The Importance of Data in Functional Testing

Author: James Lyndsay
Version: 1.1

Author James Lyndsay, Workroom Productions Ltd.
Biography James Lyndsay is an independent test consultant with ten years experience. Specialising in test strategy, he has worked in a range of businesses from banking and telecoms to the web, and pays keen attention to the way that his clients' focus is shifting away from functional testing.
Abstract A system is programmed by its data. Functional testing can suffer if data is poor, and good data can help improve functional testing. Good test data can be structured to improve understanding and testability. Its contents, correctly chosen, can reduce maintenance effort and allow flexibility. Preparation of the data can help to focus the business where requirements are vague.
Keywords Data, Functional testing

The Roles of Data in Functional Testing

Testing consumes and produces large amounts of data. Data describes the initial conditions for a test, forms the input, is the medium through which the tester influences the software. Data is manipulated, extrapolated, summarised and referenced by the functionality under test, which finally spews forth yet more data to be checked against expectations. Data is a crucial part of most functional testing.

This paper sets out to illustrate some of the ways that data can influence the test process, and will show that testing can be improved by a careful choice of input data. In doing this, the paper will concentrate most on data-heavy applications; those which use databases or are heavily influenced by the data they hold. The paper will focus on input data, rather than output data or the transitional states the data passes through during processing, as input data has the greatest influence on functional testing and is the simplest to manipulate. The paper will not consider areas where data is important to non-functional testing, such as operational profiles, massive datasets and environmental tuning.

A SYSTEM IS PROGRAMMED BY ITS DATA

Many modern systems allow tremendous flexibility in the way their basic functionality can be used. Configuration data can dictate control flow, data manipulation, presentation and user interface. A system can be configured to fit several business models, work (almost) seamlessly with a variety of co-operative systems and provide tailored experiences to a host of different users. A business may look to an application's configurability to allow them to keep up with the market without being slowed by the development process, an individual may look for a personalised experience from commonly-available software.

FUNCTIONAL TESTING SUFFERS IF DATA IS POOR

Tests with poor data may not describe the business model effectively, they may be hard to maintain, or require lengthy and difficult setup. They may obscure problems or avoid them altogether. Poor data tends to result in poor tests, that take longer to execute.

GOOD DATA IS VITAL TO RELIABLE TEST RESULTS

An important goal of functional testing is to allow the test to be repeated with the same result, and varied to allow diagnosis. Without this, it is hard to communicate problems to coders, and it can become difficult to have confidence in the QA team's results, whether they are good or bad. Good data allows diagnosis, effective reporting, and allows tests to be repeated with confidence,.

GOOD DATA CAN HELP TESTING STAY ON SCHEDULE

An easily comprehensible and well-understood dataset is a tool to help communication. Good data can greatly assist in speedy diagnosis and rapid re-testing. Regression testing and automated test maintenance can be made speedier and easier by using good data, while an elegantly-chosen dataset can often allow new tests without the overhead of new data.

Problems which can be caused by poor data

Most testers are familiar with the problems that can be caused by poor data. The following list details the most common problems familiar to the author. Most projects experience these problems at some stage - recognising them early can allow their effects to be mitigated.

- Unreliable test results.
  Running the same test twice produces inconsistent results. This can be a symptom of an uncontrolled environment, unrecognised database corruption, or of a failure to recognise all the data that is influential on the system.

- Degradation of test data over time.
  Program faults can introduce inconsistency or corruption into a database. If not spotted at the time of generation, they can cause hard-to-diagnose failures that may be apparently unrelated to the original fault. Restoring the data to a clean set gets rid of the symptom, but the original fault is

undiagnosed and can carry on into live operation and perhaps future releases. Furthermore, as the data is restored, evidence of the fault is lost.

- Increased test maintenance cost
  If each test has its own data, the cost of test maintenance is correspondingly increased. If that data is itself hard to understand or manipulate, the cost increases further.

- Reduced flexibility in test execution
  If datasets are large or hard to set up, some tests may be excluded from a test run. If the datasets are poorly constructed, it may not be time-effective to construct further data to support investigatory tests.

- Obscure results and bug reports
  Without clearly comprehensible data, testers stand a greater chance of missing important diagnostic features of a failure, or indeed of missing the failure entirely. Most reports make reference to the input data and the actual and expected results. Poor data can make these reports hard to understand.

- Larger proportion of problems can be traced to poor data
  A proportion of all failures logged will be found, after further analysis, not to be faults at all. Data can play a significant role in these failures. Poor data will cause more of these problems.

- Less time spent hunting bugs
  The more time spent doing unproductive testing or ineffective test maintenance, the less time spent testing.

- Confusion between developers, testers and business
  Each of these groups has different data requirements. A failure to understand each others data can lead to ongoing confusion.

- Requirements problems can be hidden in inadequate data
  It is important to consider inputs and outputs of a process for requirements modelling. Inadequate data can lead to ambiguous or incomplete requirements.

- Simpler to make test mistakes
  Everybody makes mistakes. Confusing or over-large datasets can make data selection mistakes more common.

- Unwieldy volumes of data
  Small datasets can be manipulated more easily than large datasets. A few datasets are easier to manage than many datasets.

- Business data not representatively tested
  Test requirements, particularly in configuration data, often don't reflect the way the system will be used in practice. While this may arguably lead to broad testing for a variety of purposes, it can be hard for the business or the end users to feel confidence in the test effort if they feel distanced from it.

- Inability to spot data corruption caused by bugs
  A few well-known datasets can be more easily be checked than a large number of complex datasets, and may lend themselves to automated testing / sanity checks. A readily understandable dataset can allow straightforward diagnosis; a complex dataset will positively hinder diagnosis.

- Poor database/environment integrity
  If a large number of testers, or tests, share the same dataset, they can influence and corrupt each others results as they change the data in the system. This can not only cause false results, but can lead to database integrity problems and data corruption. This can make portions of the application untestable for many testers simultaneously.

Classification of Data Types

In the process of testing a system, many references are made to "The Data" or "Data Problems". Although it is perhaps simpler to discuss data in these terms, it is useful to be able to classify the data according to the way it is used. The following broad categories allow data to be handled and discussed more easily.

Environmental data

Environmental data tells the system about its technical environment. It includes communications addresses, directory trees and paths and environmental variables. The current date and time can be seen as environmental data.

Setup data

Setup data tells the system about the business rules. It might include a cross reference between country and delivery cost or method, or methods of debt collection from different kinds of customers. Typically, setup data causes different functionality to apply to otherwise similar data. With an effective approach to setup data, business can offer new intangible products without developing new functionality - as can be seen in the mobile phone industry, where new billing products are supported and indeed created by additions to the setup data.

Input data

Input data is the information input by day-to-day system functions. Accounts, products, orders, actions, documents can all be input data. For the purposes of testing, it is useful to split the categorisation once more:

FIXED INPUT DATA

Fixed input data is available before the start of the test, and can be seen as part of the test conditions.

CONSUMABLE INPUT DATA

Consumable input data forms the test input

It can also be helpful to qualify data after the system has started to use it;

Transitional data

Transitional data is data that exists only within the program, during processing of input data. Transitional data is not seen outside the system (arguably, test handles and instrumentation make it output data), but its state can be inferred from actions that the system has taken. Typically held in internal system variables, it is temporary and is lost at the end of processing.

Output data

Output data is all the data that a system outputs as a result of processing input data and events. It generally has a correspondence with the input data (cf. Jackson's Structured Programming methodology), and includes not only files, transmissions, reports and database updates, but can also include test measurements. A subset of the output data is generally compared with the expected results at the end of test execution. As such, it does not directly influence the quality of the tests.

Organising the data

A key part of any approach to data is the way the data is organised; the way it is chosen and described, influenced by the uses that are planned for it. A good approach increases data reliability, reduces data maintenance time and can help improve the test process.

Good data assists testing, rather than hinders it.

Permutations

Most testers are familiar with the concept of permutation; generating tests so that all possible permutations of inputs are tested. Most are also familiar with the ways in which this generally vast set can be cut down.

Pairwise, or combinatorial testing (see Cohen, Dalal, Parelius, Patton: The Combinatorial Design Approach to Automatic Test Generation and Tai, Lei: A Test Generation Strategy for Pairwise Testing) addresses this problem by generating a set of tests that allow all possible *pairs* of combinations to be tested. Typically, for non-trivial sets, this produces a far smaller set of tests than the brute-force approach for all permutations, The same techniques can be applied to test data; the test data can contain all possible pairs of permutations in a far smaller set than that which contains all possible permutations. This allows a small, easy to handle dataset - which also allows a wide range of tests.

EXAMPLE: UTILITY CUSTOMER CARE / BILLING SYSTEM

In this example, we look at the permutations possible in a portion of the fixed input data for a simple billing system. We're all familiar with the output of such a system; each customer is billed periodically for usage, the value of their bill being affected by a tariff, and with various other customer attributes kept in the system. Each account in our hypothetical system can have the following characteristics;

A customer can have one of three tariffs (labelled 1,2,3 below). They may be billed Monthly or Quarterly, be High or Low value, and their last bill was either Paid or Unpaid.

Tariff	1	2	3
Billing period		M	Q
Customer Value		H	L
Last Bill		P	U

There are 3x2x2x2 = 24 combinations - and as the system complexity increases, so the number of possible permutations climbs ever more rapidly.

By requiring that the list holds not all possible combinations, but all possible pairs, the list can be reduced. The following six permutations contain all the pairs;

All possible pairs; M1, M2, M3. Q1, Q2, Q3. H1, H2, H3. L1, L2, L3. P1, P2, P3. U1, U2, U3. MH, ML, QH, QL, MP, MU, QP, QU. HP, LP, HU, LU.

Customer	Account	Tariff	Care	Bill
1	M	1	H	P
2	M	2	L	P
3	M	3	H	U
4	Q	1	L	U
5	Q	2	H	U
6	Q	3	L	P

This small, and easy to manipulate dataset is capable of supporting many tests. It allows complete pairwise coverage, and so is comprehensive enough to allow a great many new, ad-hoc, or diagnostic tests. Database changes will affect it, but the data maintenance required will be greatly lessened by the small size of the dataset and the amount of reuse it allows. Finally, this method of working with fixed input data can help greatly in testing the setup data.

This method is most appropriate when used, as above, on fixed input data. It is most effective when the following conditions are satisfied. Fortunately, these criteria apply to many traditional database-based systems:

- Fixed input data consists of many rows
- Fields are independent
- You want to do many tests without loading / you do not load fixed input data for each test.

To sum up, permutation helps because:

- Permutation is familiar from test planning.
- Achieves good test coverage without having to construct massive datasets
- Can perform investigative testing without having to set up more data
- Reduces the impact of functional/database changes
- Can be used to test other data - particularly setup data

Partitioning

Partitions allow data access to be controlled, reducing uncontrolled changes in the data. Partitions can be used independently; data use in one area will have no effect on the results of tests in another. Data can be safely and effectively partitioned by machine / database / application instance, although this partitioning can introduce configuration management problems in software version, machine setup, environmental data and data load/reload. A useful and basic way to start with partitions is to set up, not a single environment for each test or tester, but to set up three shared by many users, so allowing different kinds of data use. These three have the following characteristics:

Safe area	· Used for enquiry tests, usability tests etc. · No test changes the data, so the area can be trusted. · Many testers can use simultaneously
Change area	· Used for tests which update/change data. · Data must be reset or reloaded after testing. · Used by one test/tester at a time.
Scratch area	· Used for investigative update tests and those which have unusual requirements. · Existing data cannot be trusted. · Used at tester's own risk!

Testing rarely has the luxury of completely separate environments for each test and each tester. Controlling data, and the access to data, in a system can be fraught. Many different stakeholders have different requirements of the data, but a common requirement is that of exclusive use.

While the impact of this requirement should not be underestimated, a number of stakeholders may be able to work with the same environmental data, and to a lesser extent, setup data - and their work may not need to change the environmental or setup data. The test strategy can take advantage of this by disciplined use of text / value fields, allowing the use of 'soft' partitions.

'Soft' partitions allow the data to be split up conceptually, rather than physically. Although testers are able to interfere with each others tests, the team can be educated to avoid each others work. If, for instance, tester 1's tests may only use customers with Russian nationality and tester 2's tests only with French, the two sets of work can operate independently in the same dataset. A safe area could consist of London addresses, the change area Manchester addresses, and the scratch area Bristol addresses. Typically, values in free-text fields are used for soft partitioning.

Data partitions help because:

- Allow controlled and reliable data, reducing data corruption / change problems
- Can reduce the need for exclusive access to environments/machines

Clarity

Permutation techniques may make data easier to grasp by making the datasets small and commonly used, but we can make our data clearer still by describing each row in its own free text fields, allowing testers to make a simple comparison between the free text (which is generally displayed on output), and actions based on fields which tend not to be directly displayed. Use of free text fields with some correspondence to the internals of the record allows output to be checked more easily.

Example from billing system above:

Customer Name	Account	Tariff	Care	Bill
HP1 Monthly	M	1	H	P
LP2 Monthly	M	2	L	P
HU3 Monthly	M	3	H	U
LU1 Quarterly	Q	1	L	U
HU2 Quarterly	Q	2	H	U
LP3 Quarterly	Q	3	L	P

Testers often talk about items of data, referring to them by anthropomorphic personification - that is to say, they give them names. This allows shorthand, but also acts as jargon, excluding those who are not in the know. Setting this data, early on in testing, to have some meaningful value can be very useful, allowing testers to sense check input and output data, and choose appropriate input data for investigative tests.

Reports, data extracts and sanity checks can also make use of these; sorting or selecting on a free text field that should have some correspondence with a functional field can help spot problems or eliminate unaffected data.

Data is often used to communicate and illustrate problems to coders and to the business. However, there is generally no mandate for outside groups to understand the format or requirements of test data. Giving some meaning to the data that can be referred to directly can help with improving mutual understanding.

Clarity helps because:

- Improves communication within and outside the team
- Reduces test errors caused by using the wrong data
- Allows another method way of doing sanity checks for corrupted or inconsistent data
- Helps when checking data after input
- Helps in selecting data for investigative tests

Involving 'The Business'

Bringing QA into the development process early is a known way of reducing the cost of fault detection and resolution, and can improve time to market. The development process is initiated outside IT or QA teams by 'The Business', generally with some sort of business opportunity and associated requirements generation. A variety of different roles are encompassed by this vague and non-standard term, including those of regulator, customer, user and 'boss' - but one quality they have in common is that they known about the data rather better that they are likely to know abut the technical internals of the system. Data can enable effective communication between QA and 'The Business' in this early stage.

'The Business' is good at looking at data. Data can be compared with – and is often equivalent to – data used by existing process and systems. It can be easier to understand than tests, and it is often simpler to make a link between input and output data than it is to describe the internal process. Sharing data with 'The Business' can help to illustrate and model situations. This has the following advantages:

- Helps focus when requirements are vague
- Increases trust and understanding by improving ease of communication
- Helps early user identification of problems
- Improves familiarity and effectiveness of User Acceptance Testing

However, sharing the data has disadvantages. In sharing data, it is important to recognise and control the following problems:

- Data creep - where groups outside QA are using the Test Team's data, there may be pressure to include changes that aren't directly required by the Test Team. This can lead rapidly to large and uncontrolled datasets.
- Vague requirements can lead to vague data - although good data can help improve to focus requirements, truly vague requirements are likely to cause the data to change constantly until the requirements are firm. This introduces extra work, and can lead to out-of-date data.
- Incomplete data can lead to incomplete testing - if poor data is made a core tool for discussion, and testing only addresses known and discussed aspects of the data, the scope of testing may be too small and parts of the system can be left untested.

Data Load and Data Maintenance

An important consideration in preparing data for functional testing is the ways in which the data can be loaded into the system, and the possibility and ease of maintenance.

Loading the data

Data can be loaded into a test system in three general ways.

- Using the system you're trying to test

  The data can be manually entered, or data entry can be automated by using a capture/replay tool. This method can be very slow for large datasets. It uses the system's own validation and insertion methods, and can both be hampered by faults in the system, and help pinpoint them. If the system is working well, data integrity can be ensured by using this method, and internally assigned keys are likely to be effective and consistent.

  Data can be well-described in test scripts, or constructed and held in flat files. It may, however, be input in an ad-hoc way, which is unlikely to gain the advantages of good data listed above.
- Using a data load tool (see list at http://www.dwinfocenter.org/clean.html)

  Data load tools directly manipulate the system's underlying data structures. As they do not use the system's own validation, they can be the only way to get broken data into the system in a consistent fashion. As they do not use the system to load the data, they can provide a convenient workaround to known faults in the system's data load routines. However, they may come up against problems when generating internal keys, and can have problems with data integrity and parent/child relationships.

  Data loaded can have a range of origins. In some cases, all new data is created for testing. This data may be complete and well specified, but can be hard to generate. A common compromise is to use old data from an existing system, selected for testing, filtered for relevance and duplicates and migrated to the target data format. In some cases, particularly for minor system upgrades, the complete set of live data is loaded into the system, but stripped of personal details for privacy reasons. While this last method may seem complete, it has disadvantages in that the data may not fully support testing, and that the large volume of data may make test results hard to interpret.
- Not loaded at all

  Some tests simply take whatever is in the system and try to test with it. This can be appropriate where a dataset is known and consistent, or has been set up by a prior round of testing. It can also be appropriate in environments where data cannot be reloaded, such as the live system. However, it can be symptomatic of an uncontrolled approach to data, and is not often desirable.

Environmental data tends to be manually loaded, either at installation or by manipulating environmental or configuration scripts. Large volumes of **setup data** can often be generated from existing datasets and loaded using a data load tool, while small volumes of setup data often have an associated system maintenance function and can be input using the system. **Fixed input data** may be generated or migrated and is loaded using any and all of the methods above, while **consumable input data** is typically listed in test scripts or generated as an input to automation tools.

When data is loaded, it can append itself to existing data, overwrite existing data, or delete existing data first. Each is appropriate in different circumstances, and due consideration should be given to the consequences.

Frequency of Data Load

Different aspects of the data can be loaded with different frequencies and at different times, and consideration of this aspect of data load can be a substantial part of the Test Strategy's approach to data. The following triggers to data load are typical, and are listen in order of increasing frequency.

Not often

- At the start of overall test execution
- With each release of code / database schema made to the test environment
- At the start of each week (or any other calendar-based , rather than event-based trigger)
- When a tester asks for it
- At the start of every individual test

Often

Environmental and setup data are usually loaded least frequently, fixed input data is loaded often, and consumable input data is typically generated (but used rather than directly loaded) with each test. Frequently loaded data often has a higher cost and urgency associated with its maintenance, and it can be productive to find a method which allows fewer, less frequent loads.

Data Maintenance

All data needs work to keep it complete, clean, and up-to-date. The following list shows some of the reason for data maintenance. By monitoring these, time for data maintenance can be factored into the plans.

- Replacing consumed data
- Repairing broken data
- Responding to change - database schema, code, requirements
- New test requirements

Tasks to refresh the data should be built into the test plans. Repairs can be classified in the same way as bugs and fixes prioritised appropriately. Changes to the data can generally be planned for, but be aware that if the test scope has missed some tests, or if investigative tests need new data, the changes required may be short-notice and urgent.

Data maintenance can cause significant problems. Manual data maintenance is prone to error, and can introduce faults that lead to test failures, which are then logged and must be painstakingly resolved. Data maintenance is a sizeable task, and can make up a substantial fraction of overall test maintenance costs. However, the most common cause of problems is that data maintenance is generally (and often necessarily) performed by more than one group - different people tend to be responsible for environmental data than are responsible for fixed input data. Synchronising changes can be hard and effective management is needed to deal with differing priorities.

Solutions

Forewarned is forearmed with data problems, and recognising and preparing for problems can help in their resolution. The lists above may help with this task.

Change control and measurement can help to isolate changes and alert team members when they happen. Monitoring environmental variables, saving database contents and using configuration management can all be used to avoid or mitigate problems.

Maintaining data with automated, repeatable (and debuggable) scripts reduces the number of data maintenance faults.

Using small, well-known and comprehensive datasets as described above makes test maintenance simpler and quicker, and may also reduce the likelihood of errors.

Testing the Data

A theme bought out at the start of this paper was 'A System is Programmed by its Data'. In order to test the system, one must also test the data it is configured with; the environmental and setup data.

Environmental data is necessarily different between the test and live environment. Although testing can verify that the environmental variables are being read and used correctly, there is little point in testing their values on a system other than the target system. Environmental data is often checked manually on the live system during implementation and rollout, and the wide variety of possible methods will not be discussed further here.

Setup data can change often, throughout testing, as the business environment changes – particularly if there is a long period between requirements gathering and live rollout. Testing done on the setup data needs to cover two questions;

- Does the planned/current setup data induce the functionality that the business requires?
- Will changes made to the setup data have the desired effect?

Testing for these two questions only becomes possible when that data is controlled. Aspects of all the elements above come into play;

- The setup data should be organised to allow a good variety of scenarios to be considered
- The setup data needs to be able to be loaded and maintained easily and repeatably
- The business needs to become involved in the data so that their setup for live can be properly tested

When testing the setup data, it is important to have a well-known set of fixed input data and consumable input data. This allows the effects of changes made to the setup data to be assessed repeatably and allows results to be compared. The advantages of testing the setup data include:

- Overall testing will be improved if the quality of the setup data improves
- Problems due to faults in the live setup data will be reduced
- The business can re-configure the software for new business needs with increased confidence
- Data-related failures in the live system can be assessed in the light of good data testing

Conclusion

Data can be influential on the quality of testing. Well-planned data can allow flexibility and help reduce the cost of test maintenance. Common data problems can be avoided or reduced with preparation and automation. Effective testing of setup data is a necessary part of system testing, and good data can be used as a tool to enable and improve communication throughout the project.

The following points summarise the actions that can influence the quality of the data and the effectiveness of its usage:

- **Plan the data for maintenance and flexibility**
- **Know your data, and make its structure and content transparent**
- **Use the data to improve understanding throughout testing and the business**
- **Test setup data as you would test functionality**

References

Webpages and online papers:

Cohen, Dalal, Parelius, Patton: The Combinatorial Design Approach to Automatic Test Generation http://www.argreenhouse.com/papers/gcp/AETGissre96.shtml, Telecordia Technologies, pub. IEEE Software September 1996, pp. 83-87

Tai, Lei: A Test Generation Strategy for Pairwise Testing
http://www4.ncsu.edu/~kct/paper/c14.pdf
Department of Computer Science, North Carolina State University

James Bach's Perl pair test tool ALLPAIRS at www.satisfice.com
http://www.satisfice.com/tools/pairs.zip

Updates

Updates to this paper may be found at www.workroom-productions.com

Version 1.1 2/3/2002	Updated links to Tai, Lei paper Added link to Bach tool