BOOK DRAFT—DEVELOPING SOFTWARE TEST CASES

January 13, 2004

Ross Collard

rcollard@attglobal.net

Chapter 1 An Introduction to Software Test Case Design..... 2

The Definition of Testing..... 2

What Is a Test Case?..... 5

The Nature of Test Case Design 19

Factors which Influence Testing…….

Chapter 1
An Introduction to Software Test Case Design

The Definition of Testing

This chapter discusses testing and test cases, gives examples and examines the major issues we encounter with test cases.

What is Testing?

You probably already have a working definition of the word "testing", but it is still worth a minute or two to consider its meaning. Testing is an organized effort to:

1. Validate that an item works as expected. Depending on the work scope and perspective, an item can be a feature, system, subsystem or component. The tester confirms that the item is working correctly according its specifications (for brevity, specs) and the clients' expectations, and may validate that the specs are correct and the expectations reasonable. The tester also exercises the system under abnormal conditions, to check the system does not misbehave under those conditions.

2. Find and resolve discrepancies. These discrepancies are the variances between the actual vs. the expected results from using a system, and are the clues to underlying defects. The tester strives to find the most significant problems as early as possible, expedites the software engineers' fixes for these problems, and re-tests after the fixes have been applied.

3. Provide an objective assessment and timely information to decision makers about the state of a system, for example, its readiness for release. The tester collects and interprets metrics about the items which have been tested, how thoroughly they have been tested, the number and severity of the problems found, and the status of fixing these problems. The tester also helps assess the risks of additional unknown defects which could be lurking in the item.

4. Help improve quality. The ultimate goal of testing, and the ultimate role of the tester, is to provide feedback which improves items. Testers assess quality but in themselves cannot directly improve quality. Nevertheless, the results of running one test case often are worth a dozen opinions. Testing ideally goes further than finding bugs to be fixed – the feedback raises managers' awareness and educates software engineers, and thus helps to avoid creating further bugs.

The relative importance of these four testing objectives varies depending on the test project goals and priorities. The directive: "Find all possible bugs" leads to a different test strategy from: "Provide a risk-of-going-live assessment one week before each major release date." For example, let's say the decision-makers state that no more bugs will be fixed during the remainder of a project unless they are showstoppers. In this situation there is no point in continuing the hunt for

every minor defect. Other possible objectives of a testing project include surfacing the critical problems asap, which is not the same as finding all the bugs, and certifying the system conforms to a particular standard such as 510(k), the guideline issued by the U.S. Food and Drug Administration (FDA) for medical devices like pacemakers and x-ray machines.

The Value of Testing

Testing is vital, as it enables us to evaluate whether an item is likely to meet its goals, the users will be satisfied with the service they receive, and the system operation is likely to be trouble-free. Testing builds confidence that the users will not encounter problems in live operation, and is a type of risk-mitigating insurance. When utilized wisely it is a critical success factor for an organization.

The upside potential in the area of improved system quality is enormous: a competitive edge; improved profitability, market share and corporate image; and the ability to respond adroitly to changing business conditions and capitalize on opportunities. The old adage is now more true than ever: when quality goes in the name goes on. System quality also has personal benefits including job security, the ability to sleep well at night and bragging rights with one's peers.

On the other hand, the downside potential of system defects is horrendous, and testing helps us to prevent them reaching live operation. Did you hear about the one-character programming error which caused ten million telephones to go dead? This problem happened at DSC Communications, which is now defunct <<reference Collard article >>. Did you hear about the glitch which caused a loss of over two billion dollars within two days? (At eBay.) Or the network problem which delayed ambulances in London over a period of three days, contributing to 20 deaths? System defects have caused numerous human fatalities – for examples, see Robert Glass' book: "Computing Calamities" << ref. >> and the comp.risks news group. The messages are that (1) defects can be deadly serious, (2) many disasters could have been avoided with better testing, and (3) systems are sufficiently complex and critical that effective testing is imperative.

In the words of Boris Beizer: "The act of designing tests is one of the most effective error prevention mechanisms known. The thought process that must take place to create useful tests can discover and eliminate problems at every stage of development." Especially with automated testing, most of the bugs are found during the building of a test case. The test case under construction is used in a trial run, and does not work correctly if the feature being tested has a bug. The test case won't work (and give a valid pass) until the feature is fixed. Does this mean there is no point in automating test cases, because by the time the test case is ready the feature contains no more bugs? Not at all. The act of building the test case is an important way to debug the feature, and after the feature is working acceptably we still may want to re-test (re-run the test case). Re-testing, which is called regression testing, helps detect the unintended side effects of an apparently unrelated change.

Testing is not Magic

We can design and build the quality into a system, but we can't test the quality into that system after it has been built. Since it is an assessment, testing in itself cannot guarantee quality.

Chapter 1
An Introduction to Software Test Case Design - Exercise 1.1: Whoops, That's My Calculator!

Software engineers build and maintain systems, and they ultimately are responsible for the quality of their own work. As the authors they, not the testers, usually have the most influence on quality. Test teams often have the label "QA" (for quality assurance), but this is a misnomer since the testers cannot assure quality. Testers help by providing useful feedback. Testing does not include creating the system requirements and design, but can have an important role in questioning and validating them. Testing does not include debugging and fixing, but provides the information to direct those efforts.

Testing can be ineffective, inefficient or both. Effective testing discovers a sizeable portion of problems early, and primarily the more significant problems. It assists in resolving these problems quickly and reliably. Effective testing also delivers timely, trustworthy and actionable information to decision makers about the status of the system under test (SUT). It appears to be lucky and good fortune but actually is no coincidence – the testing is smart, focused, organized and performed by capable testers. Ineffective testing by contrast overlooks significant problems, produces doubtful test results and may give a false confidence in the quality of the system

Efficient testing allows organizations to respond adroitly to opportunities by expediting system delivery and modifications, and to maximize test coverage by using the test facilities well. These facilities are often collectively called the testware, which includes test equipment and automated tools, models of the SUT, methods for test case generation, test case libraries, databases of test data, and testing procedures. Inefficient testing squanders time, money and test resources, and distracts attention from other opportunities to improve quality.

Testing is not free. Productive testing takes preparation, resources and time. Many test projects are done with unrealistic deadlines and resource limitations, and the quality suffers. Testing may not be well understood or respected. There can be skepticism within an organization about the return on investment (ROI) of testing, under-appreciation of the contributions of testers, unrealistic expectations that testing will catch every bug, and us-versus-them conflicts among testers and others. We'll delve into the issues in the next chapter, entitled: "Challenges and Complications". Despite these limitations, testing remains an imperative for organizations.

Exercise 1.1: Whoops, That's My Calculator!

(Allow 5 to 10 minutes for this exercise.)

You dropped your calculator and want to check if it still works. It is not a very expensive calculator, but you would prefer to avoid replacing it. On the other hand, if it gives wrong answers (or no answers at all), it has to go. There is no damage which is obvious at first glance, for example, neither the case nor the display is cracked. The calculator performs basic functions like add, multiply and even the square root, but not complicated functions like logarithms and exponentials.

What should you do to test the calculator? Assume for now that you cannot open the calculator's case and access its insides. The next exercise in this chapter contains a suggested answer to this problem for you to compare and critique.

What Is a Test Case?

A *test case* exercises one particular situation or condition of the system being tested. What need does a test case fill? The answer is that a test case describes what to test and how, and provides direction to the person (or automated test tool) which performs the testing. This direction includes how to set up and execute the test case, what behaviors to monitor, and how to evaluate the results.

The development of a test case starts with its purpose, which is to fulfill some test requirement. A *test requirement* is simply a statement of what needs to be tested. I advocate that all test cases are based on test requirements (also known as test objectives). You can find these in the test plan, if one has been developed which contains test requirements. If not, ask knowledgeable stake holders to help you identify the test requirements.

A simple test requirement for an order processing system is:

> *"Verify that if a customer's credit is denied, an order cannot be shipped to that customer before payment has been received in full."*

To convert the test requirement into one or more test cases, we need to know something about how the system works. Or if we don't know, we need to make assumptions about how it works. Let's assume that when an order is processed, the system is expected to check whether the customer requested credit and, if so, whether the request was denied. If it was denied and the customer has not pre-paid for the order in full, the system is expected to issue this warning message:

```
"Credit is not authorized for customer aaa, account
# bbb. Cannot ship order # ccc. The credit denial
code is # ddd."
```

By itself, the test requirement is not specific enough for us to readily test it. The following example is a typical test case which we can develop for this requirement, and is specific enough to run and check the results:

1. Access the record of customer Peter Kennedy, account # 1234.
2. Confirm that this customer has an outstanding order, # 4567, which is ready to ship.
3. Confirm that a credit authorization request was made by this customer.
4. Confirm that the credit authorization request was denied with denial code # 6789. (The value of the denial code should be checked, but receiving the right response from the request – to deny credit – is what is important for this test case regardless of the specific denial code. Nevertheless, if the right decision is reached but the denial code is incorrect, the test case will be considered to have failed and the discrepancy noted in the test results log. In the judgment

of the test case designer, it is not acceptable if credit was denied as expected but for the wrong reason.)

5. Confirm that the customer has not made a full prepayment for this order.

6. Confirm that the customer does not have a prior credit balance which is sufficient to cover the cost of this order.

7. Attempt to release this order for shipment.

8. Confirm that the system does not release and ship the order and instead issues this warning message:

   ```
   "Credit is not authorized for customer Peter
   Kennedy, account # 1234. Cannot ship order # 4567.
   The credit denial code is # 6789."
   ```

You will see in the next few pages that this test case is rudimentary and incomplete, but it is still worth examining. The test case contains both desirable and undesirable characteristics. Its ability to provide a definitive pass / fail result instead of an inconclusive one is desirable. On the other hand, the hard-coded data values, such as order # 4567, make this a high-maintenance test case – every time one of these hard-coded values changes, the test case needs to be updated. Later we will examine the question of test case effectiveness in more depth.

The test case covers the stated test requirement, to check that an order is not shipped without a credit authorization or full prepayment. For simplicity, in this example I have overlooked nuances which a robust test case would consider. Despite what I said earlier about this test case providing a black-and-white pass/fail result, simply checking that the warning message in step #8 has been issued is probably not sufficient. We might want to go further and check for a stronger control, such as a lock which effectively has been placed on this order's record in the order database, to prevent the order from accidentally shipping.

And what happens later, after this message has been issued, if and when the customer pays in full in order to reinstate and release this order? We would want another test case to check that the order is then shipped as expected. There could already be a separate test requirement, with its own suite of test cases developed by other testers, for this pay-later situation. We'd have to review the broader set of test requirements and test cases to find out if this situation was covered elsewhere, and we'd want to make sure that any interdependent test cases are coordinated.

A test case which later releases an order could be dependent on the one which initially prevented releasing that same order, or these test cases also could be deliberately designed to be independent. How this independence can be accomplished, so that running the latter test case does not depend on the successful prior execution of the former one? For the latter test case to be self-sufficient and self-contained, it must include within itself all the steps needed to establish a suitable order to reinstate and release (one which earlier was placed on hold because credit was not authorized).

Note also that the test case cannot run in a vacuum: preparation is needed. The system under test must be loaded in to a test machine which is as close as possible to the target machine. The system installation, the configuration of the test machine and the loading of test data files should be checked as pre-conditions to running the test case, to minimize the possibility of false results. A customer database has to be established with a record for Peter Kennedy, account # 1234, who has bad credit. An order database is needed which contains order # 4567, this order must be for customer # 1234, and the customer has not pre-paid fully for the order. We need a record of a credit authorization request for order # 4567, together with a record of its denial with code # 6789, and these records must be stored where the test case can access them. This preparation effort is not inconsiderable.

There can be multiple test cases for a test requirement. Let's assume that the business has two types of customers, domestic and foreign, and that the system handles them in distinctly different ways. If Peter Kennedy is a domestic customer and his test case works, we may be willing to assume that the system will work correctly for other domestic customers too. But we probably would not be willing to say that the system processes foreign customers correctly, based on this evidence from a domestic test case. This means we need another test case for a representative foreign customer.

Systems have many requirements which need to be tested, so an entire system can have a very large number of test cases. The *test case library* is simply an organized repository of the test cases for a system or collection of systems. The test group usually maintains the library using version control and configuration management.

The Contents of a Test Case

What does a test case look like? While test cases come in many forms, most of them contain the following information:

1. The test case identification.
2. The pre-conditions to be met, in order for the test case to be successful, i.e., to execute as planned and reveal the desired information quickly and reliably.
3. The inputs and triggering events needed to initiate and drive the test case.
4. The test procedure, the series of steps needed to run the test case.
5. The expected results of the test case (the outcomes), including a clear indication of which results determine if the test case passes or fails.
6. The test environment, which supports the test case as it runs.

The remainder of this section describes these types of information.

1. The test case identification.

The test case identification includes its purpose, tests requirement, a brief description, and cross-references to the feature(s) it tests. Often one combined statement is sufficiently informative that it covers all four of these. This information allows us to organize, find and manage our test cases, so while this section is routine it is important. We usually want to record this information on a test case:

1.1 Test case ID number, name, and version number. The ID number should be unique and comply with the organization's or the project's test case numbering standards. The name needs to clearly and distinctly identify the test case, but also should be succinct – and meeting both needs can be a tall order.

1.2 Description and purpose of the test case. This may include the test requirement which the test case addresses, if a requirement has been defined and is not obvious from the purpose.

1.3 Test suite, group or category to which this test case belongs.

1.4 Cross-reference to the specific feature(s) to which the test case applies, to the system version(s), hardware platform(s), and so on.

1.5 Person who is responsible for this test case, either its creator or the person in charge of maintaining it.

2. The pre-conditions.

Every test case has pre-conditions which need to be met for that test case to run successfully. For example, we might need to check that we are using version 24 of a test database and not version 23, because the test case will not work correctly with the earlier version. Violations of pre-conditions are a common reason for false test results, where a test result indicates the system's behavior is acceptable and the test case passes when it should have failed (a false positive), or vice versa – the result indicates the test case failed when the system is working as expected (a false negative). The causes of these false conclusions include neglecting to clearly define the pre-conditions, and failing to determine if these pre-conditions have been met.

Let's say that a tester is preparing a test case today which she knows she will be running within a few days. The tester thinks: "It is obvious to use version 24 of the test database. I won't forget that, so there is no need to write it down." She is overlooking the futurity of the test case, the possibility that an unknown person could pick up this test case in six months and re-use it. The tester should ask: "What kind of person is likely to use this test case in the future? What can I reasonably assume they will know, and not know? What do I need to tell them?" For example, we could instruct the tester to check routinely the system configuration and document any deviations from a standard configuration or operating state before running a test case, because this saves time and avoids confusion during debugging.

We need to explain pre-conditions which are not obvious to future users of the test case. However, diligently documenting the pre-conditions carries its own danger. If you earnestly

list every pre-condition you can end up with dozens for each test case. The longer the list, the more time-consuming it is to compile – and to maintain as conditions inevitably change. Long lists are distracting – something critical buried among many mundane items may be overlooked or not adequately checked. We can organize groups of common pre-conditions which apply to multiple test cases, and provide pointers to them instead of repeating the same pre-conditions from test case to test case. This helps keep the lists manageable.

Documenting the pre-conditions requires discernment though it sounds routine. Some pre-conditions are mandatory but do not need to be documented, because they will be obvious. For example, an essential pre-condition for running any test case is to have a live electricity supply. Do we need to include this line on every test case: "If the screen is blank and the keyboard is dead, check that the electricity is turned on."? The answer is no, because regardless of who picks up a test case in the future, we are confident that person has the basic competence to check the electricity supply if necessary.

Whether explicit or implied, a directive to check the electric supply can be ambiguous. Let's say that the feature we are testing runs on a laptop, where electricity can be supplied from batteries or from a wall socket. If the laptop could work differently depending on the power source, the test case should explicitly specify which source to use. One guideline: anything that is not obvious to a new hire in the test group, summer intern or co-op student must be documented.

3. The input data and triggering events.

The inputs drive the test case and lead to the system behavior we want to examine. An input can be either (a) a specific "hard" data value which is hand-entered, (b) a pointer to a stored data field which is continually updated as conditions change, such as the current balance of a bank account which changes with deposits and withdrawals, (c) a set of rules describing how to generate the data, (d) a transaction, (e) a batch file or database, or (e) a triggering event which initiates an action. Examples of triggering events: a click on an enabled button on a Web page; a ticking clock reaching midnight; and an alarm which sounds to alert a system administrator that a piece of equipment has failed.

The term "input" can include indirect inputs, like the environmental settings of the infrastructure which supports the system, or the initial state of the software. Since the definition of an input is broad and inclusive, the inputs for a test case can overlap with its pre-conditions. I also have seen the input section of the test case include an explanation and justification of the test coverage; the rationale behind which particular data values in the input domain have been selected for testing and why other values have not been selected.

Test case design depends on selecting the "right" inputs. One danger, for example, is to repeatedly use the same input data values in test projects, instead of using intelligent variations of these values. This is dangerous because of the pesticide paradox (coined by Boris Beizer). This paradox says that test cases lose their effectiveness over time as the software engineers learn from the testing feedback what their errors are, and change their behavior to create less of these types of errors.

We are more concerned with the suite of test cases covering the bug space (the set of all bugs that the software might contain) than the input space (the set of all possible inputs). There is not a one-to-one correlation of bugs and inputs – in some situations, the number of bugs is much smaller than the number of inputs, while in other situations the opposite occurs. A system which accepts a rich diversity of inputs but has little internal computation, such as middleware, probably has a high ratio of inputs to bugs. By contrast, a system which contains complex computational logic and is driven by a relatively small amount of input data, such as a system which calculates the trajectory of a spacecraft, probably has a low ratio. It usually is easier to deal with the input space even if it is heavily populated, though, because the bug space is much harder to define.

For example, if a five-character string is an input to a system and we test every distinct value for this string, and each test case requires one millisecond to run, the total test execution time will be about 30 years. If instead each test case takes a mere second, the elapsed time balloons to 30,000 years. This assumes that the input characters are encoded using 8-bit ASCII characters; where there are about 2^{40} distinct input possibilities for a five-character string (8 bits multiplied by 5 characters equals 40 bits, and equal bit can have 2 values). Full test coverage requires a total of 2^{40} or ten thousand million test cases. In 16-bit Unicode characters, which are widely used in international systems, the possibilities number 2^{80}. At this point the tester calls his or her significant other and says: “Don't wait up for me, honey, I’ll be working late.” You might find this argument for ten thousand million test cases unconvincing, though, because the vast majority of these test cases are low value-added ones which in practice we would not bother to run.

The test case inputs do not always have to be carefully chosen. A so-called dumb monkey test utilizes random variations of inputs which are usually automatically generated, to rapidly cover more of the input space than analytical, planned testing. Dumb monkey testers often quickly find bugs which crash the system or block users from executing a feature. These test cases are generally low-value, though, because they use data values which are seldom encountered in actual live operation.

4. The test procedure.

The test procedure describes how to run a test case. Writing test procedures is laborious and is best limited to documenting the minimum information needed to ensure the test case runs as intended. The end goal is to test effectively, not to win an award for the prettiest test documentation. Sometimes comprehensive test procedures are required by government regulation or company policy. A comprehensive test procedure typically contains the following information:

- 4.1 Resource requirements for performing this procedure.
  - 4.1.1 Skills needed to perform this procedure.
  - 4.1.2 Tools, equipment and facilities needed – these can alternatively be listed as part of the pre-conditions.
  - 4.1.3 Calibration and certification requirements. These are not needed unless the item being tested is high risk, regulated or both.

4.2 The actions to take to run the test case.
- 4.2.1 Steps which need extra controls or special attention (e.g., approval of a security officer), because of their risk or difficulty.
- 4.2.2 Steps needed to check the pre-conditions of the test case.
- 4.2.3 Steps needed to set up the test case (e.g., establish the test environment; load the system being tested, etc.) Note that this and the prior step 4.2.2 may not happen in a linear sequence. Often there is a complicated inter-play between setting-up, checking the state and adjusting the environment to satisfy the pre-conditions.
- 4.2.4 Steps needed to run the test case.
- 4.2.5 Steps needed to check the post-conditions. A post-condition is any requirement of the system's last state after the test case has run. For example, if a test case processes a bank deposit, a post-condition is that the account balance increases by the amount deposited. Another pre-condition is that all other account balances remain unchanged.
- 4.2.6 Steps needed to capture and evaluate the results of the test case (this includes the post-conditions).
- 4.2.7 Steps needed to restore the test environment to a specific acceptable condition, ready for the next test case.

Most of the time, the test procedure is a suggested guideline and not mandatory – it does not have to be followed rigorously. Testers still have to use their heads and can vary from the procedure on the job as makes sense. The steps listed above, 4.1.1 through 4.2.7, may not happen in the particular sequence shown here. While there are some pre-conditions we can check before the test "set up" begins, for example, there may be other pre-conditions we can only check after the set up. We follow this cycle: 1) check the pre-condition, 2) if the pre-condition is not met, modify the system (often automatically) so that the pre-condition should now be met, then 3) check the pre-condition again. If the pre-condition is still not met, the set up has failed. When we are testing high risk, complex systems, especially in regulated situations, the test procedure is not merely a suggestion or guideline; it usually must be followed rigorously with documented evidence that each activity has been performed.

Many people do not provide step-by-step procedures in their test cases, and justifiably. They might be concerned about: (a) spending most of their time documenting minutiae rather than thinking how to test better, and losing the concept of what to test and why in the focus on the mechanical details; (b) dulling the tester's curiosity and creativity, and assuming that the test case designer (who often is physically remote) knows best and thus should closely direct the tester who is at the center of the action; or (c) committing to a stiff document maintenance effort as the procedural steps change with the evolution of the system, or building in obsolescence if the maintenance effort is not forthcoming.

Documenting test procedures is laborious and can provide little return on the effort, so we want to capture only what's necessary to successfully use the test case, and to allow for the possibility of re-using it beyond the immediate future. The details of the actions should be clear enough for a new tester to be able to execute each activity. The effectiveness of a test case is often based on its ability to be run by anyone, including inexperienced people.

The best test procedure, in my opinion, is the one we don't need to document at all because it is obvious and unambiguous to everyone involved. In this situation, the test case needs to contain only the pre-conditions, inputs and expected outputs. We also do not need to prepare a test procedure if we can point to a standard one which does the same thing, in an existing library of test procedures.

5. The expected results.

The expected results section of a test case describes the acceptable outcomes from running the test case. These results must be attained in order to declare that the test case has passed. They include:

- 5.1 Expected values of the outputs from this test case.
- 5.2 Other outcomes.
  - 5.2.1 Final (post test case) values of stored data fields.
  - 5.2.2 Measures of the final state of the system and its environment.
  - 5.2.3 Other observable changes in the system and its environment.
- 5.3 Decision rules for evaluating the test results, if their expected values have not been already pre-determined. An oracle is the entity which assesses whether the test results constitute a pass or a fail.

What are the differences among an output, outcome, and result? An *output* is any information content produced and delivered by a system, such as a printed report, screen display or data transmitted to another system. An *outcome* is any change of significance in a system or its environment after the execution of a test case. The set of outcomes for a test case include the system outputs and more. The outcomes include changes which are not explicit outputs, such as database updates or changes to internal switch settings. The term *result* is used in two different ways, to either (a) refer to an output or outcome or the group of outcomes for a test case, or (b) indicate whether a test case has passed or not. In this book, I'll use terms like "the test case has passed (or failed)". To be strictly correct, I mean that the system under test has executed the test case and provided acceptable outcomes (or not).

Pre-defining the results for a test case is a cornerstone of test case design. We observe or capture the actual outcomes of a test case, and compare them to the pre-defined expected outcomes. If they match within a reasonable margin, the test case has passed. What if we don't have a usable, pre-defined expected outcome? Without it, we'll run the test case, eye the actual outcome, compare it with our expectations of what's acceptable, and play the game of "When I see it, I'll know if I like it". Or we may encounter an indecisive result, based on our inability to make fine-grained distinctions. We could get into mystical prognostication as we wring our hands for days and mutter: "I like it. I don't like it. I like it..." There's nothing magical about pre-defining the test outcomes, of course, if we are sure we will take sufficient time for analysis after the test case has run.

Perhaps the worst situation is where a test case fails and we are convinced that we have a defect by the tail, but nobody else agrees the problem is real. The software engineer says with a sniff of distrust, "It worked on my machine." He has a different configuration than the tester, so the test case does not run the same way on his machine. Or the software development project leader

says: "That's not a valid test. The user wouldn't, couldn't or shouldn't do that." Of course, what the user shouldn't do has just corrupted the database and crashed the platform. Or the software project leader says: "That's not a bug; it's an undocumented enhancement. The feature works according to the way we coded it, not as it was specified, and the software engineers like it better that way." So an expected test result which is pre-defined, agreed-on can reduce the wear and tear in getting problems resolved.

Sometimes the feature specifications are not precise enough to calculate the test case outcomes exactly. Or the testers do not pre-define test results precisely enough because a sizeable effort is involved: the exact values of outputs are often difficult to compute independently, without using the system we are testing. In this situation, rough estimates of the outputs with ranges of tolerance may suffice. Common-sense estimates are particularly apt in low-risk, low-complexity situations. If the risk or complexity is high, then we need to develop acceptance criteria, in the form of a set of decision rules, to evaluate whether the actual results from the testing are acceptable. And many false passes of test cases occur when testers do not notice suspicious behavior which falls outside the test case's immediate focus.

6. The test environment.

No test case runs in a vacuum. One part of the test environment is the support that the feature or system under test (SUT) needs to run in test mode. An application requires an infrastructure or ecosystem, including system software such as operating systems, the other application software with which the SUT interacts or co-exists, databases, networks and hardware. The infrastructure usually must be configured in a specific way for the test case, so we define the configuration settings and set-up process as part of the test environment. The infrastructure used in testing ideally mimics the live operational environment, but often compromises are required because of test equipment limitations. These compromises may require that we adjust the test results to translate from the SUT's observed behavior in the test lab to its likely behavior in live operation. We define the adjustment process in this part of the test case.

The other part of the test environment, in addition to the infrastructure needed to run the SUT, is the testware. This is a collective term for the facilities needed to drive the testing, observe the SUT's behavior, capture the test results and manage the test environment. The testware includes automated test execution tools, if they are being used; diagnostic tools which monitor the state of the environment, such as processor and memory utilization; diagnostic tools which monitor the state of the SUT, such as tracing the path taken through the code and noting changes in internal switch settings; test databases, such as a customer database for a customer relationship management (CRM) application; and simulators of the SUT. The testware co-exists in a symbiotic relationship with the SUT. It can interface with the SUT but remain separate, or the testware can place invasive probes into the SUT.

Sometimes the test environment is obvious and its set-up and control procedures are minimal – we do not need to mention them in the test case. By contrast, sometimes managing the environment dwarfs the test case execution and results evaluation. This includes the effort to set up and check out the test environment, monitor the environment during test execution and restore

the environment after testing. The more important, complex and risky the test environment is, the more carefully we need to define it in the test case.

Usually there is a high degree of commonality in the test environments needed by the various test cases in a test suite or test case repository – it s not unusual to see 90% or more of the same information in the test cases. We need not repeat the same information for every test case: we can record it in one place where multiple test cases reference it. A different type of overlap occurs when information about the test environment overlaps information recorded in other parts of the test case, such as the pre-conditions and environment set-up. Redundant information is undesirable, because it leads to inconsistency and more maintenance effort to update the information as circumstances change. The test case designer uses common sense to determine where to place information, and whether there should be deliberate redundancy or cross-references. It is a good idea for a test team to have guidelines on what information to place in which category, in order to improve consistency.

Variations on the Theme

There is no one standard or correct way to document test cases – three projects may use three distinct formats, and sometimes one project uses three formats. Variations might be justified when test cases need different information, or can simply result from a lack of coordination. The testing community is not consistent in how it defines and uses test cases, though the test case is a fundamental concept of testing. Descriptions of test cases generally sound alike, but when we examine concrete examples we find the concept of a test case is interpreted in significantly different ways.

The level of detail in test cases varies widely. At one extreme, a test case may be nothing more than a one-line verbal directive from the boss: “Fred, go check out that Web page”. The boss reckons that Fred is sufficiently experienced and responsible that he can work out the details for himself, and that when Fred later returns and says laconically: “Yup, that page works”, we do not need a detailed record of what he has done. At the other extreme, if warranted by the risk of the situation, the complexity of the test case, and the inexperience of the tester, the same instruction could be defined in considerable depth: “First, ensure your test machine is equipped with a Pentium 4 processor, is running Internet Explorer 6.0 on top of Windows XP, has a 56 kbps modem, and is intermittently receiving e-mail through Outlook Express in a minimized background window. Make sure that Microsoft Office is *not* running on your test machine. Second, check that the load generation tool is already transmitting requests to download the same target Web page at an average rate of 100 hits per second, commingled with requests to download other pages at the rate of 100 per second. Third, confirm the Web server log is enabled and capturing the stream of page download requests....” These instructions are the start of a test case which was designed to provoke a wily bug to make an appearance. The bug was expected to appear during a page download under moderate to heavy loads (hence the load generation tool), while e-mail was being sent and received (apparently with any e-mail software package), but not when Office was running.

The amount of test case detail is right when it balances the dangers of under- and over-documentation. Insufficient documentation can lead to a test case being mis-executed, its results mis-interpreted and its re-use impeded. Excess documentation is at least as dangerous. Documenting a test case often consumes 75% or more of the total effort spent in its development, impeding creativity and productivity. And the more extensive the paperwork, the more effort is needed to update it as conditions change. We will examine the question of how much documentation is enough in Chapter tbd.

Common sense prevails: you should feel free to customize the test case outline to meet your needs on a particular project. Having said that, I now want to be a hypocrite and disagree with my advice. If our test projects do not use a common format for test cases, we risk inconsistency, loss of institutional memory (which means we don't learn from others' experience), and limited re-use of test cases. So you can consider the outline presented earlier as a necessarily generic starting point, to be customized as appropriate to fit your projects. Once you have adapted the format to fit your needs, though, you should establish the revised outline as your standard for future test projects. Don't allow pointless re-inventions of the test case format from project to project. The test case outline presented earlier is typical of manual test cases which are planned and documented, but not of all test cases. Automated test cases and exploratory test cases are two types which look quite different from the outline in this section, so we'll examine how these differ.

Automated test cases are software code modules or scripts which tell an automated testing tool what to do. In effect, the test case is documented in the test language (like a programming language), as instructions or lines of code which tell the test machine what to do. Many automated test cases are similar to manual ones—they check the initial conditions, obtain the input data to drive the test, follow the test procedure step by step and run the test case, capture the outcomes from its execution, compare the actual outcomes with the expected ones, and evaluate and record the results. However, automated test cases can do more than merely mimic manual processes: just as automobiles are more than their original concept as horseless carriages, automated test cases can undertake jobs which are infeasible with manual test cases, such as placing a heavy concurrent load on a system.

Exploratory test cases are intentionally free-form, hands-on and interactive. In an exploratory test, we do little or no separate, hands-off prior planning and minimal test case documentation before we experiment with the system on-line. We might not even have separate test cases, each with its own purpose and distinct boundaries, as one test activity blends into the next. The level of pre-defined detail for a whole suite of test cases is the one-line verbal instruction we saw earlier: "Fred, check that Web page." Even if instructions for exploratory testing are terse, in well managed test projects there is usually a clear and specific goal to be accomplished by the exploration. In exploratory testing, the test case design and execution happen at the same time, with rapid feedback loops of test results to help refine the test cases. The technique is particularly useful when the specs are missing or weak. We will discuss exploratory testing in Chapter tbd.>>

In their variations of test cases, some testers incorporate other related information which is not part of the traditional test case, such as the date and time when they ran the test case and the pass/fail results. Since we can run a test case more than once, testers more commonly capture the

outcomes from each execution of a test case in a separate test log. Other information which people capture in their test case documents include: the scope of applicability of the test case; a glossary of terms used in the test case; the test equipment needed, and how to set up this equipment; whether the test case is manual, automated, both, or manual now with future plans to automate it; the type of test case (such as a feature, performance or usability test case); the test technique used (e.g., equivalence); other test cases with which the test case interacts; where and how to report the test case results; the tool to be used to automate the test case; the history of changes to the test case; and the names of people who reviewed and approved the test case

Exercise 1.2: Survivor or Dearly Departed?

(Allow 15 to 20 minutes for this exercise.)

In the prior exercise, I asked you how we would test a calculator after it was dropped. Review the suggested approach below and answer the questions which follow it.

Answer to Exercise 1.1

Perform these eleven activities to check a calculator after it has been dropped:

a. Visually inspect and check that the calculator's electrical connections to (i) the battery and (ii) the wall power supply appear undamaged and are still capable of safely providing an electrical current to the calculator. Warning: take appropriate cautions with the wall power supply to avoid being shocked.

b. See if the calculator turns on and powers up.

c. Check if you can read the display.

d. Check that each key works: (a) the number keys, and (b) the operator keys such as "add" and (c) the special keys such as "clear".

e. Test the basic arithmetic features individually: add, subtract, multiply, and divide. (By individually, I mean test of only one feature at a time. For example, check the add feature in one calculation, then check the subtract feature in a different calculation.)

f. Test the advanced features of the calculator individually, such as the square root.

g. Test combinations of features by calculating formulas which contain a mix of basic features.

h. Test combinations of features by calculating formulas which contain a mix of basic and advanced computations, such as Pythagoras' theorem.

i. Consider whether the owner of the calculator uses it often in any particular way which is reasonably complicated, such as calculating the standard deviation of a set of numbers. If the owner frequently calculates standard deviations, let's say, then calculate a representative one as a test case.

j. Test the calculator for its usable operating duration on battery power. Let's say the calculator is expected to run on batteries only, disconnected from the wall power supply, for at least four hours of active use, This test case finds whether the calculator still meets this expectation, and if not, how long the user can anticipate the damaged calculator will run.

k. Test the calculator for endurance and robustness. Subject the calculator to a reasonable amount of shaking, squeeze the case and pound the keys hard and rapidly, in simulation of typical everyday stresses.

Assumptions

The calculator can be powered by batteries or from a wall outlet socket.

The calculator cannot be powered by other means, such as solar panels.

The calculator computations should be the same for the same inputs and series of computational steps, regardless of whether the calculator is powered from batteries or the wall socket

The calculator is not programmable.

The calculator does not support multiple methods of numeric data entry and display, such as a scientific or reverse Polish notation as well as the everyday common notation. If you think this is improbable, the vast majority of Hewlett-Packard calculators have supported both. (In the early days, HP was king of the hill in calculators.)

The calculator does not have permanent memory which stores the user's data indefinitely when it is turned off.

Level of Detail

Some of the statements are broad and open to interpretation, leading to subjective, inconsistent testing, and possibly – depending on the skill of the person assigned to test the calculator – inadequate testing. In these situations more specific instructions are needed, like these:

In step g, perform these specific calculations and see if you get the right answers:

*g1. Compute the value of [1!+ 2! + 5! + 6!], where n! is the factorial of n, or [1 * 2 * 3…. * (n-1) * n]. This test case exercises a combination of additions and multiplications. The expected answer is 845.*

> *g2. Compute each person's share of a lunch tab. The bill total of $55 will be split into four equal shares, after subtracting $8 for Nancy's glass of wine (she'll pay for that separately). Sales tax of 7% must be added to the bill. This test case exercises a combination of all four arithmetic operations (additions, subtractions, multiplications and divisions). The expected value of a share of the bill is $12.57, to the nearest penny.*

Instructions for Exercise 1.2

Answer these questions about this approach to testing the calculator:

1. Is each of these activities -- (a) through (k) -- justified?

2. Together are they sufficient to show the calculator works adequately?

3. If not, what has been omitted that also should be tested?

4. Are the activities listed in approximately the right order?

5. What do we need to do, if anything, in order to transform these activities (which effectively are test requirements), into suites of executable test cases?

6. If you were able to open its case and gain access to the insides of the calculator, what could this tell you, if anything, beyond the activities above? What additional tests would you do if you had internal access?

7. How do you suppose that the manufacturer probably tested the calculator in the first place? Is this level of effort warranted in the re-test after has been damaged?

8. The list of test activities assumes that the calculator can be powered from a battery or a wall connection. What if a calculator instead is solar powered only, with no battery or wall connection. How would we test the solar power supply?

9. What if the calculator is designed to work with floating point numbers as well as integers and fractions? A floating point number is define tbd.

10. Let's say the calculator has passed the tests and you have decided to keep using it. Is there any point in keeping the suite of test cases around, or have they become obsolete or unnecessary clutter and are better discarded?

11. Let's say that instead of a stand-alone machine in its own electronics, keys and casing, the calculator is a software-only product which runs on personal computers using the Windows operation system. How would you modify the test strategy for the software-only calculator?

12. If you do not have the information you need to answer these questions, what else do you want to know in order to complete this exercise?

The next exercise, 1.3, presents suggested answers to these questions for you to compare and critique.

The Nature of Test Case Design

Questions and their Value

We can view a test case as a question about the behavior of a feature or a system. In this view, test cases are essential—it is unlikely we will successfully build or maintain software without an effective set of test cases. Nevertheless, we can have great test cases, mediocre ones or worse – dangerously misleading and grossly inefficient ones. How do we tell the difference?

A high-value test case reveals a significant amount of useful information about the system being tested, and provides this information quickly, reliably and cheaply. We have sufficient interest in the answer to justify the effort to develop, run and evaluate the outcome of that test case. By contrast, a low-value test case provides little usable information about the system being tested, or is too time-consuming or expensive. Test cases like these are not worth the effort. In this book, we will examine examples of high-value and low-value test cases and differentiate between them. (Tbd - where to go / reference for this?)

The stakeholders in a system typically have a number of "What happens if …." questions about the system. For example, what if a gambler, who plays the slot machines we are testing, waves a strong magnet in front of a machine as its wheels are spinning? We can identify test cases by asking "What if?" questions. The attempt to answer these questions, and to reach a consensus on the answers (the test outcomes), generally leads to more "What if?" questions and thus more test cases.

Developing Test Cases

Test case design is the process of analyzing the purpose, decision logic and characteristics of a feature, determining how to test it, and developing an appropriate set of test cases to do the job. In a classic joke, the project team leader says to the software engineer: "You start writing code while I figure out what we are supposed to deliver." Testing without first preparing test cases is equally ineffective. Glenford Myers said it well: "Testing is an extremely creative and intellectually challenging task. The creativity required in testing a large program exceeds the creativity required in designing that program." <<[ref. to Myers book—tbd.]>>

A *feature* is a capability of a system, a service which we want the system to provide to its users. For example, a feature of a billing system displays a customer's outstanding balance. Another feature prints a reminder notice for any customer who has not paid on time. One test case for this feature deliberately creates an account for a customer named John Brown, who currently is overdue by 33 days. When the test case runs, the tester observes if the system prints the reminder notice for John Brown as expected.

A *non-functional requirement* does not focus on how a feature works: there are testable system requirements for other aspects, such as the system's performance, robustness, security, database integrity, compatibility, usability, installability, maintainability and so on. We will not address these test cases in this book, except tangentially.

Placing Test Cases in Context

On a system development project, the system requirements definition, design and programming are distinct activities. The same is true in testing: on most projects we have distinct deliverables like test plans, test requirements, test suites, test cases and test specifications. It is worth a moment to distinguish among these deliverables. I already have introduced test requirements and test cases, so here I want to describe the remainder of these deliverables.

The *test plan* provides the general direction for a test project but not the details. It includes the test objectives (what the test team is seeking to accomplish), the scope of the work – including what will *not* be tested, the resources needed for testing, and how they will organize our testing efforts. Test planning is part of the overall system project planning. For examples of test plans, see my companion book entitled: "Developing Software Test Plans" << to be published >>. A test requirement describes what to test, while a test case addresses specifically how we will fulfill that requirement. The test requirements usually are part of the test plan, but the specific test cases needed to fulfill these test requirements are not included in the test plan except for small, simple test projects.

A *test suite* is a group of related test cases. Because of the sheer volume of test cases needed for many systems, organizing them into suites helps make them manageable. Test cases can be clustered together based on any commonality which makes sense to the testers, e.g., by type of feature, test type (e.g., Web page look-and-feel vs. database), subsystem, test platform, test project, business area, and so on.

Some people use the term *test specification* to mean the documented form or description of a test case. The variation in the formality and size of test specifications is very wide, ranging from nothing written down to several pages of documentation per test case. Many manual test specifications are no more than a quarter-page in length, though ones which extend for five pages of longer are not uncommon.

Depending on whom you ask, the terms requirement and specification mean the same thing, or else they mean different but related concepts. In organizations which differentiate between these terms, a requirement is something a system must do, while a spec describes how the system does it (or will do it). Similarly, a *test requirement* is something a test project must do, while a test spec describes how to do it.

Other terms which testers use widely are *test library or repository* (a collection of test suites which spans multiple projects and multiple areas of the business), and *test component* (a sharable, re-usable, interchangeable part of a test case).

The Test Work Flow

Testers usually first plan the overall test project, identify test requirements and then design the test cases, though these activities can be intermingled. A test case designer's individual work assignment includes a statement of purpose, which is the test requirement. The source of this requirement may be a documented test plan or the test team leader's verbal directive. If there is not a suitable statement of purpose, the tester needs to define it.

We saw earlier that test case design is the process of turning the test requirement into a suite of test cases. While the test plan or team leader's directive provides the intent and framework, the test cases are where the "rubber meets the road"—where the testing actually will get done. Sometimes there is no explicit test plan, particularly on smaller, more informal or rushed projects. In this situation, testers develop test cases without the guidance of a documented test plan which is distinct from other project documents. As a substitute, they use the implied but undocumented test policy and approach, implied test requirements and overall system objectives.

The implementation of a test case includes reviewing the feature's expected behavior, designing the test case, planning the test environment, writing test procedures, automating the test case, generating test data, and conducting trial runs to check the test case works. By the phrase "the test case works", I mean that it runs as expected if the feature itself is working correctly, finds bugs related to the feature being tested, and doesn't produce false results. Activities related to test case design include:

1. ***Review the Test Requirement (if available)***. Ensure that you, the test case designer, understand the intent and scope of the suite of test cases, and any special considerations such as the re-use of existing test cases.
2. ***Define the Test Requirement (if it is not available)*****:** Determine what to test, from the feature specifications, test plan or other sources.
3. ***Design the Test Case*****:** For each test case needed to fulfill the test requirement, define the initial conditions, inputs, test procedure and the expected results.
4. ***Build the Test Case*****:** Create the test data or databases needed for this test case, and automate the test case if appropriate.
5. ***Execute the Test Case*****:** Run the test case using the system in the test environment, and capture the outcomes.
6. ***Evaluate the Test Result*****:** Make a pass / fail determination of the outcomes.
7. ***Refine the Test Case.*** Improve the test case, using what we have learned by running it.
8. ***Maintain the Test Case Library:*** Catalog the test case and archive it re-use in regression testing, or for possible future adaptation and re-use for other purposes.

When do We Develop Test Cases?

Tbd -- Xxxx

Factors which Influence Testing

Failures and Defects

We will be using the term “failure” and also its synonyms, which according to the Merriam Webster’s Dictionary and the Oxford English Dictionary include: problem, discrepancy, incident, symptom or error. We also will use the term “defect” and its synonyms: bug, fault, flaw or cause. It’s worth a moment to define these terms and distinguish between a failure and a defect.

A *failure* is any incident with negative consequences, where the system, subsystem or system component deviates from its expected behavior. For example, a nuclear reactor shut down, which is expensive, because of a false warning about an earthquake. If the system had worked according to specifications, it would have been able to differentiate between real and false alarms. This definition of a failure assumes that the expectation of the system’s behavior is reasonable in itself. The comp.risks news group lists thousands of incidents of system failures, including this story about the reactor.

A *defect* is the specific cause of a failure. If the defect is fixed, the failure cannot occur. An example of a defect: 'X = Y' was coded instead of 'X = ABS (Y)'. That is, X is supposed to be set to the absolute value of Y, regardless of whether Y is positive or negative. The absolute values of +37 and -37 are both simply 37. This defect caused the nuclear reactor shut-down.

The terms are imprecise and not everyone will agree with how I use the words failure, defect and other terms. I want to avoid hair-splitting, but imprecise terminology in the testing area has caused lots of heartburn. There is no universally accepted terminology, but the way I’ll define terms in this book are as much or more in the mainstream of common usage as any other terms. People sometimes use the word “defect” when they mean “failure”, and vice versa, but the meaning is usually clear from the context in which they are speaking

The relationship between failures and defects is complicated: though at first thought there appears to be a direct one-to-one, cause-and-effect relationship of defects to failures, actually we cannot equate defects with failures or with system reliability. The same symptom (the observed behavior in failure) can be caused by many different defects. One defect may cause hundreds of different failures within seconds, as the same defect is triggered repeatedly. Based on data collected from its problem reporting systems, Microsoft has discovered that one percent of the software errors cause 50 percent of the crashes which its users experience.

By contrast, many defects are hard to trigger. Many, perhaps most, lurk in systems indefinitely and are "benign" defects, i.e., they are never exercised or never cause a noticeable failure. According to Ed Adams of IBM, most defects in high reliability systems cause failures rarely—the average defect found after delivery in large mainframe systems has an MTBF (mean time between failures) of 900 years, and 35% of defects have an MTBF greater than 5,000 years. (There is a controversy about how to apply MTBF to software, which is beyond the scope of this book.) In addition, a low incidence of failure occurrences does not mean we can stop looking for

the underlying defect if its likely consequences are high when it does cause a failure. << (Reference to Adams here.) >>

It is important to understand that testing finds a symptom, not the defect, when a test case fails. Diagnosis or debugging is the crucial activity which derives the cause (the defect) from the observed effect (the failure), and this diagnosis normally is the software engineers' responsibility. Diagnosis is not part of testing, though testers often partly isolate a problem in order to expedite its diagnosis. It also is a good idea to use the term "symptom" to describe a system's behavior until the failure has been confirmed by re-running and replicating the test case's behavior; an innocent-until-proven-guilty approach. Without confirmation the incident is not officially a problem, and debugging time cannot be assigned to resolve it.

There is a lot we can learn from the patterns of failure we see, and a lot more we don't know. For example, if failures happen frequently in a system this implies many other defects remain undiscovered which have not yet caused failures. The reason is similar to the rationale for fishing where the fish are biting. If we fish in a spot where we catch lots of fish, there probably are more there, though we have no guarantee. This phenomenon is called defect clustering and we will discuss as part of risk-based testing in Chapter tbd. By contrast, some theories and studies do not support the assertion that finding many failures mean that lots more defects remain to be discovered. Defect prediction methods which use Raleigh curves, for example, assume a finite number of defects in a system, so the more failures that have happened and the more defects have been discovered to date on a project, and the less the number of defects still to be found.

Determining if a particular behavior is a failure and assigning its level of severity can be sensitive because of ingrained biases, political baggage and finger-pointing. People may argue about whether a claimed failure is actually a matter of user misunderstanding or is an undocumented enhancement. As the tester's experience grows, he or she will be able to better discern the difference. Like great art, some people cannot define a failure but say "I know it when I see it." To place this statement in context, though, most system failures are not controversial.

The Contribution of Independent Testers

Independent testers check others' work, not their own. In unit testing software engineers (authors) check their own work, but most system testing employs people who are not the authors of the software being tested. Independent testing is discredited in some circles. The arguments against it include: (1) a weakened accountability of software engineers for the quality of their own work; (2) inefficiency of the added tester headcount; (3) slower delivery times because of the bureaucratic interplay of the developers and testers; and (4) encouragement of an "us-versus-them" adversarial relationship. There is validity in these claims, but they do not seriously challenge the importance of independent testing. We need an appropriate balance of both author self-testing (by the "insiders") and independent testing (by the "outsiders") to achieve quality, because the traditional strengths of these two approaches complement each other. The table shows the patterns of strengths and weaknesses found in many systems projects:

	In-Depth Knowledge	Detachment
Insiders	**High**	**Low**
Outsiders	**Low**	**High**

The Insiders

The traditional strength of the insiders is their in-depth knowledge of the system: they currently are building it or are working with the system day-by-day in on-going maintenance. These insiders know the system better than anyone else, and this knowledge is important for effective testing and quality assurance.

The traditional weakness of the insiders, though, is a lack of detachment, This lack of detachment can occur for three reasons, First, there is the so-called proof-reading effect, The proof-reading effect occurs when we are very close to something, We cannot proof a letter we have been working on intently and revised several times, because we do not see the actual words on the word processing screen -- instead we see the words we think are there. Second, the lack of detachment results from too much ego involvement, System design and programming are personal and intimate activities. They also are intense and may require an obsessive focus to be successful. Any work activities which are described by words like intimate and intense are susceptible to a loss of objectively. If we are software engineers, we can reach a point where we regard a software component as our creation or "our baby", At this point, the software engineers have lost objectivity. Third, many software engineers are optimists, which may add to the lack of objectivity. In the words of Glenford Myers (with italics added by me): “Most programmers cannot effectively test their own work because they cannot bring themselves to form the necessary attitude: *the attitude of wanting to expose errors*."

The Outsiders

The outsiders, by contrast, usually have little difficulty in bringing a fresh, independent perspective to the validation of the system, a “breath of fresh air”, They may well have their own sets of biases, but usually these biases are different from the biases of the insiders, The outsiders bring a quality, operational and business perspective to balance the insiders’ technical perspective of the system.

The traditional weakness of the outsiders is lack of in-depth knowledge, They can never attain the same levels of immersion that the insiders have experienced. Nothing is worse, as an outsider, than being presented with a thick requirements definition document for an unfamiliar system, and being told to start testing -- by the end of the week, There is no way to learn the system adequately in the available preparation time, so the outsider is forced to perform a superficial test based more on guesstimates than knowledge.

In conclusion, we need a significant involvement of both the insiders and the outsiders to ensure quality. The insiders (software developers, maintenance programmers, database administrators, vendor technical staff, etc.) are the first line of defense. They are accountable for testing each component of their own work. However, because of the possible lack of detachment, it is also important to have the overall, integrated system tested independently by outsiders (system testers, users, auditors or consultants.)

The Value of Independence

THIS SECTION REDUNDANT – REPEATS IN CH. 3

We grapple with the problem of how to gain a sufficient understanding of a feature in order to test it competently. One solution is for the person who specified the behavior of the feature to develop the related test cases. Supporters of this approach take the position that, by having the same person develop both the specification and its test cases, the clarity of the documentation becomes a non-issue. Supposedly it does not matter if the feature documentation is unclear, the author knows what he or she meant to say and will test accordingly. In addition, the author already understands the feature, while someone who it new to it could be more error-prone in developing test cases. With this approach, we circumvent the possibility of the spec writer and the tester mis-communicating.

This solution, to have the same person specify the feature and test it, is worse than the problem. The person who specified the feature's behavior will test according to his idea of the expected behavior. If he mis-specified, then he is going to develop the wrong test cases, and approve the implementation of the feature if it conforms to the specified—but wrong—behavior. As an example of mis-specified behavior, imagine a situation where the users expect "Add" but the developer specifies "Subtract". This little mix-up between adding and subtraction almost bankrupted the treasury of the nation of Chile a few years ago. << reference >>. Most mis-specifications are much more subtle – and thus harder to see – than adding instead of subtracting. This lack of detachment is frequently exacerbated by ego involvement, an inappropriate sense of ownership and the proof-reading effect. People can't proof their own work—they are so close to the trees they can't see the forest.

In my experience, we should separate responsibilities and avoid the practice of people testing according to their own specifications. People still are responsible for their own specifications (i.e., testing in the sense of reviewing and validating the spec is correct), which is not the same thing as developing and running test cases for the feature. Many organizations which do not separate responsibilities (i.e., the person who specifies the behavior also develops and runs the related test cases), are satisfied with their test effectiveness and are not about to change. I advise these organizations to at least perform peer reviews of the test cases – more heads are better than one, and to conduct project post mortems – more bugs may be slipping through the testing than they realize.

Separating responsibilities does not help, though, if someone else other than the author tests the feature but has to test using weak specs.

Testers explore the context and problem domain to develop test cases independently, while software engineers tend to work in the solution domain. Testers are more likely to concentrate on understanding the underlying needs rather than how the software engineers have tried to solve them. We need to know about the implemented solution too in order to test the feature, but (<<or because?>>) this focus on the problem leads to better test cases.

Even this suggestion is optimistic. Few people are good testers. According to a senior tester at a large software vendor, only 10% of the test cases that their project managers suggest to the testers prove good enough to use. Those same project managers usually identify fewer than half of the interesting (potentially fruitful) areas to test. In his opinion, relying on the spec creator to create all the test cases, and using all the test cases that person creates, will result in testing disaster. Given an unclear spec, why should we assume that the author has any clearer mental concept of the feature? Lack of clarity implies a lack of clear, consistent thought.

Many organizations assign people as business analysts and system-level testers at the same time. Unless we can change this practice, we need to find ways for people to "proof their own work". Quality also improves if the test cases are independently reviewed by others.

Contributing to the Knowledge Base

As we become enlightened, it is a good idea to add our contributions to the common base of knowledge about the system—in other words, document our knowledge in a form where others can share and use it. We don't want to use a test case once, for example, and then discard it. Instead, it can have an on-going life -- maintained in a test case library, kept current with regular updates in coordination with the system updates, and fruitfully re-used in regression testing for years. In Chapter (tbd), we will discuss how to build the knowledge base, including topics like the aging of test cases and how to design them for re-use.

Exercise 1.3: Are You a Born Tester?

(Allow 10 to 20 minutes for this exercise.)

If you romped through the last exercise then you are a born tester. Review the suggested answers to the questions and see whether you agree with them. If not, you probably made different assumptions than I did. Can you see where you and I differ in these assumptions?

1. Is each of the recommended activities for testing the calculator justified?

Yes, each activity on the list is justified because it is necessary, feasible, and cost effective. Each is necessary: you would not want to keep the calculator if any one of the test cases on the list does not work. Each is feasible: all these tests can be run, and without specialized tools or skills. Each is cost effective: it is straightforward, easy and quick, and should yield useful information about the health of the calculator.

2. Together are they sufficient to show the calculator works adequately?

I cannot answer this question based on the information we have. The answer depends on three factors: (a) what we mean by the word "adequately", (b) how extensively we perform each type of testing on the list, and (c) the assumptions we are willing to make.

I will start with assumptions. One of the test activities states:

> *j. Test the calculator for its usable operating duration on battery power.*

The test case description does not mention how many times the operating duration has to be measured, and most people assume once -- we fully charge the batteries, then use the calculator until the batteries run down, and record the elapsed time. There are two unstated assumptions behind this: (1) the discharge time is reasonably consistent, so that the duration we measure one time is about the same as what we'd find if we repeated the test case. Or (2), the duration varies widely from charge to charge, but we don't care because the minimum of four hours is always exceeded.

Whether a calculator works adequately for our purposes depends on how we use it -- its context. If you are a poet who occasionally calculates meal tips where the risks associated with an incorrect calculation are low, then the seven listed activities are sufficient to test adequately. On the other hand, if you are a doctor who calculates life-critical numbers. By running the same suite of test cases that is reasonable for the poet, we may be taking inappropriate risks by under-testing for the doctor

The list tells us what to test, but not how intensely to test each type of activity. Each activity is a test requirement which can be translated into one or many test cases. Should we test every possible way the calculator could ever be used? That would take a long time and many testers will argue it is unnecessary. Let's say we test the addition of [13,287 + 764] and separately after re-setting so [844 + 3]. If these two calculations work, we're probably willing to assume that any other addition of two numbers also works and thus does not need to be checked. The number of combinations of two numbers is astronomical, even within the limited domain of the calculator, and the majority of combinations will never be exercised within the life of the calculator.

What about the numeric keys? If we check two which work, let's say the 3 and 8 keys, we probably are not willing to assume all ten numeric keys work. With only ten keys, though, we can quickly check them all – we can know if each numeric key works and do not have to assume. Since each numeric key is used frequently it is a show stopper if any one does not work.

3. If not, what has been omitted that also should be tested?

Errors of omission tend to be hard to detect. We cannot prove that there is no relevant test activity has been omitted from the list. But I have not surfaced any significant oversights through due diligence and peer reviews. Since we are given only the test requirements which are general directives of what to test, not the test cases, we cannot judge whether there are enough test cases for each activity.

4. Are the test activities in approximately the right order?

Chapter 1
An Introduction to Software Test Case Design - Exercise 1.3: Are You a Born Tester?

Yes. We want to reveal as much information about the health of the calculator as quickly and easily as we can. This means we should proceed from the test cases which are likely to be the most fruitful and easiest to run, to the less fruitful and harder to run. In addition, there is a natural flow among some of the activities based on their interdependencies. For example, we can't check if the display still works if we can't power up the calculator, so we should check the power first.

5. What do we need to do, if anything, in order to transform these activities into executable test cases?

We can view each of the listed activities as a test requirement. This requirement gives the purpose for a test case or a suite of them. We form test cases to fulfill a requirement, by defining their initial conditions, inputs, test procedures and expected results. Consider this activity from the original list:

> *e. Test the basic arithmetic features: add, subtract, multiply, and divide.*

As a test requirement this statement is broad, so first we decompose it into manageable pieces: test the add feature, then test the subtract feature, etc. We can profitably decompose these pieces further and create a top-down hierarchy. In this example of a hierarchy, I have omitted some of the constituent pieces for brevity:

1.0 Test the add feature.
- 1.1 Test the addition of integers only (no fractional numbers).
  - 1.1.1 Positive integers only.
  - 1.1.2 A mix of positive and negative integers.
    - 1.1.2.1 Two integers only, one positive and one negative.
      - 1.1.2.2 More than two integers, with some positive and some negative.
        - 1.1.2.2.1 More than two integers, with a mix of positive and negative ones, and where special keys such as the clear key are not used during the calculation.
        - 1.1.2.2.2 More than two integers, with a mix of positive and negative ones, and where special keys are used during the calculation.
- 1.2 Test the addition of fractional numbers.
  - 1.2.1 Fractions with more than ten digits after the decimal point (*).

2.0 Test the subtract feature.

… and supporting details go here.

(*) There is an upper limit on the number of digits which can be entered after the decimal point, based on the calculator's designed capacity. Let's assume this limit is ten digits for the calculator at hand. If we know that users almost never want to calculate with more than ten digits precision, and are reasonably confident that it is physically impossible to enter more digits than the limit, we'd give minimal attention to the ten-plus digit scenarios. In a thorough test, we might include one test case where we try to add ten-plus digit numbers (#1.2.1), to see what happens.

Once we have decomposed the test requirement to the level where any more pieces do not make sense, we can derive test cases for each bottom-level one where we think the testing is justified. A test case for #1.1.2.2.1 is:

Test Case Number and Description (*)	Test Requirement (**)	Initial Conditions	Input Values	Expected Results
1. Test a mix of multiple integers.	1.1.2.2.1 More than two integers, with a mix of positive and negative, and where special keys are not used.	Calculator memory and display are both clear (set to zero); calculator has power.	333; 668; -1002; 1	0 (zero); because of rounding errors, any number between plus or minus 0.001 will be accepted.

(*) The test case format does not include a column for the test procedure, because it is obvious – everyone knows how to use a calculator.

(**) This is the description of the requirement which the test case fulfills. This requirement is wordy; normally we'd summarize it and be circumspect.

In practice, many people bother documenting test cases like this one for #1.1.2.2.1, since the calculator test is straightforward and low risk. But we do want to go through an adequate thought process in analyzing the situation. There are enough nuances to overlook that it is a good idea to at least informally jot down the test cases – not to document them formally for posterity, but to use the jottings as an aid to the mental effort.

6. If you were able to open its case and gain access to the insides of the calculator, what could this tell you if anything beyond the activities above? What additional activities would you do if you had access?

With access to the calculator internals, we can do at least three additional types of testing. In increasing order of effort, these are: (1) visually examine the internals for obvious problems such as a disconnected wire; (2) tinker and explore, perhaps by poking the internals gently with the tip of a pen; and (3) check the circuitry with electronic tools. If we have the knowledge, tool and motivation, we could use a voltmeter to check if the voltages at key points in the circuitry meet the levels recommended in the manufacturer's service manual.

7. How do you suppose that the manufacturer probably tested the calculator in the first place? Is this level of effort warranted in the re-test after the calculator has been damaged?

Chapter 1
An Introduction to Software Test Case Design - Exercise 1.3: Are You a Born Tester?

The manufacturer presumably tests large volumes of new calculators every day, and uses sophisticated test equipment. Generally the manufacturers' testing is highly automated, works at high speed, and utilizes expensive special-purpose testing machines. Most persons who test a dropped calculator have no access to this test environment, and no justification to build these kinds of facilities. How does the situation differ if one of the manufacturer's test staff drops a calculator on the test lab floor? Since the test equipment is right there and the tester knows how to use it, re-running the dropped calculator through the standard set of tests makes perfect sense.

8. What if a calculator is solar powered only with no battery or wall connection. How would we test the solar power supply?

The information that the calculator has no battery or wall connection influences the testing. If a calculator has a battery or wall connection, the viability of the solar power supply is less significant -- we can use the calculator even if the solar source does not work. On the other hand, if a calculator's only power source is solar, then the question of whether the solar power works is critical to deciding whether to keep that calculator -- and a more thorough test is justified.

We'd need to check that the solar power supply can perform three tasks: charge up reasonably quickly to a level sufficient to run the calculator for a reasonable period of use, hold that charge for another reasonable waiting period, and run the calculator. For the first of these, we probably have some idea of how long it normally takes the battery to charge, and we'd check to see if this time is the same as before. We'd want to test this in both bright and dull light, since the charging times will be quite different. We'd also need a way to assess how fully charged the calculator is. If the calculator takes a full day to charge, but only retains enough energy to run for five minutes, it practically is unusable. Finally, we would want to add precision to the test cases by replacing phrases like "reasonable period of use" with specific values. I will leave the details of the other test cases (to determine if the solar power supply will hold the charge and run the calculator), for you to work out.

9. What if the calculator is designed to work with floating point numbers as well as integers and fractions?

Tbd – computation of upper limits on rounding errors, numerical analysis, sequence of ops (add / divide etc.) affects accuracy, conversions among data types.

10. Let's say the calculator has passed the tests and you have decided to keep using it. Is there any point in keeping the suite of test cases around, or have they become obsolete or unnecessary clutter and are better discarded?

Answer goes here.

11. Instead of a stand-alone machine in its own electronics, the calculator is a software-only product which uses the Windows operation system. How would you modify the test strategy for the software-only calculator?

Answer goes here.

12. If you do not have the information you need to answer these questions, what else do you want to know?

Other desirable information -- which may influence the test approach and thus is worth knowing -- includes:

> How the calculator is used, and what level of precision is required in the calculations.
>
> Its age and history – if the calculator is obsolete previously was unreliable, we may not bother to test it but simply discard it.
>
> The cost and ease of replacement. Calculators are generally cheap and plentiful. If we are in a position where this is not true, then we have more incentive to test the dropped calculator rather than immediately discarding it.
>
> The cost and ease of repair. If we have no way to fix the calculator when we encounter a problem, we should plan to abort the testing and abandon the calculator when a show-stopper bug is discovered.
>
> What problems the user can live with, if necessary, and which she cannot.

Do You really want to be a Born Tester?

We know what type of person makes a good tester. The born tester thinks that testing is great: he or she gets paid to beat the heck out of someone else's work. He has the motto: “if it is not broken, then break it”, and has an uncanny ability to sniff out the most embarrassing bugs. Wants to save the world, if only she could get the software engineers, managers and users to listen to her. Regards finding a defect as vindication that he was right all along. Does not understand why nobody wants to sit at her table in the cafeteria. Knows that the Queen in “Alice in Wonderland”, who every morning practiced believing “as many as six impossible things before breakfast”, worked in marketing for a software vendor. Likes to test software engineers’ logical skills by telling them: “When I was young, I walked five miles through the snow to school each day -- and it was uphill in both directions.” And as a child, liked to take things apart but couldn't ever get them back together again. Do you identify with these statements? Then you are a born tester.

Exercise 1.4: Reviewing and Assessing Your Understanding – Part 1

(Allow 10 to 15 minutes for this exercise.)

The purpose of this exercise is to give you an opportunity to test your understanding of the ideas in this chapter, and consolidate your knowledge. Concise answers, with a few lines in response to each question, are fine.

1. What are the goals of testing?

2. What is the purpose of a test case, and what are its major components?

3. What is test case design?

4. What is the difference between a defect and a failure?

Answer to Exercise 1.4

1. What are the goals of testing?

Testing is an organized, methodical effort to validate that a system works as expected, find discrepancies, and provide information about the state of the system. Testing enables us to evaluate whether a system will meet its goals, the users will be satisfied with the service they receive, and the system operation will be basically trouble-free and reliable. Within the overall framework, it is important to realize the different parties involved in a project – such as managers, testers, developers and users – may have different perspectives, priorities and testing goals.

2. What is the purpose of a test case and what are its major components?

A test case checks one particular situation or condition of the system being tested. The components of a test case include the test case identification, the pre-conditions to be met, the inputs needed to drive the test case, the procedure to run the test case, and the expected results.

3. What is test case design?

Test case design is the process of analyzing the purpose, logic and characteristics of a feature (or an aspect of the system's behavior which is not associated with one particular feature), determining how to test it, and developing an appropriate set of test cases to do the job.

4. What is the difference between a defect and a failure?

A failure is any incident with negative consequences, where the system or component deviates from its expected behavior. A defect is the specific cause of a failure.

Chapter 1
An Introduction to Software Test Case Design -