A Positive View of Negative Testing

Abstract:

A practitioner's overview of negative testing, dealing with tests designed to make the system fail, and tests that are designed to exercise functionality that deals with failure. Details the overall aims and management of negative testing, and describes a variety of techniques used to select, derive and execute negative tests.

Author: Error! Bookmark not defined.**, Workroom Productions Ltd.**

James is an independent test strategist, based in London. He's spent more than 10 years in software testing and has been the principal consultant at Workroom Productions since its formation in 1994. As a consultant, James has encountered a wide range of working practices, from rapidly evolving Internet start-ups to traditional large-scale enterprises.

James is a director of The Manual (http://www.the-manual.org/), a not for profit organisation dedicated to gathering and publishing basic skills.

1. Contents

1. Contents 2
2. Introduction 3
3. Aims of Negative Testing 4
4. The management of negative testing 5
5. Test selection strategies – and when to stop 8
6. Techniques to derive test cases 9
7. Semi-scripted testing 11
8. Ways of approaching test execution 12
9. Diagnosis and logging 15
10. Testing functionality that copes with exceptions 16
11. Criticisms of Negative Testing 18
12. Conclusion 19
13. Appendices 20

2. Introduction

This paper is a practitioner’s overview of negative testing. It is based on experience and observation, rather than a formal study or a comprehensive survey of existing literature, and its intention is to provoke thought and discussion, whatever the reader’s level of experience.

Some approaches to testing recommend that the purpose of the test team is to show, to a level of confidence, that a system works. Negative testing does the opposite; it seeks to show that software is not working, to dig and probe through the external weaknesses until it finds just the right way of making a bad situation worse, to hurt the system and watch it heal, or die.

The two approaches are complementary, but have entirely different aims.

3. Aims of negative testing

The British Standard definition of negative testing in BS7925-1 [2] is taken from Beizer [3] and defines Negative Testing as *Testing aimed at showing software does not work*. This can lead to a range of complementary and competing aims;

- Discovery of faults that result in significant failures; crashes, corruption and security breaches
- Observation and measurement of a system's response to external problems
- Exposure of software weakness and potential for exploitation

While a fair definition, it is far from being generally accepted. 'Negative Testing' is a term that is re-defined site-by-site, and sometimes even team-by-team. A common way that practice differs from the (British Standard) definition is that it includes tests that aim to exercise the functionality that deals with failure;

- Input validation, rejection and re-requesting functionality (human input and external systems)
- Internal data validation and rejection
- Coping with absent, slow or broken external resources
- Error-handling functionality i.e. messaging, logging, monitoring
- Recovery functionality i.e. fail-over, rollback and restoration

This paper will deal with tests designed to make the system fail, and tests that are designed to exercise functionality that deals with failure.

Strategic Directions

There are often further aims of negative testing that do not set the scope of the activity, but nevertheless dictate the choices made in designing and running the tests. These can include:

- Prompt exposure of significant faults
- Learning about function through the study of dysfunction
- Verification (and possible enhancement) of risk model* used to prioritise testing
- Documentation of common failures, characteristic symptoms, and running fixes

**See below for further information on risk models*

4. The management of negative testing

Testing as a whole is reactive, open-ended, and hard to value before it has been done. Much of the blame for this can be laid at the door of negative testing – which is why, although it is an integral part of many different approaches, it is sometimes explicitly excluded from the scope of testing.

The following sections describe a way to budget, plan and support negative testing, and discuss the ways that scripted and exploratory approaches differ.

4.1. Budgeting and estimation

Estimation is an acquired skill based on a wide range of assessments outside the scope of this paper, and the process of estimation may be more valuable than the estimate itself. Estimating the time and money needed for negative testing needs a three-part approach, where each part has its own balance between planned costs and the potential to invest in new testing when necessary.

4.1.1. Scripted tests and up-front investment

Some negative tests can be derived from formal techniques. These are often scripted and can be relatively simple to estimate in the usual granular fashion. Your estimates should reflect that 'negative testing' is not a simple, single phase, but spans – and could be included in – a range of activities.

Negative testing can compromise other tests, and may need a separate environment and specialised support. It can be greatly enhanced by custom tools and testability enhancements, and may need skills that do not currently exist in your test team. Scripted tests will lead to an understanding of some of these requirements, and should enable you to consider your investment in extra hardware, licenses, development and training.

4.1.2. Primary negative testing

This is negative testing that concentrates on failure modes, observation of failure, assessment of the risk model and finding new, unknown problems. Formal techniques and checklists may allow some granular estimates, and failure mode analysis can indicate the test lab resources needed. Depending on planning and budgeting elsewhere in the project, it may be possible to find an overlap with analysis done for design and unit testing.

A proportion of the estimate will need to be based on the available resource and the expected return – bearing in mind that if resources are limited, an opportunity taken to do a negative test is an opportunity lost elsewhere.

4.1.3. Secondary negative testing

This is testing that is found to be necessary after the start of the test cycle, and includes tests to find failures once a new exploit or weakness has been found, and diagnostic negative tests. You should also consider contingency for primary negative testing; tests in new areas, and tests in areas with new risks.

Very little of this testing can be pre-planned. Like regression testing and re-testing, risk and available resource will limit the tests in which you can invest. It may be possible to derive an estimate from prior projects, or plan for flexibility based on (for example) the bug density found as the project moves on.

On a project with fixed budget or deadlines, these tests will replace others. Working out which tests to leave out needs some kind of assessment, and you should allow for time and resource to support this.

4.2. Planning

Negative testing does not have a well-defined position in a waterfall or iterative process, and so is generally not seen as a distinct phase. Rather, it is a way of classifying some tests, and so directing part of the test effort.

Typically, negative testing is most often used during system and integration testing and is designed and performed by test professionals within those test teams.

Negative testing is an open-ended activity. As seen in the estimation section above, some elements of negative testing cannot be planned in any detailed way, but must be approached pro-actively.

4.2.1. Activities

Negative testing activities include:

- Formal design and execution of negative tests
- Designing and performing exploratory negative tests
- Assessing the risks and possible exploitations of a weakness
- Diagnostic tests and related problem logging
- Communication within the team of exploits and weakness

4.2.2. Scheduling and staffing

The ability to break software is an important and hard-won skill, and so career testers make the most effective negative testers. Their skills cannot be well used before the system is integrated into some sort of working whole, and should be used soon after. However, negative testing has its place in all phases of a project.

The following table sets out some of the scheduling and staffing considerations, broadly split by phase.

Phase	Approach	Staffing
Requirements analysis, system design and unit testing	Formal techniques can be used to derive closed sets of negative tests. These should match the exception handling and validation functionality in the code Analysis to derive the tests can itself expose important flaws.	Designers and coders of a system will often arrive at inventive ways of causing a failure, and can be very effective at spotting exploitations. However, their understanding of the strengths of the design may reduce their ability to spot weaknesses; negative testing seeks conditions that have not been considered during design and coding. Test analysts may provide a valuable service to designers.
System testing, integration testing	Execution of scripted tests, and less-scripted primary and secondary tests React to weaknesses found Assess risk and potential for exploitation Diagnosis	Scripted testing may be done by less experienced testers and partly automated Exploratory testing done by experienced testers
User acceptance testing, beta testing	Allows fair assessment of user-facing error messages. Can reveal up new mis-use cases. Valuable input to further negative testing, and can help to improve the test designers' understanding of failure modes and weaknesses. May allow a near-live system to be pushed beyond its performance limits, or tested for fail-over or recovery (tests may need to be scheduled away from general UAT)	Users may find a significant number of failures as they use the software for the first time. However, these are not found by design, and the users will typically not stray from a working path once it has been found in UAT. Restricted use and internal error handling may reduce opportunity for UAT/beta testers to observe problems. Specialist testers needed to put the system under stress.

4.2.3. Exclusion

Negative testing is sometimes explicitly excluded from unit testing and user acceptance testing. However, its aims are useful throughout the life of a project, and any exclusion should be based on a sound assessment of risk and supported by an effective re-assignment of the work. It is all too easy for contracts and strategies to exclude it in order to shift responsibility, and for planners to ignore it because it is hard to plan. If it is explicitly excluded, it helps to have some sort of definition that seeks to limit its open-endedness, as it can become a bucket for all the tests that nobody wants to take responsibility for.

4.3. Support

Negative testing can be helped by good tools, and can be hindered by a poor political environment. However, the most important element is to support the task by having appropriate skills within the team, and encouraging the team and its members to improve those skills.

4.3.1. Tester skills and support for ongoing improvement

Although experience is important to negative testing, any test team can get better at negative testing over time, as members learn about characteristic weaknesses and exploitations, and gain an understanding of the way that the system works and of the breadth of tests that can be applied. This improvement needs to be encouraged if it is going to be sustainable. Managers need to take an active role in enabling the following:

- Skills transfer within the team, particularly the more ‘intuitive’ skills from experienced and effective testers. Pair-testing allows the skills, models and strategies of ‘intuitive’ testers to be observed at close hand by other members of the team, who may absorb those skills, or use them as a basis for further techniques.
- Knowledge transfer within the team, particularly with regard to weakness and exploitations.
- Knowledge transfer into the team from designers and coders regarding the implementation and architecture of the system
- Use of ideas, perspectives and analogies from outside the project; books, analogies, training and conferences.

4.3.2. Technical support

Technical support falls into four areas;

- Negative testing, like performance and stress testing, may need its own environment to avoid compromising other test work
- Tools can make negative testing simpler and faster, and can allow perspectives that would be impossible without a tool. However, such tools also tend to be custom-built – and budget may need to be earmarked early in the project
- Testability requirements can also make negative testing much simpler, but it can be hard to correctly identify those requirements. Requirements definition is helped by an up-front effort, perhaps concentrating on characteristic failures and instrumentation that would help identification and diagnosis
- Co-operative support from coders and designers can reveal weaknesses and exploitations that might not be obvious to a black-box tester

4.3.3. Political support

As testers, we’re professional pedants. We nitpick, find fault, assess the details rather than the vision. The idea that someone should deliberately spend their time trying to break the system in increasingly esoteric ways will seem extraordinary to many people outside the test team.

Negative testing needs political support if it is to remain effective and valuable – and many test managers are already well versed in providing this kind of support. Indeed, some avoid the term ‘negative testing’ entirely, feeling it is too ‘negative’.

Whatever the approach, it is important that the business understands the value of the information produced by negative testing. Not only are testers often the first customer advocates that actually experience the integrated, working system, but the specialised skills of negative testing can reveal flaws that are fatal to a customer relationship, but may be seen only rarely in the rarefied world of software development.

It is just as important that the testers understand the value that the business places on that information. This understanding will not only motivate the team, but can help them to refine their test approaches and make an assessment of the value of faults and related tests. This is an important skill when choosing the most useful tests in limited timeframes.

4.4. Scripted or exploratory?

Formal techniques for test derivation can be used for negative testing, and are effective during design and unit testing. However, in later phases, negative testing is reactive, open-ended (see box), and can be hard to execute (see box). The reductionist approach of most formal techniques can result in a large number of time-consuming tests, with no real idea of coverage. A significant proportion of negative testing in these late stages will be semi-scripted or exploratory.

5. Test selection strategies – and when to stop

Test selection strategies guide the team in deciding which tests to plan and perform, and which to do first. Negative testing is open ended, and selection strategies need to deal with the way that one test may reveal a new set of more important tests. One way to do this is by choosing tests that allow broad and reliable observation of the system. The following approach is based on this heuristic;

- Test the effectiveness and robustness of the exception handling system early – the results of later tests will depend on its reliability and accuracy.

- Use a broad set of negative tests to observe the system from many different perspectives. This will help you verify and improve the risk model of the system that you may be using to prioritise other tests.

A **risk model** describes the known and expected product risks, and may also describe risks to the product. Some models are formally documented, while others are simply a shared understanding. The risk model is used to help design the system to deal with risks, and to help prioritise testing. One of the significant risks of risk-based testing is that the risk model itself is wrong, or incomplete.

- Explore the system, looking for potential weaknesses. Prioritise test effort to match.
- Prioritise testing based on known exploitations. This can find bugs, or allow increased confidence that the system is robust. Malicious exploitations, and exploitations that tend to lead to a crash are good candidates for this early approach.
- Prioritise testing based on known failure modes, particularly those where impact analysis indicates serious consequences.
- Stray from the path, particularly if testing is going well, and you may expose unconsidered risks more quickly. Think about all the ways of making a system fail that are not simple paths, but conditions or conspiring events.

5.1. When to stop

If deadlines and budget have some flexibility, the most obvious approach is to stop testing when you are no longer finding significant new issues. This assessment should always be backed up with an idea of how much of the system has been observed under test. You might use requirements-based or functionality-based measures.

In projects with firm deadlines, the information collected can be used to prioritise tests and justify any extension that may be required. The information can be given greater value and impact by aiming the tests at an assessment of failure modes and verification of the risk model.

Because negative testing is open ended, and can decrease coverage, test coverage (number done/number planned) is not a useful way to judge when it is best to stop.

Negative testing is open-ended

Just as the range of wrong or inappropriate answers is often much larger than the range of right answers, so the potential number of negative tests vastly exceeds the number of tests that look at acceptable input and actions. While it may be possible to use formal techniques to derive a limited number of tests to show that functionality works, the number of tests that could show that it does not is limitless.

Coverage in an open-ended set is hard to establish, as an upper bound cannot be found. Furthermore, strategies such as 'highest risk first' assume that the highest risk tests are already known, yet system problems can reveal a need for new tests that are more important than the existing tests. Negative testing has the potential to make coverage go down, as it can produce information about the inadequacy of the existing test set.

It is possible to use formal techniques to derive some closed sets of negative tests, particularly for exception handling functionality.

6. Techniques to derive test cases

Negative testing is not a test design technique, but rather an approach or a classification. It is possible to use many formal test design techniques to derive tests that can be classified as 'Negative Tests'. This section details the application of a variety of well-known techniques:

- Boundary Value Analysis and Equivalence Class Partitioning
- State Transition testing
- Test against known constraints
- Failure Mode and Effects analysis
- Concurrency
- Use cases and mis-use cases

6.1. Boundary Value Analysis and Equivalence Class Partitioning

These two techniques are based on input and output data, and an expectation of the system's behaviour.

Boundary Value Analysis (BVA) examines those data elements that can take a continuous range of values, using the requirements and design to predict boundaries where the system's behaviour changes. The idea is to produce three values – one on the boundary itself, and the other two either side (as close as quantisation permits). If the boundary is between valid and invalid ranges, the test case that uses the invalid value will be a negative test – for instance, using 66 in an age field that only accepts values from 18-65.

Equivalence Class Partitioning (ECP) looks at the range between the boundaries. Each member of a given equivalence class should, in the context of a known test, make the system do the same thing – so the tester does not have to test every value in an equivalence class. Ranges of invalid input data can be seen as negative tests – for instance, an age field may be expected to reject all negative numbers in the same way.

ECP is commonly extended to include sets of non-continuous values, rather than ranges of continuous values. Be aware that some inputs may look equivalent, but may actually show very different behaviour. For example, the input to a simple web form may be rejected if empty or too long, but the correct combination of control characters may compromise the security of the underlying web server.

6.2. State Transition testing

Given a state transition diagram, or an equivalent understanding, it is straightforward to derive test cases that explicitly examine whether unreachable states are indeed unreachable. A variation on this works in the same way as n-switch testing; after a (known) set of transitions, are the unreachable states still unreachable? Graphing tools, such as Compendium-TA [4], can help you to derive such tests.

6.3. Test against known constraints

Most systems have explicit and implicit restrictions and constraints. Treating these constraints as requirements (see Robinson+Robinson, [5]) can lead to a variety of negative tests. Examples:

- "The site is designed to be viewed with Internet Explorer 4.5 or later" – a negative test might use IE 3.0, or Netscape.
- "No more than five users will use the system at the same time" – a negative test might try six, then eight.

Typically, these tests involve measurement and observation of the system's behaviour, rather than direct tests against expectations. This is only to be expected if working outside the system's operating parameters, and the observations can lead to an improved understanding of the system.

6.4. Failure Mode and Effects analysis

It is possible to predict a system's characteristic failures from an analysis of the underlying technology, the implementation, and known faults. This analysis is the basis for tests that observe the system's behaviour under conditions of failure. It is important to capture and document this information – particularly if they allow troubleshooting on the data or environment. Such documentation is often a required output of testing for organisations which monitor their systems and have technical experts available to take action while the system is in use (i.e. banks, phone companies). Alternatively, for more widely distributed packages, the information can become part of an FAQ or troubleshooting guide.

These tests may be impossible to execute without an effective test harness or application driver. Such harnesses are often custom-built, and may need to run with instrumented versions of the code. However, tools such as Canned HEAT and Holodeck (both from the Florida Institute of Technology, [6]) allow generic failures to be introduced to software running under Windows.

External and Internal Failure

The distinction between external and internal failure is necessarily subjective. If a system is seen as a unit of working code, most failures are external, while if a system is seen as a complex collection of co-dependent processes, operating system and hardware, the same failures might be internal. Some test groups class external failure as failure of a system that is not under control of the developers, while internal failure is failure of a system that can be fixed. This reflects the response rather than the problem.

6.4.1. Families of failure

There are a range of sources that can help you develop families of failure modes. Root-cause analysis on existing faults, system design documentation, knowledge of characteristic problems of the infrastructure can all help identify failure modes and so provide the source for derived tests. Appendix II gives a short list and pointers to further resources.

6.5. Concurrency

Testing concurrent use of resources can be a very fruitful way to discover bugs. Initial analysis involves identification of data, database entities, files, connections and hardware that more than one process might try to use simultaneously. Simple, custom-built tools can help by allowing the tester to make use of a resource before the system does, and release it at a time of their choosing. Tests should also check that the second requestor does eventually get control of the resource. More complex tests will look at more than two requests, queuing, timeouts and deadlocks

6.6. Use cases and mis-use cases

Use cases, in practice, tend to deal with ‘happy path’ use of the system. Their coverage of the varieties of bad input, cycles of rejection and partial transactions is often sparse. The term ‘mis-use case’, while not remotely standard, can help to identify and differentiate these explicitly.

Use cases that exercise these paths can help to improve the design by illustrating user activities that are outside the normal range of expectation, and allow a formal approach to test selection and coverage.

Some general mis-use cases for a GUI or browser could include the following actions or variations:

- Field entry: varying order from usual sequence on form
- Re-entry following rejection of input
- Choosing to change existing details
- Use of ‘cancel’
- Use following ‘undo’
- On browsers, navigating using the ‘back’ button, bookmarks etc.
- Starting sequence halfway
- Dropping out of sequence / interaction without completion/logout.

7. Semi-scripted testing

Semi-scripted tests can introduce some of the control of scripted testing, without needing as much maintenance. They are a fine way of helping the more exploratory parts of negative testing to stay on track.

7.1. Checklists

Where a fully scripted test may be hard to write and maintain, a checklist can provide a simple way to control, limit and measure test activity. Such checklists can be maintained more easily, and if stored centrally can act as the basis for new tests throughout an organisation. Useful checklists for negative testing include:

- Lists of input that should be tried when checking validation or failure with bad input
- Known exploitation techniques
- Lists of error messages
- Environment / configuration outside constraints
- Steps along a 'happy path' and variation
- Harsh but possible sequences – 'Soap Opera' tests

7.2. Measurement under controlled circumstances

It is hard to quantitatively define the expected outcome of some tests. Some tests require measurement and observation of a number of interrelated factors, with no pass/fail decision at the end. In order to get consistent results – or to measure a degree of inconsistency - it is useful to have a scripted, repeatable approach. This also allows a baseline; testers can deviate from it and record the changes that they tried.

This approach is particularly suited to failure mode analysis, recovery functionality following system failure and problem resolution actions from the exception-handling system.

7.3. Scalability

Testing is a predictive activity. It may be necessary to use a small system to predict the behaviour of a large one, or to predict an existing system's response to a change in volume. All systems will fail under some combination of volume or stress, and tests that seek to find the system's limits by driving it to failure can be classed as negative tests.

A failure may not be as obvious as a system crash, and trends in response time and queue length may give an early warning. Observations made of a test system as it is driven to failure can help identify problems in a live system before they become fatal.

Simpler to write than to perform

Negative tests are often simple to write, but hard to do. This applies particularly to failure modes, concurrency, and exception handling. This need not be a problem; the tests have great value as thought experiments for designers and exploratory testers.

Ensure that such tests have sufficient time and tool support. Fault injection tools can make impossible tests practical, and monitoring tools allow problems to be studied 'in the wild' i.e. during a volume test.

8. Ways of approaching test execution

Any test has the potential to expose a weakness during execution. It is important that testers can recognise and exploit these weaknesses – either by adapting the current test, or with a newly designed test later in the process. Designing negative tests to exploit these observed weaknesses is, in essence, exploratory testing. This section covers ways that testers can more easily observe weaknesses, and find effective exploitations.

8.1. Recognise and exploit weaknesses

Finding and exploiting weaknesses is essentially opportunist – and, like all opportunism, can be greatly helped by preparation, and by being in the right place at the right time. It relies on the tester's experience in two areas; observational skills, and knowledge of the underlying technology.

8.1.1. Observational skills

Observational skills help the tester to spot symptoms that indicate an underlying flaw. A slow process, an extra button press, an inappropriate error message can all indicate that a system has a weakness. To the experienced tester, a weakness can be suggested by the choices available at points along a path, by a default value, by a slight change in behaviour where none is necessary.

These symptoms are hard to spot – they may not be noticed, or they may be so common and apparently insignificant that they are ignored. An effective negative tester makes a broad observation of a wide range of information, and is able to sift out the symptoms from the noise. In doing this, the tester relies on the extraordinary power of the human brain to find and match patterns in apparently disorganised information. This power is enhanced by experience.

Observational skills can be acquired on the job, but are honed by practice and discussion. Presenting models and trends within the team can both refine a tester's abilities, and help train other members of the team.

8.1.2. Knowledge of the underlying technology

Knowledge of the underlying technology (including the system under test) allows the tester to make use of their observations;

- Propose (and test) an exploitation that opens up the weakness effectively and repeatably
- Identify similar weaknesses and failure modes
- Value the weakness in terms of potential for harm and ease of attack

Using this knowledge well requires a creative approach. An inventive tester may be able to use an aspect of the system in an entirely unexpected way – and in doing so, circumvent all existing checks and balances. The knowledge is system dependent, and as such is always acquired while actively working on the system - although experience may allow some testers to acquire it more quickly. An ability to draw analogies between testing and other experiences can help to model and communicate key ideas, and can give the tester a framework for an unfamiliar system.

8.2. Build models of failure, as well as success

A tester can gain insight into the system under test by building simple models. An extension to this is to try to fit symptoms and weaknesses into the model; modelling the system as it exists, rather than as it should be. If the system deviates from expectations, change the model so that it fits the behaviour. The resulting model may have a clear potential for further failure, revealing the exploitation that will open up the weakness. The approach could be described as exploratory failure mode analysis.

Example thought process (extrapolated from a real test session):

- I am testing functionality that renames a file to the user's choice of name
- I know that the filename can't have dots in. If it does, the file vanishes – and so functionality exists to prevent the user from entering dots
- My initial model is that only those names that contain exclusively alphanumeric characters are passed. Any non-alphanumeric character causes the rename request to be rejected.
- I paste in a string that contains Hebrew characters, and find that the file is renamed to include the Hebrew characters. I could log a bug now, but I know there's more.
- My new model is either:
    - Some punctuation (dots) is blocked – but perhaps the system is rejecting invalid characters, rather than only accepting valid input. Maybe I can find other characters that aren't alphanumeric, but still accepted. My next test will explore which other characters may be entered, and I will try : and \, /, ? and * first because I know that filing systems have problems with these characters. This test may reveal a weakness that is not important, but easily exploited.
    - Although typed punctuation is blocked, pasted punctuation may be passed. Perhaps the system is rejecting keystrokes, and not checking the input string. I will try to paste in a filename that contains one or more dots. This test may reveal a weakness that is hard to exploit, but whose consequences are significant.
- Hebrew is written from right to left. If I can't show that the fault fits either model above, I might try finding a model that is affected by this (thanks to Robert Sabourin for pointing this one out!)

Modelling a system that includes potential problems is something that many testers do, and it is one of the underlying skills of testing 'intuitively'. The example above is a model of a failing validation process, but two more general modelling techniques stand out: state transitions, and bug clusters.

8.2.1. State transition – unexpected states and incomplete transitions

A state is characterised by a set of behaviours, and it is possible to model the system's internal state machines by observing behaviours that form a set – a different set of behaviours; a new state. Building a state model can be an effective way to reveal weaknesses. Problems show up when an expected state does not have the normal set of behaviours: the tester has found a breakdown in the system, or their model.

The breakdown may be a problem, or may be exploited to reveal a problem; typically, the difference is caused by an incomplete transition (if the underlying states are implemented poorly), data corruption, or inconsistency between design and implementation. Problems may not be manifest on the first entry to a state, but data corruption, in particular, can cause failures to be revealed on subsequent visits to ostensibly the same state.

Many experienced testers make these observations without consciously making a model, whether they are doing scripted tests or exploring the system. Indeed, it may be preferable to develop tester skills in such a way that observation and recognition is done constantly and almost subconsciously, allowing the tester to concentrate their attention on the task at hand, but able to notice inconsistencies and possible exploits.

8.2.2. Bug clusters

An underlying problem that is the cause of one error may yet be the cause of many more. Because of this, faults are not randomly distributed, but fall into families. Such families are sometimes called 'Bug Clusters'. (Kim Davis, Robert Sabourin, [7])

Testers should look out for bug clusters when executing and designing negative tests; a model of the underlying error and its causes may lead the tester to design more tests, with a greater expectation of finding problems.

Bug clusters can lead to lots of identified bugs, and to a greater understanding of risk and coverage. They can expose weaknesses in the software development process, and make a persuasive case for improvements in design and coding practices.

8.2.3. Finding faults without doing more testing

A bug cluster or other model may encourage testers to concentrate on faults they expect – significant or not. This is often satisfying, but the model itself can hold the key to a faster, cheaper way of finding remaining faults.

Rather than making the faults reveal themselves through failures, it may be possible to take a systematic approach to finding the errors in the code. In this sense, design and execution of new negative tests is passed up for a well-aimed review, and the bugs are found with less effort poured into test execution.

8.3. Intuitive testers: Bloodhounds, breakers and bug hunters

Bloodhounds follow a trail to their quarry. Some testers operate like bloodhounds; they follow a trail that is almost imperceptible to onlookers – often without knowing precisely what is at the end of their trail, but sure that there is a problem to be found, out there, somewhere.

Some people break things. Your users will. Feel happy if you have one or two people who break things on your team (but perhaps don't feel so happy if you have a team who can't do anything but break the system!). These people can find a bug in anything, but without some discipline or skill, they find lots of non-reproducible bugs, and have difficulty diagnosing the underlying problems.

Bug hunters are experienced testers, able to use a range of tools to separate and diagnose a bug from its symptoms, untangle interacting bugs, identify problems that are caused by the test environment/data, reproduce 'unreproducible' bugs and a host of other vital, difficult tasks.

8.4. Tool use

Negative test execution can be enhanced by a range of tools. The most effective tools for negative testing, as for exploratory testing, are perhaps those which give a new perspective of the system; tools which encourage the testers to ask '*Why* is it doing *that*?'.

Capture-replay tools have their place – particularly for setting up tests and avoiding manual errors during repetitive tasks – but negative testing depends on manual observation of the system's actions, however automated the test execution.

It can be also useful to capture some perspective of what the tester is doing, so that their actions can be reviewed after a fault is found. Such a review can allow a problem to be reproduced more easily, but may also change the tester's perception of what they were doing. No tool can capture all aspects that may affect the outcome of an action, but session logs, pair observation, periodic screen-capture tools (example Spector [8]), the 'capture' part of capture-replay tools and video cameras / webcams can all prove useful. These tools are particularly useful for studying testers who are not part of the regular team, but who always seem to break a system in the first few minutes of use. For no obvious reason, this trait is common in managing directors.

Test sessions and session logs (Bach [9], Lyndsay [10]) have a part to play in managing the scope and duration of exploratory testing, and are useful tools to direct the efforts of bloodhounds and breakers.

It is also useful to have some sort of tool that can look at the underlying data – and ideally, compare 'before' and 'after' snapshots. This is often the only way to directly discover data corruption, particularly if the data corruption is happening in an unexpected location. Although such tools tend to be custom-built, something that can take a snapshot of part of a database or the environmental variables can be very valuable. It may also be useful to have access to a debugger, to allow some kind of inspection of memory and internally held variables.

System inspectors, such as network, CPU, and memory monitors, can give an early warning of failure, and so help in designing and aiming negative tests. The unit test process may have generated customised tools that act as a reversed test harness; they watch the system under test.

Fault injection tools can cause the system to react as if an external failure has just happened, allowing testers to explore circumstances under which the system should fail, or handle failure. Custom tools or code intervention tend to have fixed capabilities, but newer tools such as Holodeck and Canned Heat [6] generically manipulate the system's environment.

A tester needs to be able to recognise a working state. This needs tools to look at internal aspects of the system, and observation skills to spot weaknesses that perhaps did not exist before the return to a working state. The tester's exploratory, observational skills will be vital, even when executing a scripted test.

A tool to refresh the system after a test is particularly important for negative testing, where failures can leave the system in a compromised state. The more frequently the test environment is refreshed, the more reliable the test results and diagnosis. This must be balanced against the reduced opportunity to observe problems from previous tests.

9. Diagnosis and logging

Making a well-supported model of a failing system leads directly to a diagnosis of that failure, and most diagnostic testing involves some degree of negative, exploratory work. However, as failures found during negative testing can be particularly hard to diagnose, diagnostic action may not always be appropriate.

9.1. Diagnosis

Testers should be aware, particularly when using a known exploitation, of well-trodden paths; if the system under test exhibits a generic failure under well-known circumstances, it may be more productive to note the weakness and its likely consequence than to follow the generic failure to its bitter end. Consider the cost of diagnosis in terms of missed opportunities; with fixed deadlines, a four-hour diagnosis can mean that you lose four hours testing, and the corresponding potential to find new bugs.

Some weaknesses enable malicious action or security breaches. However, not all flaws expose the system in this way. In practice, users often feel that they share in some of the blame for a problem that is caused by actions that seem irresponsible; depending on the state and use of the product, there may be more important work to do than to prevent users shooting themselves in the foot.

Catastrophic failures can cause a system crash that leaves no usable information about how or why the crash happened. If the problem is not reliably reproducible, it may be helpful to record as much information as possible about each occurrence, including timestamp and system conditions. This can help assess the severity of the problem, and may help identify patterns that lead to a possible cause.

9.2. Logging

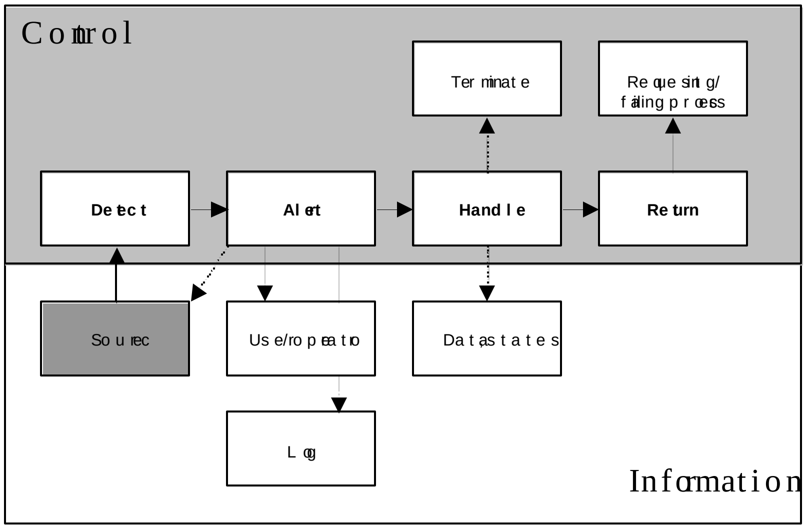
Although it is common practice to develop models of the way the system is failing, it is a poor idea to state these models as if they were facts. The model remains a model, and however useful it has been for finding further problems, may not reflect the true nature of the system.

Some projects have a policy of developing an operations manual during testing, to help the operators of the live system diagnose and possibly fix problems. This manual is jointly developed with the design and coding teams, particularly those who have been involved in fixing problems and tuning the testing environments. Logged bugs from negative testing will be the source for much of this document.

10. Testing functionality that copes with exceptions

Handling exceptions well takes a great deal of code – and the proportion of exception-handling code generally increases with system maturity and stability. The exception-handling system can itself be a dense source of bugs, and those bugs may be hard to spot. See Appendix I for details.

A typical model covers detection, notification, handling the exception and returning control to a working state:



Note that this may be a looping or recursive process; the *handle* and *return* steps may return control to the failing process, the requesting process, or terminate at a range of levels.

10.1. Exception detection

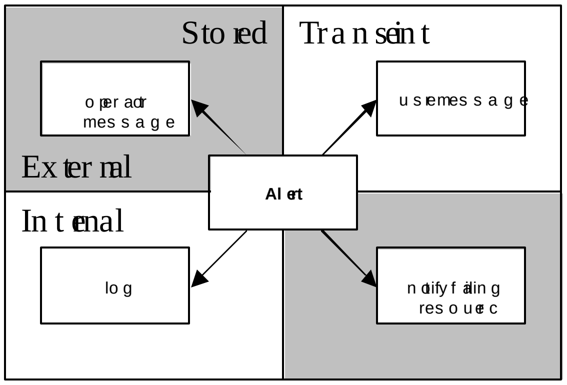
A system will detect and handle many more exceptions than the user ever gets to see. Formal techniques, particularly Failure Mode and Effects analysis, will help to identify these exceptions.

Exception detection is an important part of unit testing, and is greatly helped by test harnesses. It is harder to do well at a later stage with an integrated system. When designing tests, it is worth bearing in mind that:

- The problem may be detected well after the problem has happened
- Some problems that are detected as an absence of resource may be mis-identifying a delay in response from that resource – further functionality may be required to measure time taken for a response.

10.2. Alerts and logging

Information about a detected exception can be sent to many places, and may be stored for later use;



Many systems have generic routines to enable the information to flow, and generalised test cases can be derived using formal techniques. Test designers should be aware of common failings (Appendix I).

It can be useful to extract the external error messages from the central store, or (if hard-coded) from the code. This will give an insight into the scope of the error-messaging system, and help assess coverage.

10.3. Problem resolution

A problem can be resolved in many ways, and tests need to reflect the specific solution implemented. Tests should be structured to allow observation of other processes and their reaction to any resolution. Note that the resolution can cause further rejections. When deriving tests that cover successive rejections, modelling this aspect of the system as a state machine can be useful.

Some problems can only be resolved by system recovery. Robustness is crucial, and if a decision is made to exclude negative testing from a phase, it is important to consider recovery functionality separately, from a point of view of business risk.

Recovery Functionality

Some problem resolutions invoke functionality that enables the system to avoid or recover from catastrophic failure. Examples include:

- Fail-over
- Restart following failure
- Restore from backup
- Rollback

These resolutions may be part of a well-specified process and often involve skilled diagnosis and manual intervention.

10.4. Return to a controlled state

It is hard to know how much needs to be done to get a system back on track, and this makes it hard to design and code functionality to do it well. It also means that it is hard to test. While it is possible to script tests that check that a system has done all it was *meant* to, this is only half of the story. Failure to return to a working state may be revealed by intermittent or non-reproducible failures, and may only be shown much later than the original error.

10.5. Unscripted testing of an exception handling system

Negative testing exercises the exception handling system, and testers should have a good opportunity to observe weaknesses without actually testing the system explicitly. Appendix I contains a short list of characteristic problems that may allow weakness and exploitation.

However, because the exception handling system perturbs the system under test, it can also be a good tool to open the opportunity for an exploitation. For instance, as a response to bad input, the exception handling system may generate a modal dialogue preventing further input – but if the dialogue is closed or bypassed, it may be possible to pass that input into the system.

Whittaker's *How to Break Software* [11] gives succinct and practical advice, and it is not the purpose of this paper to paraphrase the excellent information in that source.

11. Criticisms of Negative Testing

Negative testing comes in for a lot of flack. It is questioned, excluded and occasionally ridiculed. This attention and frequent self-justification means that it is often well aimed, and an effective way of finding good bugs. Some frequently heard criticisms are:

11.1. "That's an acceptable failure"

Counter to received wisdom among testers, some failures are acceptable.

Counter to received wisdom among coders, some unfixable failures are still unacceptable. A failure that does not have a fix does not become an acceptable failure.

It is best to defer to the customer, then the design. If the requirements state that "It should fail under those conditions", they could be changed to "It should fail like this under those conditions"

11.2. "But that would never happen in normal use"

It is hard to make predictions about real-world use, but it is sure that real users will find ways of using the system that were not originally considered. It is also worth considering the 'real-world use' of malicious users, who understand characteristic weaknesses, and seek to exploit them. Negative testing can be used to identify these weaknesses, and give confidence that the system can cope with changes in use. It has been said that testers should spend their time where the users spend their time [Erik Petersen, STARWest02]. This is not necessarily true for negative testing, particularly when dealing with the potential for significant failure.

A list of common exploitations can be shared with the people who assess bugs – not only is this easier to read and digest, but the exposure may result in action to find and fix all bugs that share the same exploitation, rather than reject them piecemeal. Be careful, however, of calling all bugs that share the same exploitation 'duplicates' – the coders may not fix the bug, but make the system harder to exploit. This approach keeps the weakness, and a better exploitation may be found. However, it may be the only solution, particularly in cases where the weakness is inherent in the design, or exists in a third-part product.

11.3. "Don't the other tests do that?"

Most tests that look at input and output are weak tests. Their success is based on the output fitting some criteria of correctness. A stronger test would want to check that all of the right results have been returned, none of the returned results are wrong, and that nothing unexpected has changed. Negative testing can be strong, compared to weak functional testing, and can spot problems that would be missed by simply checking the functionality.

11.4. "Why find a problem we can't fix?"

Negative testing discovers errors outside 'code' – design, other components, even other hardware.

Why go after errors in stuff we can't fix? Because we are customer advocates, not coders. We want to know when our paymasters will have a bad day – not when we have made a mistake. Maybe there is an area we need to look at harder, maybe there is a complexity we have missed, maybe someone else has made a mistake (unthinkable!).

Any fault found, regardless of ownership, gives us valuable information about the experience the users will have using our product. Whether it is our fault or not, we would prefer the customer not to experience crashes, data loss, corruption, security breaches – and negative testing can help us discover areas where we could improve their experience; failing more gracefully, giving more information, allowing a 'safe' option. This may increase the complexity of our product – but it may also be an opportunity to add business value.

12. Conclusion

The following statements summarise the author's experience of negative testing.

- Negative testing is a core skill of experienced testers, and requires an opportunist, exploratory approach to get the best value from the time spent.
- Negative testing can not only find significant failures, but can produce invaluable strategic information about the risk model underlying testing, and allow overall confidence in the quality of the system. However, it can reveal fundamental flaws in the project, as well as the product.
- Negative testing is open-ended and hard to plan granularly. It needs to be managed proactively rather than over planned.
- Although negative testing is a powerful and effective approach, it is also a hard-to-manage task that has the potential to produce unwelcome information. Attempts to ignore or exclude negative testing may need to be robustly opposed.

13. Appendices

13.1. Appendix I: Characteristic exception handling failures

The following examples may help to identify possible tests;

- A single fault can trigger many failures, particularly when processing a large amount of repeating data. Example; a switch re-configuration causes all records of phone calls to be improperly formatted – now the system will pass all calls through non-optimised error-handling routines, and write details to a log. Handling the volume of problems can cause the system to slow, and cause its storage to fill. This may, in turn, cause further errors, and the problem can accelerate.
- If an exception is not detected, the system will carry on as if no problem has been found. This can often lead to problems at an Operating System level, as requests are made that cannot be fulfilled.
- If an exception is detected, but the message detailing that information is lost or delayed, the system may carry on as if no problem has been found. This can often lead to worse problems.
- When asking a user for a corrected input, the system may block further activity. Systems can get stuck in this state, and may also restrict valid activity, such as screen re-draws and other background tasks. This problem can also be influenced by handling control to another process (a screen-saver, say) while in a restricted state.
- Simple logging mechanisms may not scale, introducing problems of concurrency, disk space or resource hogging.
- If a great many messages are sent to an observer or log, they may be ignored (so many that the problem becomes expected) or obscured (one important message is hard to spot in many irrelevant ones).
- Coders often write code to act on known error conditions. An unrecognised error condition may slip through undetected. With input validation, if the code excludes only known input errors, an invalid entry that is not recognised may slip through. It is better practice to exclude everything that does not fit the expectations of valid input than to exclude everything that fits expectations of invalid input.
- Poor messaging is a common flaw. Obscure, incomplete, duplicate or ambiguous messages not only cause problems to users, but also make negative testing harder – and may signify misunderstandings in the design/coding team or other systematic errors.
- Localisation / internationalisation will require the text of user-facing error messages to be chosen from an appropriate language set. This may introduce new functionality, and will certainly be problematic on systems that have hard-coded error messages.
- Generic messaging systems typically store their messages centrally, referring to them by an identifier. This can lead to problems where the wrong message is sent; either because the data is corrupt or badly set up, or because the coder has used the wrong identifier. It may be necessary to test the data itself.
- Resuming control flow from the correct point is not necessarily trivial. It involves passing complex information, and returning control to a state that can cope, but is not functionally different from the state that threw the initial error. It may be necessary to carefully check the state of the system before executing more tests.
- Tidying up after an error is also not trivial. A system may need to resubmit requests, tidy screen and memory, rollback transactions, restart processes and return control to the right point, in the right state. If the error has been caused by a bug elsewhere, inconsistent information may have been passed into storage – and later tests may be compromised.
- Exceptions that are handled uniquely, outside an existing exception-handling process, may be less reliable, and less tested, than those handles by a centralised, generic process.
- Error messages can have multiple or ill-defined meanings (i.e. poorly maintained legacy system). A definitive list of errors can reduce the impact of this problem on testing, and including an unambiguous numeric code can hep testing and support.

13.2. Appendix II: Families of failure

The following list is by no means exhaustive, but may help trigger further ideas about possible failure.

- External resource: unresponsive, slow response, unintelligible / inappropriate response
- Co-operative process failure: particularly intermittent processes, multi-tasking and recursion
- Concurrent use: resource locked, request for lock refused, deadlock, lock response delayed
- Sacrificial processes: processes allowed to fail and recover in a controlled manner
- File system: File cannot be found, opened, read, written to, permissions changed, file system recognises media error, media removed, media full
- Network: Network down, network busy/slow, transmission segment lost/corrupt/out of sequence, dialogue between processes interrupted
- Memory: not enough for requested allocation, fragmented
- Limits reached: Queues, licences, threads, connections, array size, resource allocation

Ordered and classified lists of faults at:

Bug Taxonomy and Statistics (2001), B. Beizer, O. Vinter

http://inet.uni2.dk/~vinter/bugtaxst.doc

Testing Computer Software (Second Edition): Appendix: Common Software Errors: Cem Kaner, Jack Falk, Hung Quoc Nguyen

http://qacity.logigear.com/nr/qacity/documents/common_errors.html

13.3. References

[2] BCS SIGIST: *Standard Glossary of Testing Terms* (British Standard BS 7925-1)

[3] Beizer, Boris. *Software Testing Techniques*, van Nostrand Reinhold, 1990

[4] Compendium Developments (http://www.compendiumdev.co.uk): *Compendium-TA* (http://www.compendium-ta.com)

[5] Robertson + Robertson, *Mastering the Requirements Process*, Addison-Wesley, 1999

[6] Florida Institute of Technology, *Canned HEAT* and *Holodeck*, http://www.se.fit.edu/projects/CannedHEAT/home.htm and http://www.se.fit.edu/projects/holodeck/

[7] Kim Davis, Robert Sabourin, *Exploring, discovering and exterminating Bug Clusters in Web Applications*, QWE2001

[8] SpectorSoft, *Spector*, http://www.spectorsoft.com/

[9] Jonathan Bach, *Session-Based Test Management*, STARWest 2000 http://www.satisfice.com/articles/sbtm.pdf

[10] Lyndsay, *Adventures in Session-Based Testing*, STARWest 2002, http://www.workroom-productions.com/papers.html

[11] Whittaker, *How to Break software*, Addison-Wesley 2002

13.4. Further reading

Whittaker, *How to Break software*, Addison-Wesley 2002

Kaner, Bach, Pettichord, *Lessons Learned in Software Testing*, Wiley 2002

Murphy, http://murphy.pescatore.ch/index.php?use=display&id=0 , http://www.romankoch.ch/capslock/murphy.htm, http://www.lorober.com/Favorites/documents/Murphy.htm

13.5. Thanks

My great thanks to my reviewers, without whose help this paper would be far longer, and much worse:

- Alan Richardson
- Robert Sabourin
- Joanne Webb
- James Bach
- Erik Petersen