

Fifty Ways to Improve Test Automation

Planning and Management

1. Define objectives	Be clear how test automation is to help the project. Avoid confusion and competing efforts.	E.g. faster testing, reduce test cost, run more tests, repeatable testing, unattended testing.
2. Quantify objectives	Ensure objectives are measurable in a relatively easy way. Removes subjectivity.	E.g. How much faster / cheaper? How many more tests? How many unattended hours?
3. Agree objectives	Ensure buy-in from project team and managers (stake holders). Successful automation is most easily achieved when everyone pulls in the same direction!	There may be more than one objective and they may change over time. Ensure they support project goals.
4. Assign specific responsibilities	Testing and automating tests are two separate tasks. Project management must make it clear who has which responsibility and when.	Testing software is one responsibility, automating tests is another. Improving test automation methods is yet another.
5. Objectively measure benefits	Don't assume everyone will see and understand the benefits of automation. The chosen metrics must be appropriate to the stated objective(s).	Equivalent Manual Test Effort (EMTE). Hours of additional testing. Others metrics include average number of tests per hour, average effort per test case run, number of tests run.
6. Measure build effort.	The effort required to automate one or a set of (related) tests (as proportion of EMTE).	E.g. Hours to add new or existing manual tests or as factor of EMTE: e.g. "2 times EMTE".
7. Measure failure analysis cost	Effort spent analyzing the cause of an (automated) test failure (i.e. obtaining information to complete a defect report).	E.g. Average hours (or minutes) per failed test case. Can be captured in defect report.
8. Measure maintenance costs	All effort spent updating automation testware to new version of software.	E.g. % test cases requiring maintenance, average effort per updated test case, % of EMTE.
9. Specify targets for reducing costs of automation	If no targets are specified, don't expect improvements! Identify most significant costs and target improvements.	E.g. Decrease build effort by 10% per year. Maintenance effort measured as percentage of EMTE not increased.
10. Improve capabilities	Modern tools and ingenuity can support richer testing.	E.g. New types of test. Give more flexibility, better reporting.
11. Adopt service provider attitude, broaden scope of automation.	Seek opportunities to help testers become more efficient. Identify greatest opportunities and easy wins.	Automate tester chores with tools and utilities purchased, developed or freeware.
12. Pilot project	Give opportunity to experiment. Specific improvement goals.	1 to 3 months, 2 or 3 people. A few goals.

Scripting

13. Develop scripting standards / guidelines	Consistent and sensible script layout and style greatly improve ease of understanding (by others) thereby reducing costs.	E.g. Template with standard header incorporating user documentation, skeleton script or common structures.
--	---	--

Fifty Ways to Improve Test Automation

Scripting (continued)

14. Measure script attributes	Knowing typical range of measures for scripts means outliers (those with a different measure) can be checked.	E.g. Lines of code (LOC), complexity, structure, fan-in, fan-out,
15. Provide tool support for scripting	To check conformance to standards / guidelines of some key aspects. Static analysis to measure and help eliminate common scripting errors. Searching / reporting.	E.g. checks template used, header exists, comments used, not too long, not too complex. Extracts user documentation to compile a catalogue of scripts.
16. Peer review script samples	Openness to discussing scripts constructively aids learning and helps ensure consistency.	Make them fun to do and keep them short (10 – 20 minutes?).
17. Design scripts for reuse	A little more effort put into script design can save a lot of scripting effort in the future.	Generalise, parameterise. Incorporate default input values, more checking (be defensive).
18. Actively choose appropriate scripting techniques	Don't stick with a single approach if other approaches will be more effective in some situations.	Structured programming works well on a small scale. Data driven is often more effective. Keyword driven offers a big return on a fair investment.
19. Apply configuration management	At least implement version control with some traceability.	Particularly important for large and dispersed groups.
20. Monitor script use	Record statistics of script use. Map dependencies between scripts and other scripts, tests and applications.	E.g. number of tests using each script.

Comparison

21. Identify definitive comparison requirements	Each application will have a finite number of comparison requirements (unique set of 'how to' instructions). Likely this will change little over time.	E.g. One set of comparison instructions required to compare any telephone bill for a residential customer.
22. Design useful comparison processes	Add intelligence to the comparison process where this will help the tester.	E.g. Comparator knows where to find expected result so needs be given only the actual result file.
23. Standardise comparisons	There is no value in having different people each solve the same comparison problem.	Having a particular test output having a choice of comparisons will not help.
24. Minimise tester work	Different capabilities may be required for different tasks.	E.g. Match/No Match answers only to verify test result. Details of differences if analysing fail.
25. Support stand-alone comparison	For use with manual testing and with further analysis of automated test results.	Can be a great help to manual testing. Automated comparison is much faster and can be 100% reliable.
26. Refine comparisons	Intelligent comparison ignores expected differences. More intelligent comparisons check the format or content of what is to be ignored when appropriate.	E.g. Check date is in the correct format, check date is a valid date.

Fifty Ways to Improve Test Automation

Comparison (continued)

27. Report only significant results	Tailor comparator output to the needs of the testers.	Comparison report detailing which files were compared can be redundant.
-------------------------------------	---	---

Pre- and Post-Processing

28. Recognise and formalise setup and clear up tasks	Setup and clear up tasks are many and similar, come in packs, and mostly can be automated simply.	E.g. Copy, move, delete files and directories. Compress or decompress files. Extract data from a database.
29. Automate setup and clear up tasks	Done manually, they can be error prone, particularly when done in a rush.	Testware setup prior to an overnight run of automated tests can cause the run to abort if not 100% correctly.
30. Design for reuse	Generalise, generalise, generalise.	E.g. Sensible default input values that can be overridden to support different tests.
31. Implement standard solutions	Encourages reuse and reduces build and maintenance costs.	More effort can be invested in a single solution to create more powerful functionality.
32. Automate non execution tasks	These make up the difference between automated tests and automated testing.	Automating the execution of a test is not great benefit if 80% of the time to run it is spent setting up data for it.
33. Use as entry and exit criteria for test	Only execute a test if the pre-processing (entry criteria) is successful. Pass a passed test only if post-processing (exit criteria) is successful.	E.g. If an input file is missing (entry criteria) there is no point in running the test anyway. If an unexpected result file (exit criteria) is produced something has gone wrong so fail the test.
34. Design simple interfaces	Use default input values where appropriate (and as much as possible).	This will encourage their use.

Testware Architecture

35. Plan the architecture	Don't let it evolve blown by the wind. Consider how current structures will work with other applications and with many more tests.	Automation projects work well with small teams regardless of architecture but these generally do not scale up or last when new people become involved.
36. Define a structured and consistent testware organisation	Structured rather than haphazard is easier to learn and work with. Consistent is important to leverage further benefit of tool support.	With thousands upon thousands of testware artefacts (scripts, data files, expected results, actual results, etc.) serious thought needs to go into it.
37. Organise testware around testers	Organise testware to the advantage of the testers (those working with the testware day to day). Don't conform to an arrangement suggested by a tool if it is not the most convenient.	E.g. Test execution automation tools frequently require scripts in one place, expected results in another. Such a simple scheme may not work well with most teams.
38. Build knowledge of architecture into tool support	Many chores associated with testware can be tool supported thereby eliminating human error and, frankly, much tedium.	E.g. A tool that knows where to find all the scripts doesn't need to be told the (potentially many) locations by the tester.
39. Adopt clear naming conventions	Take the guesswork out of deciding what someone named that script!	Also helps to reduce build cost since we don't take time out to decide what to name it.

Fifty Ways to Improve Test Automation

Testware Architecture (continued)

40. Keep all testware under configuration management	Best when tool supported and done with a CM specialist. Home grown tools can work well and a simple spreadsheet approach can help to start with.	Main aim is to prevent silly mistakes and ensure traceability. E.g. Disallow overwriting old versions and be able to track all changes to specific versions.
41. Be consistent, don't diverge	Even if you know the structure is wrong. Updates are much easier when all testware is structured in the same way.	The more consistency there is the greater use can be made of each piece of tool support.
42. Keep all references relative	Don't hardwire machine or pathnames into testware. Use relative references, symbolic or logical references.	Important to be able to relocate some or all testware without causing any but the simplest of updates.

Testware Maintenance

43. Manage and monitor testware maintenance	Understanding exactly how much effort is spent where and what the maintenance work involves is a requirement for keeping it manageable.	Even when it is a relatively small cost, monitoring the costs will give early warning of when it starts to turn into a problem and (hopefully) why.
44. Design tests mindful of maintenance	Learn from maintenance work carried out and apply the lessons when automating.	E.g. Access GUI through standard set of scripts.
45. Measure maintenance costs	Most frequent costs, largest costs (what takes greatest effort) as proportion of EMTE.	Consider ways of combating most frequent and largest costs.
46. Learn from maintenance experiences	Some maintenance problems are unpredictable.	Review and discuss the maintenance work carried out (debrief).
47. Provide tool support for maintenance tasks	Many updates to testware are similar across large numbers of files. Providing some tool support can pay dividends.	E.g. Adding new column to spreadsheet data can be undertaken by a batch script.
48. Include cost of testware maintenance in plans/proposals for software updates	This requires knowledge of typical costs for common software updates. The true cost of a software update may outweigh its value.	More informed decisions should follow!
49. Good pilot topic	Because maintenance usually done piecemeal in a rush. Pilot gives opportunity to research typical and possible future changes and solutions.	A little ingenuity can go a long way but it has to be given the opportunity to develop.
50. Make maintenance an upfront activity	Not a backend or background activity. Raise its profile. Proactively learn / seek lessons.	It is not a job for someone to do in their spare time. This belies its importance.