

7 ground rules for Smart Test Automation

**TEST 2008
INDIA**

Authors

Abhishek Talwar(abhishek@adobe.com)

&

Abhinav Agarwal(abhinava@adobe.com)

**Adobe Systems India Pvt. Ltd.
Noida**

Table of Contents

Topics	Page Number
Abstract	3
Introduction	4
Be futuristic & pro-active	4
Be damn precise	5
Be comprehensive	7
Be smart	7
Be tightly coupled with host application	7
Be enterprising	8
Be presentable	9
Conclusion	10
Bibliography	10
Boography	11

Abstract

Automation today is an integral part of every testing project. Automation is of various kinds and works under its own set of guidelines. Automation needs an environment to run on.

This paper tries to present some scenarios which can make your automation testing smart, for any kind of automation, within the automation guidelines and the environment on which automation is capable of running on. Usage of these tricks can help you automate some impossible to automate looking test cases.

This paper is divided into 7 logical units which we call the '*Seven ground rules for Smart Test Automation*'

- 1) Get ready in advance for the next feature
(for example if the current product supports 'n' formats for export today, support should be given in such a generic way that change in number of supported format, the UI, the parameters supported shouldn't cause a huge break in software. Also, next format added can be supported without a single line of code). Here we introduce the concept of 'target the module' which is least likely to be changed approach)
- 2) Catch difference of even a millisecond for every test run. Every drop adds to the ocean. Small mistakes could lead to bigger problems of future.
- 3) Catch exceptions at every possible level (not just at test case level, but try to go to every validation line of your automation code). This way you have implemented a debugger for even release builds.
- 4) Don't re-invent the wheel (petty things should be handled commonly and with industry standard ways). The petty tasks in this generation of computing are almost 'bug free', use them and be sure of some part of code to be bug free ☺
- 5) Be as native as possible – use the support exposed by the product as much as possible to get most out of the product
- 6) Try to be more intelligent than the product you are working on (find simpler and more efficient ways of automating the product, in layman's terms use smart workarounds wherever possible)
- 7) Customize your result logs to the "ground zero" level. This is the only thing which the upper management wants to see, make it customizable to the lowest level. In short work hard on checking you have correct results AND you report them properly!!

Using these seven smart techniques your automation is bound to get better.

Welcome to the world of STAT (Smart Test Automation techniques) ...

Introduction

Automation has come a long way, earlier people just used to automate 'repeatable' tasks. Now they want to automate every task which is not 'impossible' to automate.

Core graphics features in layout software are no longer non-automatable.... Multiple platform (Win/Mac/Linux) automation scripts using the object model are no longer considered to be a humungous project.... Zero click automation (test once scheduled, runs on every new build and on every platform supported, doing every task all by itself) is no longer considered as a dream project.... Stitching of multiple automation tools to enable usage of best features from each tool is no longer considered a tough task....

In short, there are very few things which are now defined in the purview of 'impossible to automate'- doing all these things has been possible because of the smartness that has been incorporated in the way in which automation testing is implemented. Our paper tries to present 7 of those features which we think can help carry out a complex automation project with minimum pain and a smooth glide path to product release.

Be futuristic & pro-active

Be damn precise

Be comprehensive

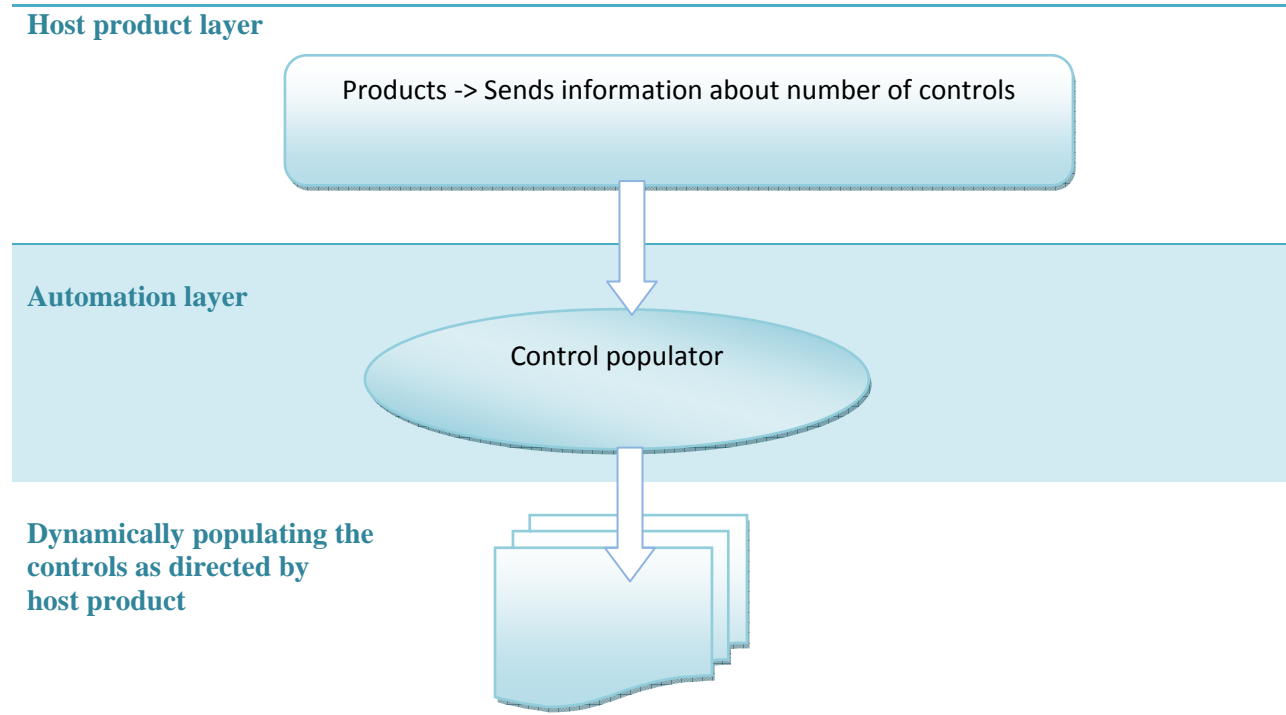
Be smart

Be tightly coupled with host application

Be enterprising

Be presentable

Be futuristic & pro-active:



Get ready in advance for the next feature. User expectations from a product keep on constantly increasing. The products themselves support numerous languages, platforms & flavors. This puts a strain on the PDLC as a whole. As is the case with anything in the world, better planning can help provide long term solutions and that too with minimum amount of effort. Automation too follows the same principle. One needs to not just think of how things can work now, but on how things world work in the next cycle, or next to next cycle. Truly, things are way too dynamic to be predicted 100% accurately, however smartly predicting them would definitely be better than just chalking out plan on the present requirements and specifications. To demonstrate the above facts lets take some practically seen examples:

Illustration 1: A user can save a file in 'n' formats in version 1.0 of the product. However, as new formats keep on getting added every cycle. Normal approach involves tackling each of these newly added formats on 'as they come basis'.

Suggested approach: Use your code to count the number of 'save' formats at runtime, the number of formats user can save to, assign them an index (ID) and use this index (instead of the name) to save the document to each format that 'save' supports. This approach would help the user code require zero maintenance cycle over cycle. Also, this approach would work even for certain software which have different flavors with different number of 'save' formats.

Illustration 2: Most users use the default settings / recommended settings for different product features. A pro-active approach of verifying feature usage with default settings in a headless way, not just covers the most common scenarios, it also helps in faster certification of build. A mechanism thus needs to be build so as to verify the 'most commonly used' product areas in a headless way.

Be damn precise:

Catch difference of even a millisecond for every test run.

Today's customer is loyal till he gets his solution the way he wants. Earlier automation was used primarily for regression test workflows, however automation is expected to step into the cases where manual testing techniques aren't capable or reliable enough.

- 99% accuracy might not be fine with people developing satellites
- Memory leak of key kilobytes might add up to take considerable memory for repeated actions
- It is very important for a layout software which is used to print multi-page posters to print each part-page with exact precision.

Catching these manually is not possible, so automation needs to step in for this.

Illustration #1: Measuring time to export a file in milli-seconds

Table below shows a result which has accuracy up-to the level of milli-seconds, also, not just failure but the broad failure reason has been flagged, in addition a pixel by pixel comparison is also carried out ONLY for 'export status' pass test cases. All image comparison results below a given threshold value are marked fail, along with the % deviation.

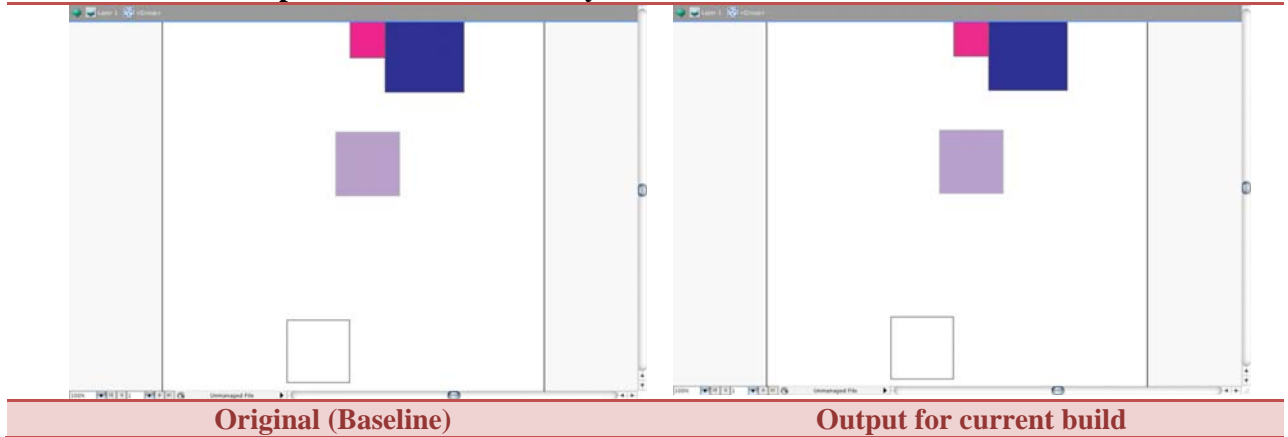
Test Case ID	BuildNo	Date & Time Stamp	Time in milli-seconds ToExport	Export Status	Image Comparison Status
TC_1	192	Wed Oct 11 02:17:15 IST 2006	0	FailEncodeError	NA
TC_2	189	Fri Oct 06 20:00:34 IST 2006	1186188	Pass	Pass
TC_3	187	Thu Oct 05 02:04:31 IST 2006	0	FailEncodeError	NA
TC_4	1000	Fri Mar 23 15:29:37 IST 2007	10888656	Pass	Pass
TC_5	364	Tue May 08 13:01:43 IST 2007	15720703	Pass	Pass
TC_6	328	Mon Apr 02 23:56:29 IST 2007	15408328	Pass	Pass
TC_7	310	Sat Mar 10 05:54:01 IST 2007	11810469	Pass	Fail(0.02%)
TC_8	282	Fri Feb 02 14:59:05 IST 2007	0	FailMicrosoftError	NA
TC_9	280	Wed Jan 31 17:20:34 IST 2007	0	FailEncodeError	NA

TC_10	233	Wed Nov 29 21:41:14 IST 2006	5876671	Pass	Fail(0.02%)
TC_11	231	Tue Nov 28 13:34:38 IST 2006	0	FailEncodeError	NA

Illustration #2: in graphic critical applications, 99.5% is a failure!!!
 {Manually 'false pass' cases detected very effectively by automation}
RESULT SHOWN AS FAILURE EVEN WITH 99.75% ACCURACY

Results	Date	Action	Result	Compared		
	2008-07-02 19:46:21	Compare	Mismatch	Image/Diff		[annotate]
Pixels	Baseline	Job	Result	Diff		
	Macintosh_FMP_10373-CheckPoint2.png	Macintosh_FMP_10373-CheckPoint2.png	Mismatch	99.57351%		

Difference is non-decipherable for the naked eye



Be comprehensive:

Catch exceptions at every possible level

Code, of any nature, is a piece of individual's thinking put into logical steps. Each logical step has a logical output. Since the automation has the single biggest role of validating things, each of the logical outputs should help in achieving this goal. So much so, that even a harmless assert should be caught and shown to the user of the automation software. This way we catch some of the error hints which have not yet shown up as errors, but are potentially dangerous piece of code which can cause a error. This way we are able to catch 'hidden' errors early in the cycle and thereby reducing the project cost (earlier the bug caught, cheaper it is to fix). This also helps in dividing the complex set of operations in set of small components which can be given to the user as common functions as configurable properties. For example, if in an imaging software, for a user to draw a shape the following needs to be done.

- 1 users has to select a shape
- 2 then user has to go to a particular location
- 3 finally user has to drag-drop the shape

In this case it is better to define exceptions for the subsets instead of just the overall outcome. This is preferred because of the following three factors:

- 1) The exact cause of the problem can be found

- 2) The sub-steps can be re-used by other parts (for example in the example above, step 2 and 3 are same for all different shapes).
- 3) Some problems might go un-reported in case of complex data

Be smart:

Don't re-invent the wheel

This is an old saying, but still holds a lot of value in current scenario. The world of software development is now object oriented. One of the very important aspects of Object oriented concepts is the concept of re-usability. Re-writing a database iterator to read records of a file might be a cool exercise, but an inefficient code written by you might slow down execution of automation run when done on real time data with a large number of fields. Similarly, writing a nested menu list with menu items, sub-menus and leaf nodes might work fine for all cases you could think of, some cases might have been left un-handled because you simply missed their existence. (like reading the child menus of a disabled parent menu).

Be tightly coupled with host application:

Be as native as possible

- Get into the code, instrument the code for memory leaks, code coverage
- Identify the object, not the UI elements (much more effective and scalable)
- Check the strength of your product's core API's in addition to just the product workflows

The various types of testing which we can achieve by being native are:

- Scripting
- Headless version
- Product specific feature automation

Scripting

Being native we are at the same level as the host product. This gives us the unique power of being the only automation player capable of giving directions to product's scripting framework as a product would do. The best we can achieve by being non-native is to acquire 'powers' equivalent to that of the scripting plug-in. Being one step ahead gives us unprecedented power of passing custom arguments, setting priority of processes and in some cases even modifying the scripting action functionality if needed.

Headless version

Being a head entity inside a 3rd party software is tough. In short, any non-in-built Automation tool/ automation software claiming to be headless is not truly headless. Surely, it doesn't have to pop up its UI on the top of the product, but it has no way to get the internal triggers and signals from the host software. This makes it handicap in knowing when the product is actually done. Things become even more messed up when nothing is shown on the UI to the user as a result of the action. In the above cases, the only fool proof solution is being native. Not only does it 'get' all the information automatically, it's power can be exemplified in the sense that it can, if need be, modify product control passing. This gives it a unique advantage over the rest of the tools.

Product specific feature automation

Image automating a graphic software makes a custom shape like arrow or circle. Using normal automation we have no way other than the UI route to do so. And as we all know, using the UI route would induce lots of screen coordinate dependency and hence would neither be scalable nor be fool-proof. In case of being native one has the control in the form of an exposed “drawArrow” or an “drawCircle” API and the results are same as that we get from the host product. Not even a single pixel difference across platforms, across machines or across browsers.

Be enterprising:

Try to be more intelligent than the product you are working on

Gone are the days when the following quotes used to hold.

“100% testing coverage manually is not possible, one needs to use automation”.

Now-a-days 100% coverage of test cases in present day complex products with multiple platforms, environments, devices is not possible even with automation. A simple control which takes single values might have thousands of valid test cases (as might open up an array of new scenarios when each of those values is inputted) and of-course quite a few negative ones. Same application may behave differently in different environments. Errors prone areas might become so humungous that checking each and every scenario might not be possible with automation also. The need of the hour is being enterprising. Test the maximum without actually slogging.

A product exposes innumerable number of dialogs, a large number of controls, large number of interactions and a host of ‘cool’ but tough to test features. Do remember that a product is written to make life of user easier and no consideration is given to what it entails to the level of effort required for the test automation engineer. On the top of it add the complexities of multiple platforms, environments, devices. It is therefore required for the automation engineer to out-think the product. Think, how can one achieve maximum coverage without getting bogged by the factors listed above.

Some of the best practices to do this are as follows:

- Don’t just do black box testing, run your automation tool on *debug* builds, each assert thrown by the product is important and if ignored might be a potential blocker or show-stopper.
- Use tools to do tasks like memory and resource leaks instead of simulating the repeated execution manually. In addition to that, don’t just use the product, use the instrumented version of the product code to get to the root cause of the problem. Catching a leak in generic check-box control is much more effective than finding a leak in a form which contains 10 other fields, one of them being a check-box control.

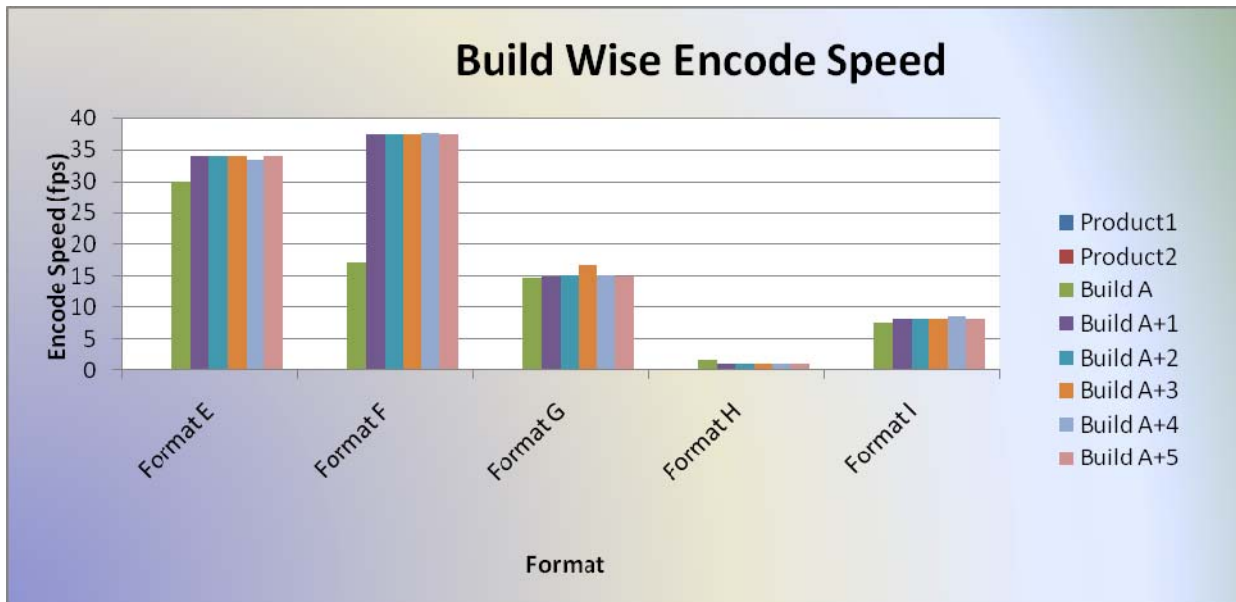
Study the product, know more about the core architecture. This would help to find the prone areas.

Testing is all about maximum coverage... try out innovative ways to achieve maximum coverage....

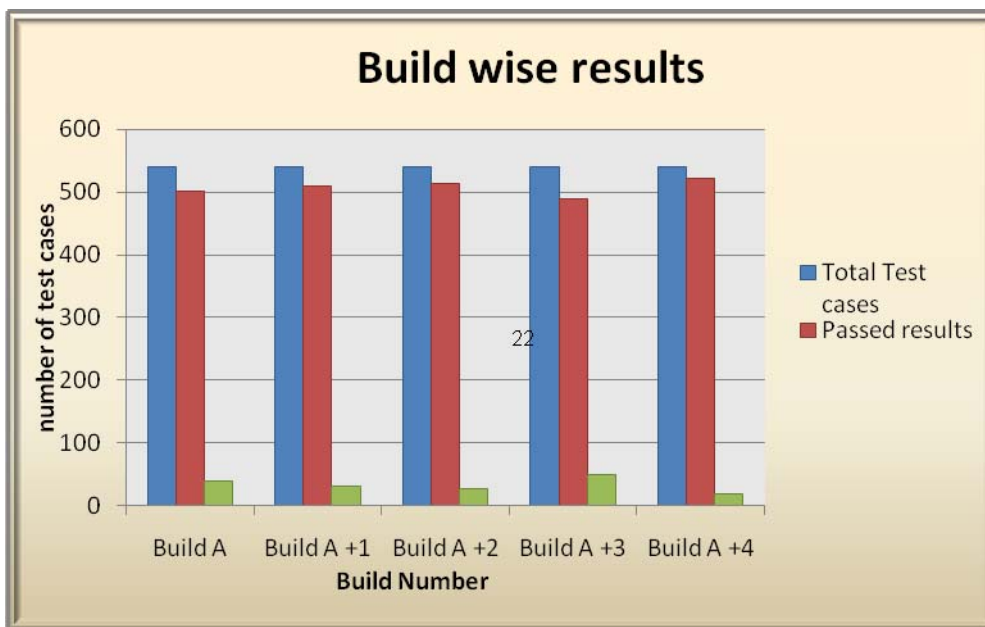
Be presentable:

Customize your result to the “ground zero” level

- What might pass for 1 test case might fail for another. (example of 99% case)
- When a file is saved, do check if the file content is correct, assuming same sized file without parsing is a bad thing....
- Graphical results are always easier to analyze and to view
- A power user should be able to analyze the results to the maximum possible detail



Encode Speed comparison for different builds for multiple formats



Graph depicting no. of test cases passed/failed for automation runs on successive product builds

Conclusion

Automation is such an equipment in the hands of a Quality Engineer, that if used effectively can help not only to make the quality of product better build over build but also make the life of the entire product team much less stressful in this challenging software development environment. Using the 7 ground rules mentioned above, a Quality Engineer is surely going to get more effective!!

Bibliography

This paper has been written using the various experiences of the authors in the field of the automation during their combined stay of around 10 years at Adobe across products in different domains.

No websites/books have been used as references for this paper per-se

Biography

Abhishek Talwar is Lead Software Engineer at Adobe since last 4 ½ years. He has a total experience of around 5 ½ years in various domains like Video, Imaging and Telecommunications. He has worked on various fields of testing including White Box testing, Automation, scripting and Black Box testing. His current role involves developing tool which make the life of a tester, at Adobe, easy. Abhishek holds an international patent to his name in the field of video. He also holds a patent publication in the field of Documents. He has written five international paper publications in the field of API testing, Scripting, testing best practices, and project management. One of his submissions “API Testing: Catching hidden bugs early in cycle” was presented at the STC2005 main conference in Hyderabad. One of his papers on Quality mantras was voted as the best paper in Adobe India QE summit, 2005 and was also chosen as an article in QE quarterly newsletter.

Abhishek holds a B Tech degree in Information Technology from University School of Studies, Delhi and MBA (finance) from IMT Ghaziabad (Distance Learning). He is an ‘Indian Testing Board’ certified foundation level tester.

Email: Abhishek@adobe.com

Abhinav Agarwal (abhinava@adobe.com) is Software Quality Engineer at Adobe since last 3 years. He has worked in various domains like Video, Imaging and Codec .He has worked on various fields of testing including Automation and Black Box testing. Currently he is single handedly responsible for all codec related R & D testing tasks where he is handling the entire QA alone.

Abhinav holds a B Tech degree in Electronics Technology from Harcourt Butler Technological Institute, UP along with below certifications



CERTIFIED EXPERT
Premiere®



Software Testing

#5922406