

Deployment and Testing

Cross-domain Testing

Why this is important

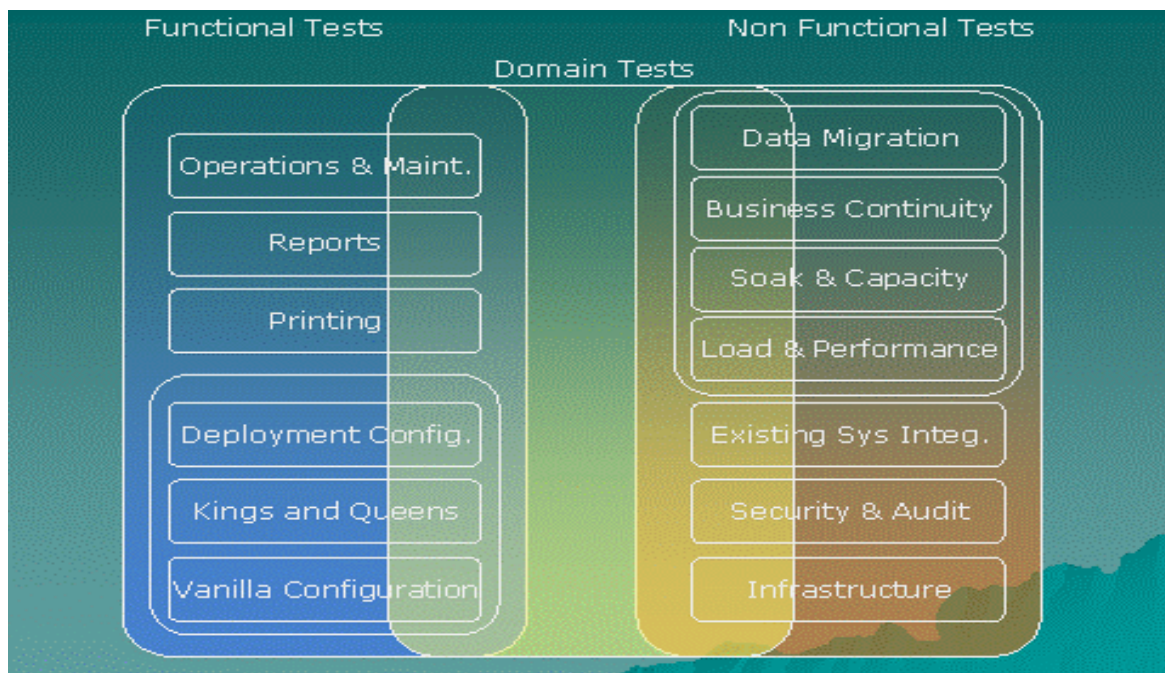
NHS IT organisations are experienced in the complexities of implementing new and in particular, replacement healthcare management software. A key element in planning for success is the thoroughness of the product testing. The structure of the Connecting for Health (CfH) contract and the attendant arms length relationship between software supplier and the NHS Trust makes the testing and certification of product readiness a particular challenge.

Complex CfH solutions, with multiple trusts sharing domains, require detailed and precise testing. Inadequate testing can lead to errors in patient handling which pose major safety issues.

Testing Principles

System testing can be considered as part of the discipline of software engineering. Research and analysis over many years has characterised testing activities, and provides a number of principles to assist with the process, as part of the goal of ensuring projects and programmes are successful. It is important to remember that it is mathematically impossible to test exhaustively any substantial piece of software. It follows that large software systems will never be entirely “bug free”. This does not prevent the production of usable, highly functional and safe software. It does mean that chasing perfection is doomed to fail, and some level of latent errors is to be expected in operational software. These may remain undetected for many years, only to emerge when a change in procedures or data values causes the software to change behaviour.

A second key observation of software engineering is that a defect discovered earlier is substantially cheaper and easier to fix. Common measures suggest that defects found at the specification phase (by validation activities) are 1,000 times easier to fix than those discovered when the software is in operation. Defects found and fixed during system testing are still 10 times easier to fix than those that arise post go-live.



Deployment and Testing

The diagram above illustrates the areas of testing for a complete system and divides tests into functional and non-functional areas. Each area is examined to ensure the new systems are fit for purpose. The functional areas are broadly those where the software behaviour is specified in terms of outputs for a given input. The non functional areas are those where the specification requires performance that does not specify functionality. For example performance may be specified in terms of transaction times, regardless of the specific functionality of a transaction.

The diagram covers the full scope of testing activities at the top level. What this top level analysis immediately reveals is that every area of testing includes domain level considerations. Thus each area needs to be examined to discover if additional domain level testing should be specified.

Configuration Control and Baselines

Configuration control is the key to effective testing. In principle any software or data change is capable of affecting the operation of any part of the system. Software architecture and design practice seeks to minimise this effect. Examination of architecture should allow one to understand when items may impact outside their immediate area, and when data may impact software performance. However the present state of the art does not allow 100% confidence in this analysis and so all previous testing is potentially compromised by any change. For this reason it is essential that the software and configuration data are “frozen” during testing whenever possible, and that all changes are done in a controlled fashion and documented. The version of the software and test data is usually know as a “baseline”.

Measurement during testing can reveal the extent to which software is “globally coupled” (the chance that a localised change has far reaching effects), and the chance that data is “control coupled” (data changing the way the software behaves). These measures provide objective guidance to the sensitivity of the system to changes, and hence the extent to which testing needs to be repeated following any change.

Coverage and Completeness

While testing can never be complete it is important to measure the extent to which test cases have covered both the required functionality of the system, and the software components that comprise it. Completeness of functionality tests ensures that all required functionality has been implemented and works as expected. Completeness of code coverage ensures that all software has been exercised at least once (though not exercised in all possible configurations). Completeness of functionality tests is verified by cross reference of test cases and scenarios to the user requirements. Completeness of code coverage usually requires instrumentation of the software, and is thus best achieved during development at the module test level. It should be noted that systems in use tend to be operated in a very small range of the overall code coverage, so that test coverage levels of 100% are both unnecessary, and probably detract effort from useful other tests that could be carried out. What is required is to achieve sufficient coverage of the likely “flight envelope” in which the software will normally operate, and to exercise sufficient of the alternative elements to be confident of reliable operation in real use

Testing Overlap

Testing overlap is the extent to which tests carried out for one Trust need to be repeated for other trusts. At one extreme all Trusts could carry out the same test scripts, but this limits the overall coverage to the subject of these scripts. At the other extreme Trusts could carry out completely distinct scripts, which would maximise the coverage, but might fail to reveal any domain, or control data issues between the Trusts. A mixed approach is recommended, but the best bias is unclear (and is dependant upon the cost of repeating a test, which in turn is modified by automation).

Test Case Design

Test case design can be carried out using a “black box” or “white box” approach. The black box approach makes no assumptions about how the system is implemented, and simply seeks to exercise the software across its full range of activities. The black box approach tests software in a uniform fashion, and has an “equal” chance of finding any particular issue. The white box approach makes use of design knowledge to generate test cases most likely to reveal issues in the software. Because the white box approach relies on human skill and judgment it is subject to missing the “unexpected” errors, but it is much more powerful in examining complex code areas. It is recommended that a mixed approach is used, with at least 25% of all test cases being generated by a black box approach.



Deployment and Testing

Erroneous Input

Erroneous input cases should be included in all tests, to ensure that input validations are sufficient for everyday use. Human error rates when typing and selecting are too high for tests without this to be truly representative. These tests must include incorrect selections, and essentially random keyboard inputs. At the extreme there is the test tool “Fuzz”, which involves generating a pseudo-random stream of input. This test does not seek particular expected results, but rather checks input validation, and overall software robustness. By its random nature it may find security flaws or defects which are very difficult to expose by other methods. It is none-the-less repeatable, and this assists in identifying and rectifying issues discovered.

Suspend and resume criteria and re-baselining

Continuing testing in the presence of errors may not be possible, if the errors prevent test scripts from running. Even when it is possible the fact that errors need to be fixed means that other tests are potentially compromised by the software changes carried out. The act of fixing errors may introduce new errors, or may be incorrectly carried out. Also unfixed errors may “mask” other errors, which will only become apparent when the first error is fixed. This means that even if a complete set of tests is run, and then all the errors are fixed, there may be new errors when a retest is carried out.

The testing approach can seek to minimise testing effort, or to minimise the elapsed time to test software. Minimising the effort usually means stopping testing after any significant error, and only resuming when the error is fixed. Minimising elapsed time will usually mean continuing testing, and carrying out the preparation of fixes in parallel, with the fixes being applied to the system under test only after the maximum number of useful tests have been carried out on the first baseline. This approach typically requires additional test and development environments, and substantially larger resources.

Repeatability and Regression Testing

Due to the fact that software or data changes may induce new issues testing should be repeated following any change to ensure that the software has not “regressed”. Such regression may be observed in the failure of tests which previously passed. It is normal to repeat a proportion of tests that have previously passed to check for this. Ideally all tests would be repeated, but statistical analysis can be used to provide a confidence level for non-regression, given repetition of a sample of tests. The need to test for regression means that it is essential that previous tests can be repeated exactly as before. The wish to minimise regression testing means that it is valuable to keep statistics on the emergence of new errors following a fix. These statistics also provide an objective measure of the maintainability of the software. Automated testing tools allow regression testing to be carried out quickly, thoroughly, and with minimum human intervention. Such tools may also allow testing to be scheduled for out-of-hours time which would otherwise be unused. The use of automated testing tools does have some disadvantages, including the need to setup and maintain the tools, and in particular to setup the expected results. However, long term software engineering statistics suggest that on average each test will be repeated 6 times during a system deployment. For this reason automated testing is to be preferred whenever the effort and costs involved in setup and maintenance of the tests is less than the effort to run tests six times. Automation may be considered even when the effort exceeds this, because of the testing discipline it imposes.

Reporting Metrics

The reporting and documentation of tests is described in the Generic Deployment Test Plan and the Generic Deployment Test Specification. In addition, previous deployment test reports must be assumed will be included in all future reports. These include root cause analysis for errors, and trend graphs. This data should be used to help reduce errors in future deployments, and to concentrate testing on areas of known issues.

