Presentation Notes
Paper
Bio
Return to Main Menu

PRESENTATION

F6

Friday, November 5, 1999
11:15 AM

Divide and Conquer (Making Sense Out of Test Planning)

Kathy Iberle
Hewlett-Packard

International Conference On
Software Testing, Analysis & Review
November 1-5, 1999
San Jose, CA

Divide and Conquer:
Making Sense of Test Planning

By: Kathy Iberle

Hewlett-Packard

STAR WEST ‘99

Creating your first test plan...

- Where to start
- How to divide the testing into groups
- What to do next
- Scheduling the testing

A Little Terminology

- A *software feature* is “a distinguishing characteristic of a software item (for example, performance, portability, or functionality).”
- A *test plan* “describes the scope and approach of intended testing activities. It identifies….the features to be tested.”
- A *test design* “specifies the details of the test approach for a software feature and identifies the associated tests.”

IEEE 829

- The IEEE 829 Standard says that a test plan shall include:
  - **A list of features and combinations of features to be tested.**
  - **A general statement of approach for each feature or combination of features**
  - **Identification of the test design associated with each of those features and combinations.**

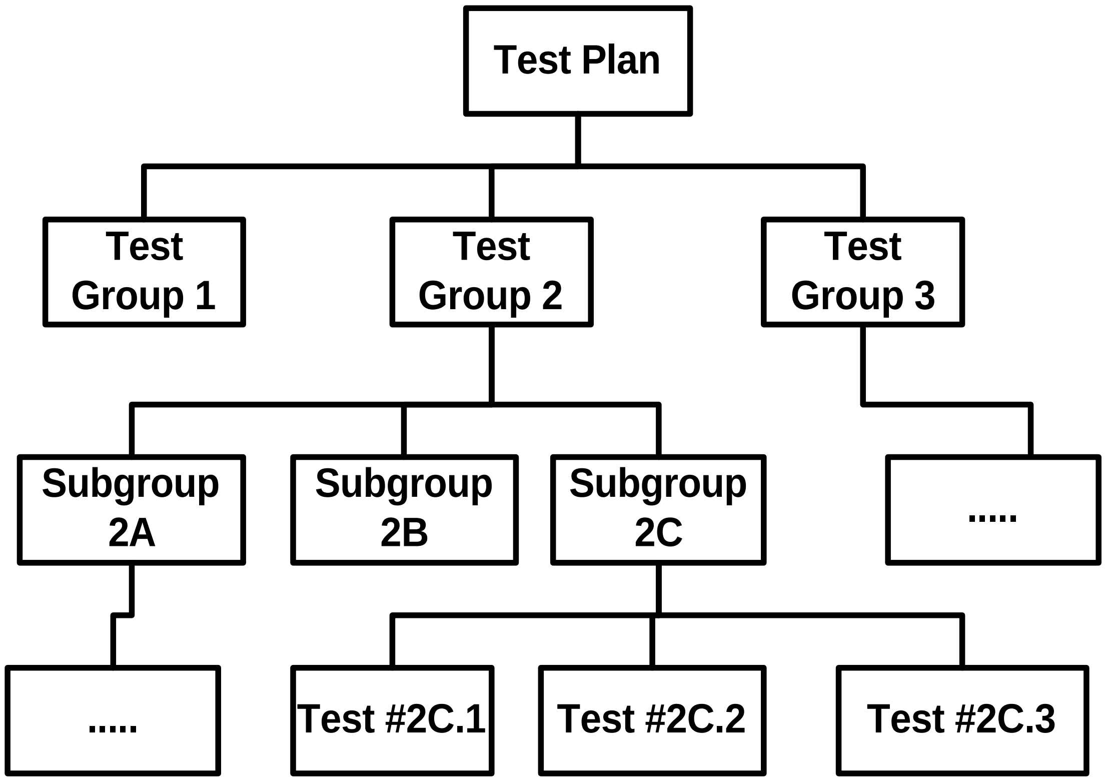
Key Learning #1:

The “Features To Be Tested” section
is supposed to cover
all of the system behavior,
not just the functionality!

Beizer’s Test Hierarchy

- A *test case* is a single set of inputs, actions, and expected outputs.
- A *test* is a set of test cases that share the same setup and execution environment and cover closely related functional areas.
- A *test subgroup* is a group of tests that can be designed and executed independently of any other subgroup.
- A *test group* is the “broadest functional subdivision of the system”.

Beizer's Test Hierarchy



The “Divide and Conquer” Method

- Start the set of test groups with the list of functionality.
- Each function becomes a test group.
- Each system test type (stress, reliability, performance, configuration, …) becomes a test group (except functionality)

Key Learning #2:

Half of the “Features to be Tested” are the functions of the system.

The other half are the system test types!

Example: Inkjet Printer

- Printing
- Paper sizes
- Collation & multiple copies
- Installation
- Paper out, paper jam, etc.
- Cancel/Pause
- Online Help
- Reliability & Stress
- Performance
- Compatibility
- Configuration
- Localization
- Usability

Variations on a Theme

- If the test designer has any visibility into system design, major underlying subsystems can also be test groups.
- For instance:
  - **memory management**
  - **database insertion/deletion routines**
  - **inter-process communication routines**
  - **error logging**

Example for Medical Records Management System

- Database input/output
- Database search
- Error logging
- Report writer
- Editor
- Datacomm with medical devices
- Backup/restore
- Miscellaneous
- Reliability & Stress
- Performance
- Compatibility
- Localization

Properties of a Good Division

- No more than 15 test groups
- Test groups are within 5x of each other in size
- One or more discrete groups can be assigned to each test designer
- Functionality groups can be mapped to individual developers reasonably well
- Any function that might be dropped is in a group by itself

What Happens Next?

- Each test group is given to a test designer.
- The test designer starts listing potential test cases.
- If the test cases are quite numerous, the test designer splits the group into subgroups.
  - **Functionality groups split into smaller functionality groups, often in the order in which the functionality is to be delivered**
  - **System test groups split into smaller groups of the same system test types**

A Common Error

- A common error is doing only functionality testing for a functionality test group.
- This delays finding other kinds of defects until the system test type groups are run.
- Method:
  - **List all the expected functions of the feature in question as test cases**
  - **Go through each system test type and ask yourself if it is applicable to this feature in isolation**
  - **If it is, add test cases to the test case list**

Key Learning #3:

Use the system test types with each functionality test group.

Scheduling the Testing

- The test plan and the test schedule are not synonymous.
- In the test schedule, arrange the functionality test groups in the order in which the functionality is expected to be delivered.
- The system test type groups are usually run after the functionality groups are completed.
- A subset of each group is chosen to be a regression test, which is run on later releases.

Testing for Each Release into Test

- Acceptance or smoke test - is the release sufficiently functional to be tested at all?
- Turn-on test - are the advertised new features sufficiently functional to be tested at all?
- Main testing - full testing of the new features and/or planned system testing
- Regression testing - check-up on previously tested features

Summary

- The “Features To Be Tested” section is supposed to cover *all of the system behavior*, not just the functionality!
- Half of the “Features to be Tested” are the functions of the system. The other half are the system test types.
- Use the system test types within each functionality test group.
- Read the article for more information and references.

Divide and Conquer: Making Sense of Test Planning

Kathy Iberle
Hewlett-Packard
At: STAR '99 West

Abstract

The neophyte test planner is often overwhelmed by the complexity and scope of test planning. This paper presents several practical methods for dividing the testing into manageable pieces, based on the author's experience over ten years of testing a wide variety of products.

Introduction

Organizing the testing itself is one of the most challenging parts of test planning for the beginner. Many neophyte test planners find themselves staring at the nascent test plan with absolutely no idea of how to turn the huge, amorphous mass of testing into some sort of manageable structure. There is plenty of guidance available in the literature for the test planning before, after, and around this point, but advice on this topic is oddly missing in most books.

This paper explains the relevant section of the IEEE 829 standard in simpler terms and presents several different ways to break down the testing into groups of features. The method is based upon the work of Boris Beizer, with some modifications developed during practice. The experience with this method spans ten years at several Hewlett-Packard divisions, primarily involving medical products and inkjet printers.[1]

Terminology

The reader may appreciate reminders on a few common testing terms which are used in this paper.

The IEEE Standard 829 for Software Testing Planning defines the following:[2]

A *software feature* is “a distinguishing characteristic of a software item (for example, performance, portability, or functionality).”

A *test plan* “describes the scope and approach of intended testing activities. It identifies…the features to be tested.”

A *test design* “specifies the details of the test approach for a software feature and identifies the associated tests.”

[1] This does not constitute an official Hewlett-Packard methodology nor does it imply a recommendation or endorsement by Hewlett-Packard.
[2] Section 2 "Definitions" of ANSI/IEEE Standard 829-1983.

The term *system test type* refers to some commonly used categories or types into which system test is often divided. A typical list includes performance, compatibility, configuration, security, supportability, usability, reliability, stress, and recovery testing. The classic list appears in Glenford Myers' *The Art of Software Testing*. Similar lists appear in most books about system testing, tailored to the author's particular field of interest.[3]

The Starting Point for a Test Plan

Many test plan authors start from the ANSI/IEEE Standard 829. This doesn't imply that they intend to produce every piece of documentation called for in the standard - the standard is often used as a guideline for what should be in a test plan rather than a strict template.

The IEEE Standard says that a test plan shall include:

- A list of features and combinations of features to be tested.[4]
- A general statement of approach for each feature or combination of features[5]

This is entirely consistent with most test books and with the approach described in this paper as long as one recalls that a software feature (according to the IEEE definitions) includes behaviors other than functionality. Unfortunately, in common use the word "feature" is frequently synonymous with "functionality".

The "features and combinations of features" are meant to cover all of the software *behavior*, not merely the functionality. The customers are buying the software to get the benefit of its behavior. The software functionality obviously is a large and important part of its behavior, but other aspects such as performance and reliability are also critical to the customer and must be assessed.

Using "feature" as a synonym for "functionality" often leads to an interpretation of the IEEE Standard 829 in which the testing is organized entirely around the functionality, which makes it difficult to plan adequately for the rest of the necessary testing. This in turn leads to an impression that the IEEE Standard 829 is not useful.

Boris Beizer included the idea of dividing testing into "test groups" in his book *Software System Testing and Quality Assurance* (1984). Beizer's test groups correspond to the IEEE 829 concept of "features and combinations of features to be tested." His example makes it very clear that that he was including features beyond functionality:

- Startup and Initialization
- Operator Commands
- Configuration Control

[3] Humphrey, Watts S.; *Managing the Software Process*; Addison-Wesley Publishing Company; 1989; pp. 201-204
Kaner, Cem; Falk, Jack; Nguyen, Hung Quoc; *Testing Computer Software, 2nd ed.*; International Thomson Computer Press; 1993; pp. 54-57
Myers, Glenford; *The Art of Software Testing*; John Wiley & Sons; 1979; pp. 112-119

[4] Section 3.2.4 "Features to Be Tested" of ANSI/IEEE Standard 829-1983.

[5] Section 3.2.6 "Approach" of ANSI/IEEE Standard 829-1983.

- Interfaces and Protocols
- Restart
- Switchover and Recovery
- Performance Testing
- Stress Testing
- Terminal Management
- Off-line Database Generator
- On-line Table Changes
- Security Testing[6]

Beizer followed this example with a discussion of how to generalize the example to a wider variety of software applications. Unfortunately, his discussion was limited to mainframe information-services applications of the mid-eighties. It doesn't include any method for producing a "feature" list for a randomly chosen piece of software.

Using Test Groups in Practice

We started with Beizer's concept of test groups, which is completely consistent with IEEE 829 standards and worked out some methods for actually doing the division of the testing into test groups.

Since the confusion between "feature" and "functionality" has been pretty widespread, we have avoided the term *feature* and used both the term *functional area* and the term *feature area* (in different organizations) to denote a test group, with mixed success. (I suspect that using the term *test group* might be more successful.)

We discovered that the best method for breaking down software behavior into test groups depended on the software development process and the size of the team.

The simplest method that we used is *Divide by Functionality*. This worked well where the test group was separate from the development group and was doing strictly black-box system testing. This method would probably also work in cases where the development group has no apparent plans, doesn't write specifications, and isn't sharing the software design with the test group.

A slightly more sophisticated method that we used very successfully is *Divide by Design*. This worked well where the developers were either planning the testing themselves or working very closely with the test planners. This is an appropriate method for development groups that use staged or evolutionary delivery, have written specifications, and share the software design with the test group. The structure of this method supports subsystem and module testing as well as system testing.

We also investigated a method that I shall call *Divide by Test Factor*. This appeared to be best suited to organizations that customarily define, plan, and manage projects from a risk-based perspective. So far, we have not yet had an appropriate opportunity to try this out.

[6] Beizer, Boris; *Software System Testing and Quality Assurance*; Van Nostrand Reinhold; 1984; pp 186-188.

Divide by Functionality

We found that a simple and successful method for dividing a test plan into test groups is to start with a list of all the functions of the software, and then add every system test type. Some merging of groups may take place after that.

To create the list of test groups:

1. Start by listing the major functions of the product. This list should contain ten to twenty items. The product's data sheet, marketing material, or written specifications (if they exist) will usually provide this list.
2. Be sure that you don’t overlook installation software, users' guides and other learning products, and ancillary software.
3. Add test groups for each of the classic system test types - performance, configuration, compatibility, and so forth. (Do not add a group for functionality testing - the feature groups have already covered that territory.)
4. Merge groups which only have a few tests with another, similar group. Sometimes it has been convenient to merge all the system test types into a single test group named "System" or something similar.
5. Decide where you will put any tests that don't fit neatly into any of the named groups. Sometimes a group named "Miscellaneous" will save time as people think up really odd tests.

When defining test groups for system test types, keep in mind that the system test type lists in most testing books are tailored to the author's particular business. You may need to create your own test type list for your business. For instance, printer drivers don't have large legacy databases to contend with, so conversion testing is not needed. However, the printer and driver must conform to certain externally defined standards, and thus conformance testing is added.

Most projects define a set of groups by trying to split the project up several different ways, then merging the results into a comfortably sized list that satisfies the project’s priorities. Don’t be afraid to iterate! You may need to go through the steps several times, reviewing your list of groups with the developers each time, before you are satisfied with the division.

Example 1: Test groups for a printer driver

- Print Modes (best, normal, econofast)
- Paper sizes (A4, letter, envelopes...)
- Media Handling (duplex, collation, number of copies...)
- Paper Types (regular, glossy, transparency....)
- Installation/Uninstallation
- Application Compatibility (works with various software applications...)
- System: Performance, Reliability, Stress....
- Exception Handling (paper jam, paper out...)
- Cancel/Pause/Resume
- Online Help
- Localization

Notice that some of the system test types (compatibility, localization) have their own group, while the others are all tossed together. This was decided strictly on the expected size of the test group - the compatibility and localization test groups contain quite a large number of tests.

The steps for Divide by Functionality have assumed a black-box approach - that is, the test staff knows nothing about the internals of the software. However, you may well have some information about critical internal structures that aren't visible to the customer - memory management, inter-process communication control structures, error logging. If these are testable at your level, they can form a group of their own. If there's more than three or four of these, you've probably got enough information to use the Divide by Design method instead of Divide by Functionality.

Divide by Design

We found that a more effective test plan could be created by dividing up the software behavior along the lines indicated by the high-level design or architecture of the software. This method produced a test plan that encouraged thorough module and subsystem testing, while making the relationship between lower-level tests and system test very clear. The clarity reduced needless repetition of tests while maintaining the effectiveness of the testing, which we measured indirectly using post-release defect metrics.

However, this method worked only when the architecture was reasonably well documented and the customer-visible features mapped fairly cleanly to identifiable modules or subsystems. When that was not the case, it was not possible to write a test plan from the architecture that looked like it would achieve reasonable coverage.

To create the list of test groups:

1. Start by listing each major subsystem.
2. Go through the product specification or data sheet and map the customer-visible features to the subsystem test groups. The features that are implemented by a single subsystem should already be on the list. There will likely be some features that span more than one subsystem - that is, it takes more than one subsystem to make them work. Add these features to the test group list.
3. Add test groups for each of the classic system test types - performance, configuration, compatibility, and so forth. (Do not add a group for functionality testing - the feature groups have already covered that territory.)
4. Merge groups which only have a few tests with another, similar group. Sometimes it has been convenient to merge all the system test types into a single test group named "System" or something similar.

In a system where the software design maps fairly cleanly to features, this often ends up looking very similar to Divide by Feature, except that the underlying architecture of the system is more visible.

Example 2: Test groups for an application that stores and manages medical test results

- Database I/O - schema for test results and routines to insert and delete test results

- Database Search - indices and routines to find test results
- Datacomm - low-level communication with medical test devices that supply test results
- Editor - displays lists of tests and allows the addition of physician's comments to the tests
- Report Generator - prints the stored tests
- Backup/Restore
- Archive - long-term archival of results
- Error logging
- System - stress, reliability, etc.
- Localization

The test groups were run in more or less the order shown. The underlying database operations were coded, tested, and fixed before the customer-visible features that used them were written. (We had to write test harnesses to exercise the database operations.) The testing and fixing of the customer-visible features proceeded quickly because the database operations no longer failed and therefore any observed bugs were most likely in the newer code.

Divide by Test Factors

We discovered an interesting alternative method in *Perry's Effective Methods for Software Testing* (1995). Perry split the testing into *test factors* - “attributes of the software that, if they are wanted and not present, pose a risk to the success of the software, and thus constitute a business risk. For instance, if the software is not easy to use, the resulting processing may be incorrect.” [7] Perry's examples are test factors that would be applicable to information systems.

This method looked like it would be appropriate in cases where the development and marketing organization are prepared to discuss business risk of each feature, and particularly where the requirements process has led to a documented direct mapping between features and customer needs. If this information is not available, we believe that the test factor method cannot be used because there isn't sufficient information on which to base the definition of test factors.

Checklist for Dividing into Groups

We used this checklist for determining whether a division into test groups was "good":

- No more than 15 test groups.
- Test groups are within 5x of each other in size.
- One or more discrete groups can be assigned to each test designer. More than one test designer working on a single group isn't good because they get confused about who is doing what. If the test groups are so large that you must have multiple test designers within one group, then you need to have subgroups (see below).

[7] Perry, William; *Effective Methods for Software Testing*; John Wiley & Sons; New York; 1995; pp. 25-29

- Functionality groups can be mapped to individual developers reasonably well.
- A discrete test design can be written for each test group. If it seems reasonable to write one test design covering several groups, then the groups should be merged.
- Everything that you would want to test can be reasonably assigned to a group, though it may be a fairly specific subset of that area.
- There is at least one group in which to put tests that apply to the whole system.
- Any major functionality that might be dropped is in a group by itself. (This makes it easier to provide estimates of schedule and resource changes.)

What Comes After Defining the Test Groups?

Once the test group list was established, the testing for each test group was analyzed in detail. The end result of this test design was a list of test cases, which would be grouped into tests. The typical test was a written test procedure to be run manually, covering three to ten test cases. The tests were deliberately sized to be executed in two to four hours, because this worked best in scheduling work for the day in the test lab. (Automated tests often covered considerably more test cases in a single test and the execution time was not such an issue.)

Here's a commonly used method for generating test cases for a test group:

1. List all the expected functions of the feature in question as test cases.
2. Go through each system test type and ask yourself if it is applicable to this feature in isolation.
3. If it is, add test cases to the test case list.

This may seem like duplication of testing - why have a reliability test for a particular feature? Why not simply have a reliability test for the entire system? We found that early testing of aspects other than functionality paid off handsomely in certain cases. For instance, reliability testing of data communication subsystems was extremely valuable more than once in very different products. The early testing allowed datacomm problems to be easily isolated from other problems, and the use of a test harness allowed us to force a wide variety of test conditions that would have been difficult to create otherwise.

The test design method encouraged the invention of such tests by requiring the test designer to walk through the system test types while brainstorming test cases for any feature, subsystem, or module. It very rarely happened that every one of the system test types generated useful test cases - typically just a handful of test cases in three or four of the dozen or so system test types would be added.

It was not unusual for the same test case to show up in the brainstorming for more than one test group. We experimented with keeping the test cases in a database, indexed by test group, by system test type, and by test. We could then search easily for possible duplicates and reduce redundant testing. (The result of reducing redundancy was that an individual test procedure sometimes covered test cases from more than one test group. We considered the test procedure to "belong" to the same test group as the majority of its test cases). The database also turned out to be extremely useful in assessing coverage of the system as the test plan grew.

The division into test groups, the generation of test cases, and the grouping of test cases into tests resulted in a structure like this:

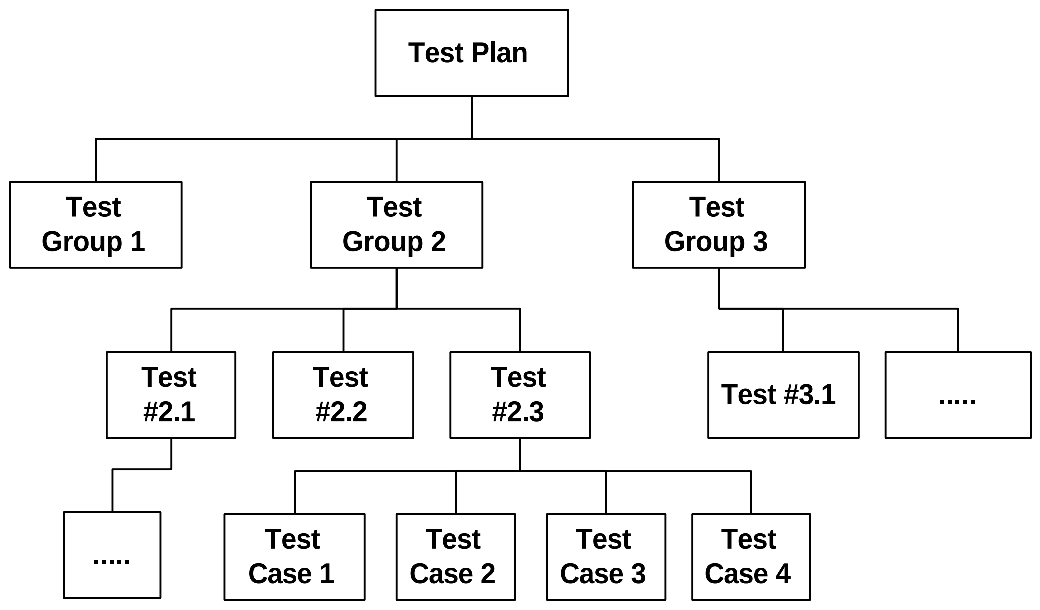


Figure 1: Test Hierarchy for a Small Project

The Bigger Project

Projects that involved more than twenty engineer-years of development generally outgrew the hierarchy shown above. A workable alternative was borrowed from Beizer's *Software System Testing and Quality Assurance*[8]:

A *test group* is the "broadest functional subdivision of the system".

A *test subgroup* is a group of tests that can be designed and executed independently of any other subgroup.

A *test* is a set of test cases that share the same setup and execution environment and cover closely related functional areas.

A *test case* is a single set of inputs, actions, and expected outputs.[9]

Beizer's "test" corresponds to what many practitioners call a "test procedure" - that is, a procedure that is expected to be executed in its entirety, covering several closely related

---

[8] Beizer, Boris; *Software System Testing and Quality Assurance*; Van Nostrand Reinhold; 1984; pp 186-188.

[9] Beizer uses the term "subtest" rather than "test case", but it is clear from the definitions that he is using "subtest" to mean the same thing as the IEEE definition of "test case".

test cases. This might be a sequence of written steps to be executed manually over several hours, or it might be an automated test.

Beizer's test subgroups were a very useful organizing idea for the larger test plan - particularly where the tests in a test group were too numerous for a single designer to design and write in the desired time.

The test group would be divided into subgroups to make it possible to keep track of the test cases. The point at which this must be done is partially determined by the method used for keeping track of the test cases. If any type of database or file-card indexing system is used, hundreds of test cases can be tracked with only test groups to identify them.

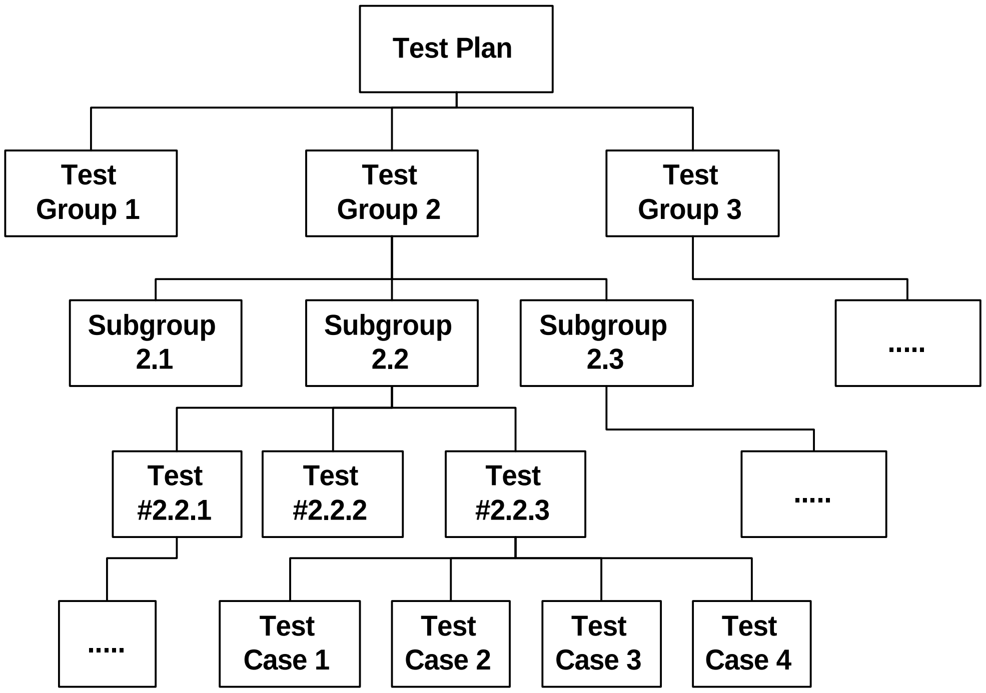


Figure 2: Test Hierarchy for a Medium-Size Project

Test Plan Organization for Large Projects

We found that projects involving multiple teams or durations of more than a year generally needed to add another level to Beizer's structure. We did this by introducing the idea of test *phases* (sometimes referred to as *levels* or *stages*).

The recommended phases vary widely among authors. Typically, unit testing is the first phase. Unit testing is generally agreed to be testing of the individual procedure, function, or method by the developer. This is followed by some combination of module testing, subsystem testing, function testing, and integration testing, depending on the author. The definitions of these phases overlap from author to author. The general idea is that some portion of the system that is larger than an individual function and smaller than the entire system is tested. This sequence is followed by system testing. Products custom-built for a single customer usually finish up with acceptance testing.

We chose to go with a subsystem test phase and a system test phase. Unit testing was performed by the developers and not included in the test hierarchy. The subsystem test phase definitely included some testing that would properly be called module testing.

Each subsystem had its own test plan, with its own test groups and subgroups. There is also a system test plan, which also has its own test groups. The test plans were all listed in the Master Test Plan, which was the top level of the hierarchy. We found that test groups in some of the subsystems recurred in the system test phase. It paid off to keep track of the overlap, because often tests for individual subsystems could be modified for reuse in system test.

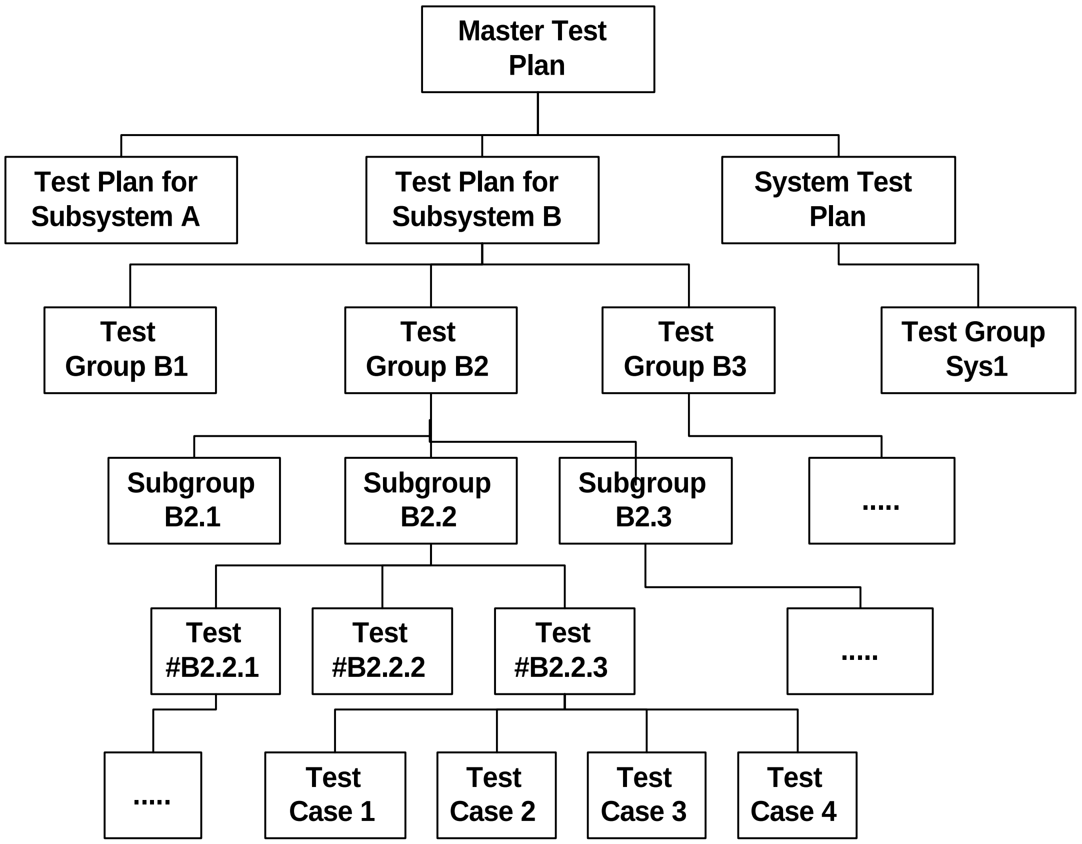


Figure 3: Test Hierarchy for a Large Project

Comparisons with Other Authors

Around the same time as Beizer's book was written, Bill Hetzel took the "features to be tested" and split them into requirements-based, performance capability based, and design-based sections of system testing.[10] The requirements-based section covers the same types of test groups as our functionality-based test groups. The performance capability section picks up stress, performance, reliability, and volume testing. Hetzel also suggests that some of the other classic system test types might be used as performance capability tests.

Our method has the advantage of presenting the test designer with a clear checklist of test types to be considered as potential test groups. I also believe that separating the test groups into requirements-based and performance capability can muddy the waters since the written requirements often include some of the desired performance parameters. This leads to the possibility of writing tests twice by accident, or of neglecting a test because it is believed to be in the "other" section.

Hetzel's design-based tests map pretty well to the Divide by Functionality method for creating test groups. It isn't practical to go this route if the test planner doesn't have access to the software design.

In 1995, Edward Kit presented a method for designing requirements-based validation tests that closely resembles our method.[11] Kit considers usability testing, function testing, and system testing to be separately planned activities. Our method covers the same ground by establishing functionality test groups and system test type test groups, and then revisiting the system test types within every test group . This encourages the planning and execution of interesting non-functionality tests at all stages of the testing.

Kit also stresses the importance of a good test architecture - which includes the organization of the tests into groups.

Summary

Ten years of experience with this method of test planning have demonstrated that it leads even beginning test planners to quickly produce a reasonably efficient and effective test plan. The main points to remember are:

- The “Features To Be Tested” section in IEEE 829 is supposed to cover *all of the system behavior*, not just the functionality!
- Beizer's test hierarchy still works pretty well.
- Use the steps in Divide by Functionality or Divide by Design to split the testing into reasonably sized test groups. Then create test cases for each test group. Use the system test types as ticklers to make you think of good test cases.
- Larger projects will require more levels in the test hierarchy. A very large project will require multiple test plans and a master test plan to coordinate them.

---

[10] Hetzel, Bill; *The Complete Guide to Software Testing, 2nd ed.*; John Wiley and Sons; New York; 1988; pp. 134-138.
[11] Kit, Edward; *Software Testing in the Real World*; Addison-Wesley Publishing Company; 1995; pp. 126-129

Acknowledgements

Thank you to my many colleagues at Hewlett-Packard who have participated in the development of this method over the years, particularly Lou Schoof, John Kent, and Ernie Kaps.

Bibliography

Beizer, Boris; *Software System Testing and Quality Assurance*; Van Nostrand Reinhold; 1984.

Hetzel, Bill; *The Complete Guide to Software Testing, 2nd edition*; John Wiley & Sons; 1988.

Kaner, Cem; Falk, Jack; Nguyen, Hung Quoc; *Testing Computer Software, 2nd ed.*; International Thomson Computer Press; 1993

Kit, Edward; *Software Testing in the Real World*; Addison-Wesley Publishing Company; 1995

Marick, Brian; *The Craft of Software Testing: Subsystem Testing*; Prentice-Hall PTR; 1995

Myers, Glenford; *The Art of Software Testing*; John Wiley & Sons; 1979

Perry, William; *Effective Methods for Software Testing*; John Wiley & Sons; New York; 1995;

KATHY IBERLE

Kathy Iberle has worked in software product development and testing at Hewlett-Packard for nearly 15 years. She has been involved with a wide range of products from data management systems for medical test results to inkjet printer drivers. Kathy has worked extensively on training new test engineers, researching appropriate test methodologies for different situations, and developing processes and templates for effective and efficient software testing. She is currently working as a software process consultant at the Hewlett-Packard site in Vancouver, Washington.

Kathy has an M.S. in computer science from the University of Washington and an abundant collection of degrees in chemistry from the University of Washington and the University of Michigan.