Out of the Crisis*, MIT Press, p. 28, "Inspection does not improve quality nor guarantee quality. Inspection is too late. The quality, good or bad, is already in the product."

Rather than buying into the notion that we testers have to constantly fling ourselves in front of ship-date-driven trains, I try to recruit the other people who have a stake in the product's quality, and give them a say in the level of testing that each area of the product will see. In doing so, I gain understanding, support, and valuable advice. But I lose independence.

In any testing project, no matter how much time we have, or how many people and machines we have, we will never be able to test everything. We will always wish we had another month and another tester. But we won't have these, so we will face difficult prioritization issues. Some things will be tested less thoroughly than others.

Given that we will test some areas less than others, I think it's wise to consciously decide which areas will be shorted, rather than discovering at the end of the project that some things weren't tested as thoroughly as they would have been if we had done our planning.

So suppose that we will prioritize consciously. Who should do it? *Not just the testing group!*

In every project, some reported bugs are "deferred" (to be fixed next time) or otherwise marked as not to be fixed. This decision might initially be made by a programmer or project manager, but it is often very actively examined and reconsidered by representatives of the marketing, testing, customer service, documentation, and other groups in the company. The ultimate decision is a business decision, made jointly by several stakeholders in the company.

When we prioritize an area as low – to be lightly tested – we are making a decision that will result in bugs being missed in this area. The same people who participate in the decision to defer a bug after it has been found should be involved in the decision to not search for the bug in the first place.

STARTING THE NEGOTIATIONS

On the timeline of the typical project that I'm involved with, negotiations would start in earnest four or six weeks before the "alpha" milestone.[3] Your projects might work on different timelines. I'm looking for a time that has the following characteristics:

- Enough implementation is done that we know where most of the toughest unexpected implementation challenges are.
- Enough implementation is done that we know the relationship between the product and its specification (or we know the look of the work-in-progress if the spec is too sparse to be useful).
- We are early enough that there is still uncertainty in the schedule. There might be a published schedule and you might be subject to published staff-and-time constraints but it's early enough that no one will be surprised (even if some executives will feign surprise) if these *predictions* turn out to be incorrect.
- A few identified (on-staff) testers have been assigned to the project. This includes the project's lead tester, who will be actively involved in this process. Others might be added later, but you have a core group of at least two people.
- It's getting to be time for the testing group to publish a firm and credible estimate of the testing effort, against which the staff can be held accountable.

---

[3] Long before this, you must have reached several other agreements (or it's probably too late now). For example, you should have already reached agreement on error-handling conventions. If you are using an automated testing tool that can be made less efficient if the programmers use the wrong development toolkits, you have compatibility-tested their and your tools and dealt with the results. You have prepared a preliminary list of hardware and software environment configurations that will be required for compatibility testing and have started to arrange for loaners, rentals, or purchases. You have worked with the programming staff to build in support for debug monitors, memory and other resource meters, and other testing support code that can be easily designed in early, but only added later with difficulty.

DEVELOPING THE BOTTOM-UP TASK LIST

This is a multi-day task, started by the core group of testers on the project and then expanded. We start by taking over a reasonably large conference room. One member[4] of the group takes on the role of facilitator, running brainstorming sessions. This person probably also records the information on flipcharts.

As in all brainstorming sessions, the goal is to facilitate the free flow of ideas. The facilitator will speak to bring the group back onto topic but will record all comments and suggestions without criticism or other editorial comment.[5]

Day 1 – Gather Information

On the first day, I spend up to four hours building a list of every significant source of available information about the project. Well-organized projects require less of this work. In a project that lacks a formal specification, you can still find a great deal of information. For example:

- e-mail between the project manager and the programmers, defining features and resolving ambiguities in the implementation and design. Many project managers will share this material;
- customer support call records and letters from previous versions of this product or similar products;
- books, magazine articles, *BugNet* reports, data from third-party support providers (buy service contracts for competitors' products and call for support), and other sources of information about the kind of bugs that have been found:
  - in competitive products (so that you can test for similar problems in yours);
  - in other products that have used the same development toolkits that your project is using (so you can look for comparable tool-derived failures);
- externally defined specifications and test suites (such as specs and suites for the C compiler, user interface guidelines or localization guidelines);
- the first draft of the user manual or help (probably not yet available, but it will be soon).

Having listed the sources of available information, we break for the rest of the day and collect as much of this material as we can. We'll refer to this material as needed over the next few days, to generate task list items and to get an idea of the complexity of specific tasks.

Day 2 – Generate an Overall Project Task List

We start with another brainstorming session. The goal is to list every significant area of testing. This includes every functional area of the program. It probably also includes issues that are not exactly functional areas, such as configuration/compatibility, load, race conditions, compliance with externally defined regulations or standards – anything that will take a significant amount of testing time. Anything that could take a few tester-days of work is "significant." "Administration" and "Vacations/holidays" are valid areas. The resulting list contains overlapping areas. We deal with the overlap as we expand the detail, but not here.

---

[4] In the ideal case, you'd like the facilitator to be another tester who has been assigned to a different project, and is on loan to your team only to facilitate/record this meeting. In any case, the lead tester need not act as the facilitator.

[5] For more on effective facilitation and recording, see Kaner, S., Lind, L., Toldi, C., Fisk, S., & Berger, D. (1996) *Facilitator's Guide to Participatory Decision-Making*, New Society Publishers; Doyle, M. & Straus, D. (1976) *How to Make Meetings Work*, Jove Books; Freedman, D.P. & Weinberg, G.M. (1982) *Handbook of Walkthroughs, Inspections, and Technical Reviews*, Third Edition, Little, Brown & Co.; Michalko, M. (1991) *Thinkertoys: A Handbook of Business Creativity for the 90s*, Ten Speed Press.

This list might have as few as 20 or as many as 60 areas of work.

It's easy to run into diminishing returns while generating this list. Stop the session as you start running out of steam, but leave the list open so that anyone can add new topics over the next few days, as they come up with new ideas that they think are worthwhile.

Day 2 / 3 – Transcribe the Overall List Onto Separate Charts; Add Sub-Tasks

Make one chart page for each significant testing area. The test lead[6] should eliminate or combine obviously redundant or overlapping areas. From here, add detail, breaking each area down into tasks and sub-tasks. List anything that will require at least a half-day of work.[7]

If there are many charts, it won't be possible for everyone to work on every chart. Break into sub-groups to save time, but try to have at least two people generating ideas for each chart. If a member of your staff has volunteered to take one of the areas, she should be one of the people who details the chart for that area.

Try to assign an estimated amount of time to each task. You're guessing here. Your guess includes:

- the amount of time to plan and create the specific tests within this task;
- the amount of time to run these tests the first time;
- the amount of time to report bugs within this task area, and to regression test the fixes or follow-up on the deferrals;
- the amount of time to regression test or analyze bugs in this area that were reported by other people;
- any additional unstructured testing in this area;
- ongoing regression testing within this area.[8] The amount of ongoing regression testing increases with every additional cycle of testing. Implicitly, then, you are also guessing the number of testing cycles.

Analytically, this estimation process might seem ridiculous because we are guessing with such a degree of uncertainty. It isn't ridiculous, though, because we are working backwards as well as forward. As long as we operate from the same base (such as, all estimates assume 8 full start-to-finish cycles of testing), we obtain relative estimates of the different areas that are quite useful.

Different testers will supply different estimates for these tasks. When you see a large discrepancy, don't browbeat the tester who gave the large estimate. Probe the estimate instead. Try to break the task into sub-tasks – are these redundant with other areas or is this tester seeing more genuine complication and work than the other estimators?

Day 3 / 4 / etc. – Use a Museum Tour to Check the Estimates

Leave the charts up. If they don't all fit within the conference room, take over some other wall space or outside-of-cubicle space near the conference room and put the overflowing charts there.[9]

---

[6] I'll assume in this paper that you are acting as the test lead, and will often say that "you" should do certain tasks that a test lead should do. When I want to specifically stress that a task belongs with a test lead (rather than, for example, the lead's manager or the testing team), I'll still say "test lead."

[7] My goal is to capture everything that will take a day or longer, but testers are such notoriously optimistic underestimators that the only way to do this is to set the bar at half a day.

[8] Unfortunately, "regression testing" has two meanings. Sometimes you test against a specific bug report, to see whether that particular bug was fixed. Other times, you test everything else, to see whether the latest waves of fixes has had any side effects. By "ongoing regression testing," I intend the second meaning.

Send a memo to everyone on the project team. Invite them to drop by at their convenience over the next few days, to look at the task lists, to add tasks, and to supply time estimates or notes on priorities or risks. Have a curator handy. Most testers who have participated to this point can do this -- the curator provides a tour of the charts and answer questions about the process.

Give everyone who comes by a different colored pen, and keep track of who used which color. Have them write their notes directly on the charts.

Follow up on time estimates that are noticeably higher or lower than your group's estimates. For example:

- You might think that a feature is trivial and easy to test. The marketing manager might say that this is the most advertisable feature in the product and he wants to make absolutely sure that it works perfectly. This might lead you to you increase the time you spend testing this feature.
- The marketing manager might also point to a complex feature and ask why you're spending so much time on something that will matter so little to customers. Maybe you will spend less testing time on this as a result, perhaps by convincing the project manager to simplify the feature's design, thereby reducing the need for testing.
- The project manager, or a programmer, or a customer service representative might explain to you that you have underestimated the difficulties you are likely to encounter with a feature, providing details that help you add more sub-tasks and re-estimate the work.

You might leave these charts up for a week. Your staff will mainly be doing other work, not fleshing out the charts, during this week.

At the end of this process, you have an extensive list of areas to test, and tasks and subtasks within each estimate, along with notes on testing times.

ASSESSING THE TIMES INVOLVED

As the project's lead tester, you will collect the charts and enter the areas, tasks, and subtasks into a project or task management program. Eliminate or merge redundant or overlapping tasks.[10] Resolve discrepant time estimates for each task by using your best judgment.

The total time requirement will be impossibly large. On a project that currently allows you three tester-years of work, you might see fifty or sixty tester-years of work on this list. The test lead has to cut this down, by prioritizing the tasks and deciding which areas to under-test.

I find it useful to talk in terms of levels of testing depth. In the black box world, I often use four levels, which I call mainstream, guerrilla, structured, and systematic regression. These particular levels and definitions are much less important than the notion of using a small number of well-defined levels:

- ***Mainstream-level testing***: These are relatively gentle tests of the program's performance under "normal" use. I include line-by-line verification of the manual and help against the program within this category (though I normally break out and budget the documentation verification as a separate category).
- ***Guerrilla-level testing***: Along with doing the mainstream-level tests, during one or two cycles of testing, add 2-10 hours of unstructured testing. This consists of the nastiest test cases that the tester can think of, executed "on the fly." There is no archival test plan. The tests

---

[9] If your charts cover over light-colored walls or peoples' posters, make sure to put a blank backing sheet behind every chart. Otherwise, somebody's marker will leak through the first sheet and deface the wall or the poster. You don't need this problem.

[10] Many redundancies don't become evident until the tasks are listed in detail. Also, many overlapping areas were left on the charts because this allowed the staff to explore the extent of the overlap and the degree to which they are separate areas. Now it's time to clean this up.

probably include boundaries, error handling, challenging-looking interactions with other features, etc. Anything that the tester thinks is likely to expose a bug is fair game.

- ***Structured, planned testing***: Tests are based on a more rigorous analysis. For example, boundary charts are created for the variables in this area. Error handling is approached systematically. Requirements are approached systematically, etc.

  Typically, an area that is subjected to this intense level of testing has also had mainstream and guerrilla-level testing. Those levels quickly expose obvious instabilities and design failures. The cycle or two of intense, structured testing of an area is done after the obvious problems have been resolved.

- ***Systematic regression testing:*** Along with developing a structured test set, the tester develops a strategy for sampling from the set, or for adding new test cases, to monitor the stability of this area over time.

  In the previous case, the structured test series was executed once or twice during development of the product. Regression testing at other times is neither thorough nor systematic. In this case, we develop a strategy for checking, on an ongoing basis, that an area that achieved stability has not been destabilized by later code changes.

  If you develop automated regression tests for an area of the program, you are operating at this level for that area.

Other definitions of levels might look at the extent and type of coverage achieved, or at the extent of automation or other tool-based support, or at some other salient, important characteristic of the testing effort. The important thing is that these definitions be clear, understandable to a non-programmer, and demark clearly different levels of effort and investment.

Suppose, for the moment, that you adopt these four definitions. Then for every area / task in the program, decide what level of testing you would recommend. If you've reduced the level for an area from structured, planned testing, then you can probably reduce your time estimate for that area. If you stretch to systematic regression, you'll probably increase your testing time estimate.

Make a total of the initial time estimates. Make a total of the revised estimates.

With your revisions, you have probably come from 60 tester-years of work down to 5 years. This is an excellent reduction, even though it still isn't quite good enough.

RUNNING THE NEGOTIATING MEETINGS

After you have your reduced estimate, send out a memo to the other key members / managers of the project team (project manager, marketing manager, customer service, documentation, etc.).

> To: Project Team
>
> From: Test Lead
>
> Many thanks for helping me put together the testing tasks list. Not surprisingly, when we add up all the time for all the tasks, it works out to *60 tester years*. Obviously, we can't afford to do all that work.
>
> I've prioritized the tasks and worked out ways to save time. This involves some shortcuts that you may want to think about. I got the workload down to 5 tester years – a big improvement, but still not enough. The budget calls for 3 tester years, not 5.
>
> I'd really appreciate it if you could help me trim the tasks further. We'll have to take some shortcuts, and this will probably mean that we'll

> miss a few bugs. I'd like your feedback on the best places to take a few risks.
>
> <insert meeting-scheduling-stuff and closing-thank-you>

Rather than the usual plea for more time and resources, you're inviting everyone to help you bring the testing costs down. Who can refuse such an invitation?

Once you've scheduled the meeting, circulate the task list.

- The front page is a cover memo. This is probably where you define your levels of testing and explain that your are trimming testing time by testing some areas less than others.
- The second page has four columns – *Area of Testing* (show only the top-level areas), *Initial Time Estimate* (show them, and total them to 60 years), *Second Test Group Estimate* (show them, and their 5 year total), and *Project Team Estimate* (add these in this meeting).
- Subsequent pages break down each area, as on the flipcharts. Show the initial estimates for each task, along with your suggestions and the project team's final estimates.

Explain the ground rule for trimming time: to reduce the total amount of time to be spent on one area, the team must reduce the amount of time on specific, identified tasks within that area.

Settle on a Baseline

The first agreement to reach is that there should be a baseline. The baseline is the minimum level of testing that all areas of the program are subjected to. This general agreement shouldn't be difficult to reach. The interesting discussion comes next, when you try to define the baseline.

One candidate for the baseline is mainstream-level testing. If you achieve this much, at least you know that all the menus take you to reasonable places, that the dialogs accept data, that the program prints when it's supposed to, and that everything in the manual has been checked.[11]

Explain how limited this level of testing is, with examples of bugs that wouldn't have been found with testing at this level.

Some companies will embrace mainstream testing as their baseline, recognizing that they will adopt more thorough approaches for more risk-intensive tasks (such as disk I/O and disk-related error handling). Others will demand more thorough testing of everything. Rather than being the ardent proponent of a higher level, be a teacher in this meeting. Let the battle for a more intense testing baseline be led by the team member from customer service, or marketing, or documentation.

If the group agrees on a higher minimum, you'll have to revise your estimates for any task that you had planned to do at a mainstream level.

Review Your Proposals for Limiting Testing

Highlight your riskier cuts and ask whether the other members of the team think that *more* testing than you propose is needed in these areas. Make sure that your proposals are subject to review by the team.

Don't spend most of the meeting explaining and reviewing your own proposals instead of bringing out new ones from other people. Do summarize the tradeoffs and risks clearly enough so that no one can say later that they were surprised by how little testing you were doing in an area or what risks you were recommending that the company accept.

---

[11] I am sidestepping the issue of the level of glass box testing (code-driven testing such as unit testing or testing designed to reach a line coverage criterion). If the testing group does this work, incorporate it into this meeting and include it in the task analysis. If the testing group doesn't do this work, I would not normally include it in this meeting.

Generate Additional Proposals

This meeting is a problem-solving session, to generate ways for you to do less testing.

Of course, if the group can't find anything to cut, or runs out of things to cut, and you still don't have enough people to time to do what's left, you have a very strong argument (and probably some allies) for getting more staff and/or more time.

Many suggestions will be unpleasant for you to accept. The other members of the team have different senses of what risks should be managed, and to what degree. When someone proposes that a task be cut or reduced in level, do explain what risks are involved and (if you disagree with the suggestion) why you think it's a bad idea. But treat the proposal with respect. And accept that some of these proposals will be acceptable to the rest of the team, in just the same way that many bug deferrals seem more acceptable to everyone else than to you.

Don't dominate these discussions. Let the other stakeholders defend the product and the level of testing involved. For example, if you think that skimping on configuration testing will result in a large increase in customer service call volume, ask the customer service representative to comment on it.

Sometimes these sessions will yield new tasks or upgraded priorities for tasks. This is an excellent way to discover these requirements, because your additional efforts are being demanded by the rest of the team. You're in a strong position to get the resources you'll neet to satisfy them.

Draw Realistic Lines

On a tight schedule, you'll probably be faced with several pressures. I can't coach you, in this paper, on how to negotiate these issues,[12] but here are some notes on the issues themselves:

- You'll feel pressure to do a higher level of work in less time. If your estimates are reasonable,[13] then you can't do this.
- Don't build expectations of (unpaid) overtime into your scheduling. Testers work overtime voluntarily, to make up for lost time, recover from mistakes, or add creativity or depth to their work in a way that goes beyond the scheduled requirements, in order to meet their own professional standards. This is important flexibility, for them and for the project. Don't give it up.
- Allow time for vacations, sickness, and holidays on a reasonably long project. Don't agree to magically make these go away or to miraculously make up for them.
- Allow time for administration, staff development, and other non-testing tasks. You probably listed many of these on the flipcharts. Don't agree to pretend that people don't attend meetings, spend time on reviews, help people on other projects, etc. Stick with realistic estimates of this overhead.
- If your efficiency in a task depends on someone else meeting a milestone and delivering something to you (code, documentation, requirements, equipment, etc.) by a certain date, make that dependency known and make the lost-time consequence of failure to meet the milestone understood.

You can agree to save time by doing less testing, or (within limits) to save calendar time by adding more staff, or to work creatively to achieve some efficiencies. But don't agree to a schedule that you can't meet or to a schedule that you can only meet by hiding the fact that you are doing less testing.

---

[12] Freund, J.C. (1992) *Smart Negotiating: How to Make Good Deals in the Real World*, Simon & Schuster is an excellent source of negotiating advice. Also quite helpful: Fisher, R., Ury, W., & Patton, B. (1991, 2nd Ed.) *Getting to Yes: Negotiating Agreement Without Giving In*, Penguin; and Fisher, R. & Ertel, D. (1995) Getting Ready to Negotiate: The Getting to Yes Workbook, Penguin.

[13] I assume that you don't have some ready means of improving your efficiency. If you do, then of course, do it.

Follow-up Meetings

You might not reach agreement on all points in the first meeting. You might run out of time. You might run into so many suggestions that you need a break to re-estimate the project. The other project team members might want time to gather data. It makes good sense to break this discussion across two meetings. More than two meetings might be too many.

RISKS

I mentioned that this approach doesn't always work well. Here are some key problems:

- The test lead may be too timid in negotiating meetings.
- The test lead may be too overbearing in meetings with her staff or with the rest of the project team. This can break up the brainstorming, lead to unrealistic task estimates, or stifle discussion in the team meetings.
- Beware of endless listing of tasks. On a complex project, don't have every tester work on every flipchart -- run things in parallel. And do the listing and estimation with a sense of urgency -- set objectives each day, and if necessary, stay late to meet them.
- Unrealistic project management is sometimes deliberate, at the project manager's level or well above it. If the rest of the project can't finish on time, you won't be able to reach agreement on a realistic plan to finish testing the product on time.

In some companies, the only people who *are* concerned about quality are the testing and customer service staff. The project manager, the marketing manager, and their management, are glad to sign off on an absurdly shallow level of testing of the product. This can go to extremes: in some companies, people deliberately ship software that they know will erase or corrupt a non-trivial percentage of customers' disks or data.

You learn who you're working with during these negotiating meetings. Rather than throwing yourself in front of their train, if you really are working for quality-less fools, get your resume on the street and work on something worthwhile.

REWARDS

If the process is a complete success, you have an almost-complete[14] list of tasks, an agreed-on set of time estimates, the resources you need, and a lot of understanding and support from the rest of the team.

If the process is less successful, you probably still get an almost-complete list of the testing tasks, and some useful estimates of the relative times needed to test different areas of the program at different depths of testing. You have a strong basis for managing the black box testing of the product, and you developed this material in a matter of days.

During this process, you probably will discover that most people care intensely about the quality of the products that they build and they will accept accountability for quality. They may have dramatically different ideas of what makes a product have high quality. This process allows those differences to surface in useful ways.

If the product must be developed in a limited time, on a limited budget, testing will be more limited than you would like no matter what process you use. This approach has the advantage of making the issue explicit, and driving the company to make explicit business decisions about the quality-related risks that it will take in developing and releasing the product.

[14] There will always be surprises. Budget for them.

