Communicating With Non-Testers

Catherine Powell
Abakas, LLC
291 Shawmut Ave #1
Boston, MA 02118
+1-617-901-1526
catherine@abakas.com

ABSTRACT

In this paper, I provide an experience report describing how we select forms and methods of communication. To ensure clear and effective communication, we identify the form and content based on a set of guiding questions we have developed

Categories and Subject Descriptors
K.6.1 [**Project and People Management**]: Management Techniques – *life cycle, and training.*

General Terms
Management, Documentation, Human Factors, Theory

Keywords
Semantics, Communication, Writing,

1.INTRODUCTION

Testing is a discipline that deals in information. What the system does, what it doesn't do, risks of the system, etc. - all this information helps teams make good decisions. Having information, though, is useless unless that information can be conveyed in a clear and effective way.

Communicating with non-testers requires a lot more thought and a lot more explanation than conveying information to testers who share your background. Management wanting status reports, release teams asking about when something will ship, developers and semi-technical customers wanting analysis of a defect or risk: all these are non-testers who need information presented in a way that lets them make good decisions. Communicating test concepts and outcomes to non-testers is a skill that turns a good tester into an invaluable team member.

This experience report describes some communication techniques and the reasons behind using those technique using real life examples.

2.GUIDING QUESTIONS

We provide information to non-testers in the form of communications – conversations, emails, bug reports, etc. To do this effectively, we have identified a list of questions that help us decide how to communicate a set of information. Consider this a checklist for communications development:

- ☒ Who will see this?
- ☒ What will the recipient get from this?
- ☒ How can I get this information transmitted to the recipient?
- ☒ What contextual information do we share? Not share?
- ☒ How do I differentiate certainty from uncertainty?

The following sections apply these questions to several examples.

3.EXAMPLE: PROJECT DASHBOARD

3.1Who will see this?

We get asked for project status in different ways: "Where are we?" "What is the status of testing?" "Can we ship yet?" "What's the status of the widget project?" So who is asking?

Engineering. Testers and developers on the project team keep track of how a project is going. We check the status of the project at least one an iteration and generally much more frequently.

Management. In our company, the release team includes product management, program management, marketing, and upper-level engineering management. They want to see the status of testing to make an informed release decision. The release team meets weekly and checks status for each meeting.

3.2What will the recipient get from this?

Although we have multiple recipients, they each want the same thing: to know what's going on with the project. However, when I asked for a bit more detail, I got different answers:

From engineering: "To know what to test next, and what bugs we should fix first to unblock testing."

From management: "To understand what stories have been done, what hasn't been done and what regression testing is finished."

These answers imply very different levels of detail. Engineering is working at a very granular level – bugs and tasks – while management is looking at a higher level – features and stories. It's the same underlying information, but we have to present it in one way (bugs and tasks) for engineering and differently (features) for management.

3.3 How can I get this information out?

Our users look at status weekly or more often. We decided that meant we didn't want to spend a lot of time generating any report, much less two reports. How can we convey this information?

- Email
  - Pros: easy to get information to those who need it
  - Cons: Time intensive to put together; difficult to search
- Conversation
  - Pros: instant feedback
  - Cons: with so many people who need status, have to have the conversation multiple times; human memory is less than ideal
- Wiki Page
  - Pros: common repository for information already
  - Cons: Has to be hand-updated from Jira, where the stories (features) and tickets are
- Jira Dashboard
  - Pros: this is where we keep the data already
  - Cons: to see it, user have to be online

We already use Jira to track all the information we need for status report, so using a Jira dashboard makes a lot of sense. It also lets people get up to date information as frequently as they want.

3.4 What context do we share? Not share?

Since the entire audience is internal, we share a lot of context. We all know when the release is scheduled; we know what's in the roadmap and can look at the same internal development documents, designs, requirements documents, etc. Since all that information is accessible to the audience, we don't need to include that in our project dashboard.

The context we don't share is the state of the software, our testing, and the release. This is why we need a dashboard: to share status context.

3.5 Differentiating certainty from uncertainty

Fact is what we know for sure. Opinion is what we believe, guess, or deduce will happen. In the case of project status, what we've done is fact. This is not a place for opinion, just an accounting of what we know and what we don't.

3.6 What it looks like

Because we have two groups of people looking at the dashboard, we have two dashboards, one for engineering that is bug and task-oriented, and one for management that's more feature and story-oriented.

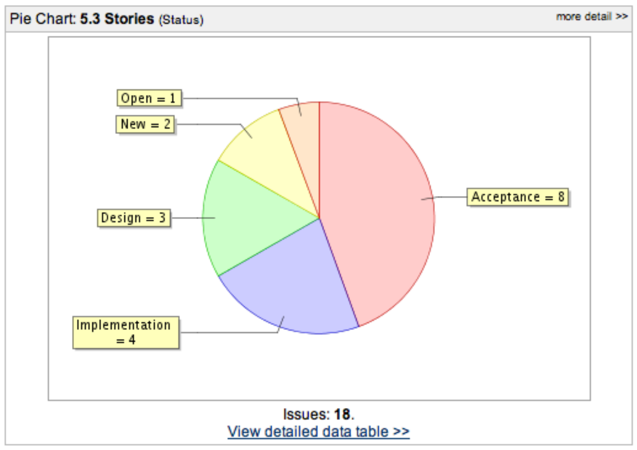
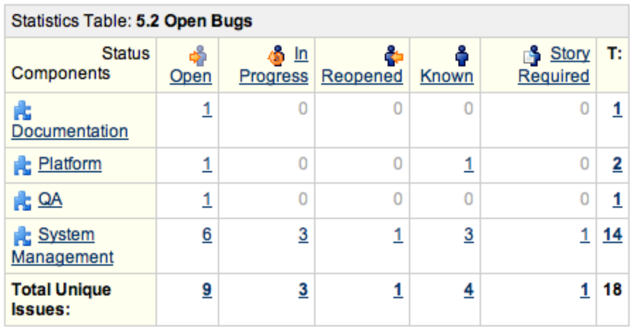


Figure 1 Management Dashboard

The management dashboard is a high-level summary that shows the number of items in each development phase. Clicking on any one gives a list of the stories or tests that are in that status.



Statistics Table: **5.2 Open Bugs**

Status / Components	Open	In Progress	Reopened	Known	Story Required	T:
Documentation	1	0	0	0	0	1
Platform	1	0	0	1	0	2
QA	1	0	0	0	0	1
System Management	6	3	1	3	1	14
Total Unique Issues:	9	3	1	4	1	18

Figure 2 Development Dashboard

The development dashboard shows tasks and bugs, broken down by both team and status.

With both dashboards, we're trying to show a broad overview without a lot of detail. The development dashboard includes more information about which team is working on something, and about smaller tasks, while the management dashboard shows less granular tasks and is concerned about state rather than the inner workings of the engineering team.

4. EXAMPLE: DEFECT SUMMARY

4.1 Who will see this?

Most of the defects we write are for internal use only. We find a bug, write a defect, development fixes it, and testers verify it. The bug never sees the outside world, and so we don't have to explain it to anyone outside the engineering team.

Some defects, however, are found in the field. Once we understand them, we have to explain the bug to the customer. If it's a bad problem or a highly visible problem, the customer sometimes forwards the defect summary to their customer or to their management.

4.2 What will the recipient get from this?

Like the project dashboard, the different recipients will get different things out of a defect summary. We looked at the last two years of defect reports, which ones were reopened or modified, and which ones resulted in resolving the problem (including making the customer happy). Successful defect explanations provide:

From engineering: enough information to fix the bug and to write a test that shows the extent of the problem. Good information includes stack traces, specific build information, a test showing the problem (manual or in code), and expected behavior.

From the customer: an explanation of the defect, including how it will be fixed, in language the customer can use in explanations internally. Useful summaries describe the severity and rarity of the defect, what workarounds or preventive measures are available, when it will be fixed, and a summary of the problem using terms the customer already understood and could explain.

4.3How can I get this information out?

A defect summary is used once; it gets written, referenced until the issue is resolved, and then put away. For engineering, we use Jira, because it then gets into the same task list the engineers use for new development and the defect becomes just another thing to do.

When we summarize a defect for a customer, we need to provide it to them in a format they can forward to their customers or managers as needed. Email works well for this, since forwarding is simple. For defects that involve pictures or diagrams, we use PDF documents so the images don't get stripped out of email. PDF has two major features we like: (1) it's easy to forward; and (2) it doesn't change – content or formatting – as it is transferred around.

4.4What context do we share? Not share?

This is where our audiences diverge. The test team and the development team share a lot of context and access to the same additional information. The customer has less context in terms of the internal workings of the system, but has context in terms of how they have experienced the problem and its effects.

Our engineering team shares most of the context the test team writing the defect summary has: knowledge of the design of the system, module names, code availability and usually desired behavior are all the same. The defect summary needs to convey the remaining context the helps the recipient resolve the issue: what the bug is, how likely it is, and where it was found.

With the customer, the shared context is the consequence and severity of the defect, and the externally-visible workings of the system to some level of detail. The system design, source code, and internal workings of the system are not available to the customer, so we can't use them in the defect summary without causing confusion. For example, we had a bug with mapping from Windows ACLs to POSIX ACLs, and the customer didn't know what POSIX ACLs were, since he had never worked with a non-Windows system before. Our defect summary had to include a definiton of POSIX ACLs versus Windows ACLs as background information for him to understand the problem.

4.5Differentiating certainty from uncertainty

A defect summary contains both certainty and uncertainty. The effect of the defect is usually certain because it's the first thing we see – system crashed, or a field accepted an invalid value, or an error displayed unexpectedly. Often we're also certain of the steps required to reproduce the error, whether it's a 100% reproduction or a race condition that we can trigger 25% of the time by running a certain automated test.

Usually, we are less certain of how frequently a problem will recur. We can make guiding statements – every time file type X is written to the system – but can't necessarily provide a timeline. So we make statements about the uncertainty, and describe frequency with more ambiguous terms like "rare" or "frequent".

When we have both certainty and uncertainty in the same defect, we specifically note what we are sure of and what we are less sure of. When it's not clear, we get repeated questions like, "do these steps make it happen every time?" Looking at our past defect summaries, fewer defects spawned requests for clarification when we used phrases like "we know" or "every time" for certainty and phrases like "25% of the time" and "we think" for uncertainty.

4.6What it looks like

This example shows the same defect, written once for development and once for the customer. For this particular defect, the problem was that the customer was using a Windows system to administer a share on a system that supported POSIX ACLs and was getting unexpected behavior when trying to use extended permissions (unsupported by POSIX ACLs). The resolution was to confirm in engineering that there wasn't an underlying problem, and to educate the customer.

Take the user from Use Case 1 and change their permissions by: selecting or deselecting the new permission, selecting the "propagate" option on the advanced tab, and clicking Apply.

When you ADD WRITE permissions, you get:

- on all folders, the user is still present and still has "Special Permissions"
- on child files, the user is still present and has all checkboxes checked, including "Full Control"

When you REMOVE WRITE permissions, you get:

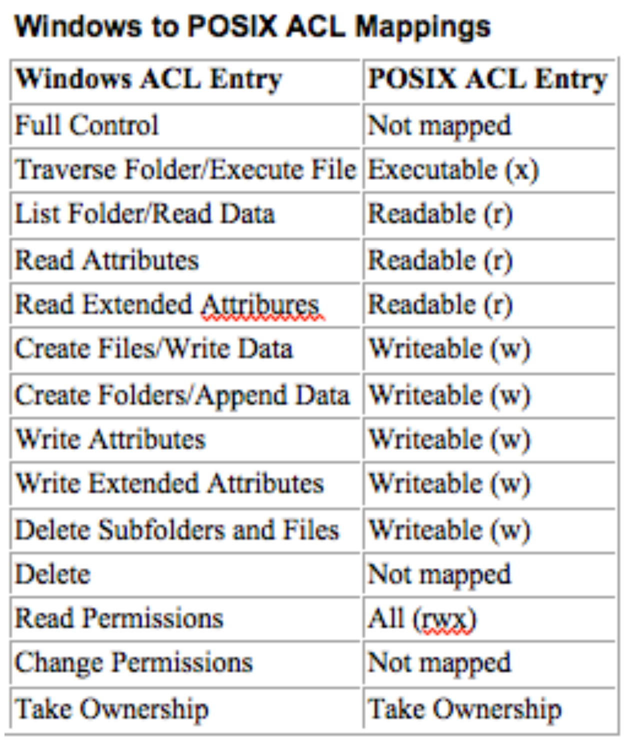
- the same state you have when you first create the user with default read-only permissions. (correct behavior)

When you ADD ADVANCED permissions, you get:

- on all folders, the user is still present and still has "Special Permissions"
- on child files, the user still has the PREVIOUS settings. This means the permissions have not propagated. This only occurs when you set an extended permission (e.g., take ownership

Figure 3 Engineering Defect Summary

For the developers, the defect summary describes what the customer sees and references logs. In this case, the customer was very Windows-oriented but had little familiarity with how the Windows ACLs he worked with mapped to POSIX ACLs.



Windows to POSIX ACL Mappings

Windows ACL Entry	POSIX ACL Entry
Full Control	Not mapped
Traverse Folder/Execute File	Executable (x)
List Folder/Read Data	Readable (r)
Read Attributes	Readable (r)
Read Extended Attribures	Readable (r)
Create Files/Write Data	Writeable (w)
Create Folders/Append Data	Writeable (w)
Write Attributes	Writeable (w)
Write Extended Attributes	Writeable (w)
Delete Subfolders and Files	Writeable (w)
Delete	Not mapped
Read Permissions	All (rwx)
Change Permissions	Not mapped
Take Ownership	Take Ownership

Figure 4 Customer Defect Summary

The main portion of the defect summary for the customer was a display of how the Windows ACLs - with which he is familiar – display when he looks at them on the system that supports the POSIX ACLs. In this case, the mappings are definitive, so no uncertainty terms are used to indicate probability rather then certainty. The defect summary further showed in detail, with screenshots, how to successfully propagate permissions on the POSIX ACL system and what displays in each case.

The developer-oriented defect summary references internal tools – "Use Case 1" is in the test plan – that are not mentioned in the customer defect summary. The internal defect also goes into more detail about exactly what steps are performed just in case there's a non-obvious problem. By the time the customer defect summary was written, we knew there wasn't an underlying bug, so the focus was on explanation for a Windows-oriented customer.

5.EXAMPLE: TEST PLAN

5.1Who will see this?

The test plan describes what we test and how we do those tests. For some of our projects, the only people who view the test plan are the testers themselves. For others projects we do, the customer's engineering team wants to see the test plan to use it as a basis for their own testing, or to use to evaluate our software and our development process.

5.2What will the recipient get from this?

Like the project dashboard and defect summary, each group that reads the test plan will get something different out of that reading. Two different groups are looking for different things from the test plan:

Internal engineering: uses the test plan as a reference for status and for defects. Most of the time, they want to see a section of the test plan or a single tested case so they can get more detail on something that was summarized in a defect or a status.

Customer engineering: uses the test plan to evaluate how we test software and to look for holes that they need to test. Some customers also use the test plan as a guideline for their own testing, as a source of ideas. Our customers use phrases like "test philosophy", "test coverage", and "test design" when we ask what they are seeking when they ask for a test plan.

5.3How can I get this information out?

We do a test plan for each release, generally basing it on the test plan for the prior release. For engineering and for our own use, we divide the test plan into small manageable chunks. This breaks the product down by feature and test type, and provides ideas for testing. As testing continues, testers fill in the test plan with session notes, test cases, and references to bugs. Over time, it resembles a brain dump of test information, with links to useful information, defects, test setup information and other tests. We keep this in a wiki because it is so linked, and easily updated. The wiki is available to everyone in engineering.

When we provide a test plan for a customer, we provide it as part of the software delivery. In a few cases, the test plan has been a referenced document in statements of work. Both of these requirements mean that the test plan must be a single document. We use a PDF document for delivery, since it can be read as a unit and is easily transferable to clients. We deliver test plans two ways to clients: (1) as a standalone document emailed or downloaded from a website; or (2) as part of the binary software deliverable (a .tar.gz).

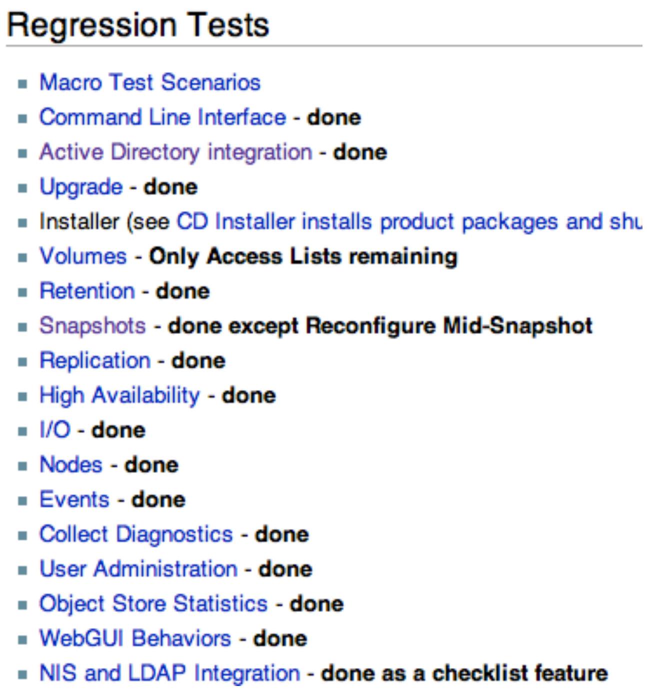
5.4What context do we share? Not share?

As with the other documents, we share context with the engineering team, including development and test strategy, an understanding of the system and its constituent parts, and availability of resources (e.g., the wiki or test configuration). We're all on the same team developing software the same way, so most context is shared.

Also as with the other documents, we share less context with customers, so we have to put more of the test context in the test plan. If we say to a customer, for example, that we do "sample program tests", that customer doesn't know what we do to those sample programs. Internally, that phrase is sufficient, but without the context that "test" means "perform exploratory tests of a component, with emphasis on deployment and supported and unsupported uses of that program", the customer can't evaluate whether our tests of that sample program are sufficient or interesting. Because of this lack of context, we define in our test plan what an exploratory test is, and we provide additional information about what we test in addition to how we test. Our goal is to minimize the number of times a customer reading our test plan has to ask a clarifying question.

5.5What it looks like

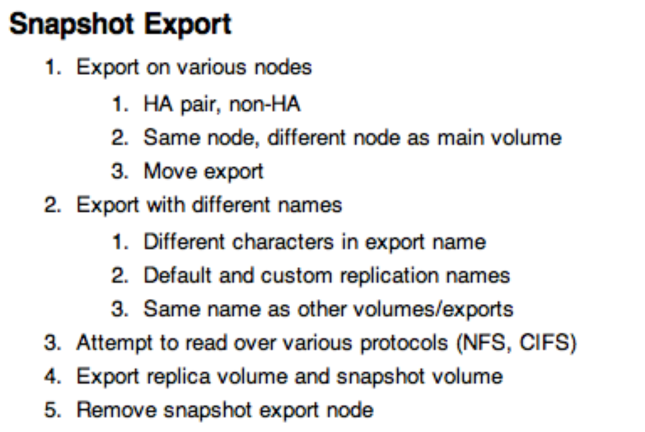
This example shows a portion of a test plan for internal use by the engineering team. Notice all the links, which lead to wiki pages that describe detailed tests performed, test setup, to bugs, and to other parts of the test plan.


Regression Tests

- Macro Test Scenarios
- Command Line Interface - **done**
- Active Directory integration - **done**
- Upgrade - **done**
- Installer (see CD Installer installs product packages and shu
- Volumes - **Only Access Lists remaining**
- Retention - **done**
- Snapshots - **done except Reconfigure Mid-Snapshot**
- Replication - **done**
- High Availability - **done**
- I/O - **done**
- Nodes - **done**
- Events - **done**
- Collect Diagnostics - **done**
- User Administration - **done**
- Object Store Statistics - **done**
- WebGUI Behaviors - **done**
- NIS and LDAP Integration - **done as a checklist feature**

Figure 5 Engineering Test Plan (Summary)

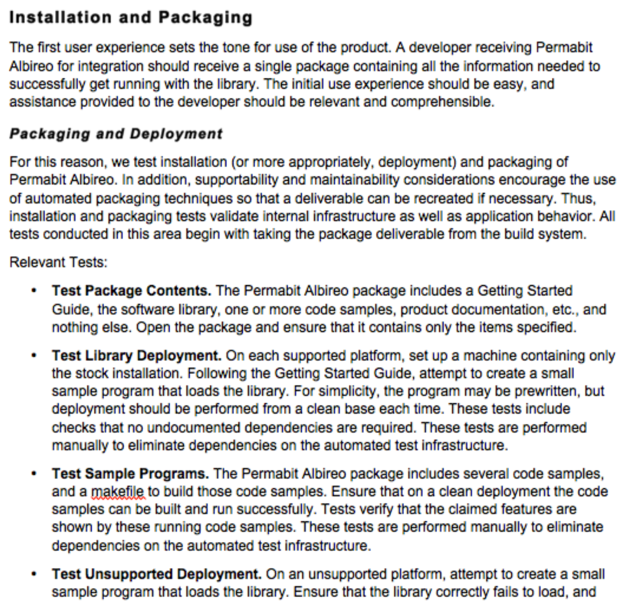
For the developers, the test plan summary provides an entrance to the parts of the test plan they're interested in. From there they can dive into more details as tests are performed, including checklists and guides that provide ideas for tests, and the results and notes from tests performed.


Snapshot Export

1. Export on various nodes
    1. HA pair, non-HA
    2. Same node, different node as main volume
    3. Move export
2. Export with different names
    1. Different characters in export name
    2. Default and custom replication names
    3. Same name as other volumes/exports
3. Attempt to read over various protocols (NFS, CIFS)
4. Export replica volume and snapshot volume
5. Remove snapshot export node

Figure 6 Engineering Test Plan (Checklist)

The customer receives a document that uses more formal language and conveys more information.


Installation and Packaging

The first user experience sets the tone for use of the product. A developer receiving Permabit Albireo for integration should receive a single package containing all the information needed to successfully get running with the library. The initial use experience should be easy, and assistance provided to the developer should be relevant and comprehensible.

Packaging and Deployment

For this reason, we test installation (or more appropriately, deployment) and packaging of Permabit Albireo. In addition, supportability and maintainability considerations encourage the use of automated packaging techniques so that a deliverable can be recreated if necessary. Thus, installation and packaging tests validate internal infrastructure as well as application behavior. All tests conducted in this area begin with taking the package deliverable from the build system.

Relevant Tests:

- **Test Package Contents.** The Permabit Albireo package includes a Getting Started Guide, the software library, one or more code samples, product documentation, etc., and nothing else. Open the package and ensure that it contains only the items specified.
- **Test Library Deployment.** On each supported platform, set up a machine containing only the stock installation. Following the Getting Started Guide, attempt to create a small sample program that loads the library. For simplicity, the program may be prewritten, but deployment should be performed from a clean base each time. These tests include checks that no undocumented dependencies are required. These tests are performed manually to eliminate dependencies on the automated test infrastructure.
- **Test Sample Programs.** The Permabit Albireo package includes several code samples, and a makefile to build those code samples. Ensure that on a clean deployment the code samples can be built and run successfully. Tests verify that the claimed features are shown by these running code samples. These tests are performed manually to eliminate dependencies on the automated test infrastructure.
- **Test Unsupported Deployment.** On an unsupported platform, attempt to create a small sample program that loads the library. Ensure that the library correctly fails to load, and

Figure 7 Customer Test Plan

The test plan describes the feature inline rather than by reference, since a reference is not usually available. It describes each test type, why it is performed, and how it is performed. The terms used here (e.g., "automated test infrastructure") are described earlier in the document. This means the customer can read the document, use it as a reference, and pass it along without losing information.

6.CONCLUSION

Testing is an information-oriented discipline. A successful test has two parts: the targeted gathering of information, and the conveyance of that information to others. The prior examples have shown different methods of conveying information to various audiences, and how that information exchange varies based on the audience receiving the information. Communicating with testers and with non-testers lets you provide valuable information and make your tests successful, but only if you communicate the information in a way the audience can understand and use that information.