Through the Testing Glass

Pedro Pereira, Márcio Neves, António Sá Pinto, Bruno Ferreira, José Pedro Tavares
Telbit
Aveiro, Portugal
{pmpereira, mneves, asapinto, bferreira, jtavares}@telbit.pt
www.telbit.pt

***Abstract*—Despite the growing trend regarding software quality (or should we say...lack thereof it), in a non-neglectable part of the industry, methodical and systematic testing is still an undervalued relative on the development life cycle.**

Based upon our field experience, in this article we shall try to dissect some of the main causes of this (mis)practice and propose some practical answers to overcome the daily restraints which emerge on the way of those willing to deliver better quality software.

Along the way, it will also be described the anatomy of a software testing tool, its accomplishments on the field, what we learned from its deployment on several organizations and also to explain why — in our perspective — a tool like this is suitable to ease the burden of software testing in the real world.

***Index Terms*—software testing, software development lifecycle, software quality assurance.**

Conflict of interest statement: Telbit (the authors' affiliation) develops a product — TeStudio — which has the following mission: to provide software development teams the ability to deliver quality software products without the hassle of making them aware of a painful and heavy Software Quality Assurance (SQA) process. TeStudio includes a module to manage, design and automate software tests — TeStudio Automation Lab.

I. Introduction

A. The blue pill[1]

Edward Stanley wrote in 1873:

> "Those who think they have not time for bodily exercise will sooner or later have to find time for illness."[2].

The previous quote regarding the care and wellness of a complex system — the human body — has a huge — although maybe not so obvious — corresponding analogy on our industry.

A century later, Boehm [2] described a similar effect on the Software Development Life Cycle (SDLC): neglecting software quality in the early stages of development will dramatically increase the cost of products' maintenance and operation once deployed in the wild, i.e. the later the defects are found, the more they cost to fix.

So, being this statement nowadays so consensually obvious, what is it that makes systematic testing so unwilling to be included as a first class duty on the SDLC? Are most of the software development managers mad? Yes they are. Even thought, given the current industry mindset they would be even madder, if they didn't act the way they do.

Software development stakeholders (seem to) have serious and very rational arguments for this kind of behaviour:

- Besides critical systems like the ones built for medical devices, the space and aeronautics industry[3], etc., defects are (still) somewhat tolerated in a software product as long as it — although far from perfect in a "defect free" sense — still brings value to the end user.
- Since most of the times, tests are not "deliverables" by themselves[4], this usually leads managers — when deadlines approach — to shrink the allocated time to their execution (if and when there were time allocated to that purpose).
- Given the urge to deliver and the tightening schedules, (some) *exploratory/ad hoc* [4] or *smoke testing* [6] usually (seems like to) provide the best cost/benefit option.

B. The red pill

In the classic *No Silver Bullet — Essence and Accidents of Software Engineering*, Brooks [1] stated:

> "From the [software development] complexity comes the difficulty of communication among team members, which leads to product flaws, cost overruns, and schedule delays. From the complexity comes the difficulty of enumerating, much less understanding, all the possible states of the program, and from that comes the unreliability."

[1] *The Matrix* (1999) — Andy Wachowski, Lana Wachowski. In this movie, the question of which pill to take illustrates the personal aspect of the decision to study philosophy. Do you live on in ignorance (blue pill) or do you lead what Aristotle called "the examined life" (red pill)?

[2] quoted in the May 2004 issue of the Harvard Men's Health Watch

[3] even thought neither these ones are immune to severe malfunctions caused by faulty software [13]

[4] *software test factories* are a counter example, since test plans and their execution results are the provided service's main outcome

In the same article, the author presents the reasons why is software one of the most complex human creations. Given the embryonic state that Software Engineering discipline still presents, we must accept that testing cannot prove by itself the absence of software defects [3].

Despite that, it surely can — depending on its coverage extent — give valuable and measurable hints regarding the match (or mismatch) between what software is supposed to do, and what in the end, it actually does. Moreover, testing can also rise to the surface what it is that the System Under Test (SUT) is *not expected* to do, and actually does.

Our real world experience, reveals the huge — and unreal — gap between day zero, when a company only performs ad-hoc testing and the day when it intends (and actually starts) to begin systematic testing. Therefore, our proposal is to accomplish this with a soft, cumulative, non disruptive and self-rewarding investment.

We must state that, this gradual and self sustained path, was itself a pattern we saw emerge on the behaviour our users in their different business contexts, and it goes like as following:

1) Turn requirements into test cases: More or less formally defined, controlled and tracked, almost every software system, has on its base a given set of functional requirements.

So, the very first step is to generate at least one test case record for each requirement. At this stage, we should not expect these test cases to be very fine grained since their main purposes (at least for now) are: provide a basis to start tracking along the way which tests already run and to stamp their pass/fail outcome results.

This small step — a tiny one — brings out:

- a centralized *testware* repository (although an incipient one) which enables knowledge sharing through all the SDLC stakeholders
- a reusable test set able to be the groundwork for the upcoming testing cycle, preventing the "start-all-over-again-from-scratch" syndrome
- requirements/tests/defects tracking capabilities

2) Breaking down tests into steps: After some iterations appending and fine graining tests, it turns out very clear that tests share some common steps like:

- Set the SUT to a known state
- Perform some kind of authentication
- Do a bunch of actions
- Make some SUT's properties validation
- Tear down/clean up the system

The natural way to manage this fact, is to start decomposing tests in their atomic steps. These steps, may start as an informal/textual script to guide testers on the actions they have to perform to accomplish the test.

Therefore, test designers end up building a step library which allows them to create new tests mostly by (re)composing a subset of already existing steps, intertwining them in a suitable workflow. Tests end up being an always update receipt to follow with some checklist like facilities.

In this stage, the information system by itself also encourages testers to attach the output results collected by tests' executions for future reference/auditing.

3) Steps automation: So far, our tests and steps have been "helpers" providing guidance to the testers through the tasks they have to perform — manually — to run the tests.

It is time to turn on the automation engine and have some of these steps automated, freeing testers to more intellectually attractive tasks.

In many cases, developers and testers already have handy scripts/small utilities (batch files, shell scripts, sql scripts, etc.) in their daily testing/development weaponry.

As tests are built from steps' composition, the next stage is to set these up with the above mentioned scripts. From now on, every single test making use of these steps will run and collect their output in an automated way.

With testers now (a lot more) free from time constraints, it is now feasible to progressively "arm" each and every test with a discretionary number of steps to perform tasks like:

- verify if the system is in a well known state
- check environment properties
- capture log files
- run database dump/restore scripts
- setup services configuration
- etc.

Since these are the most likely reusable steps, it should be obvious by now that, along the way, the "effort" to automate these step types is well worth it. The main reasons are:

- virtually every test makes use of them
- usually grab precious information for future defects' debriefs
- frees testers from the burden of having to run manually half a dozen different tools/utilities

to effectively run a test and grab manually their outcomes
- dramatically increase the overall testing process determinism without adding significant runtime overhead

4) Steps evaluation: In the previous stage we described the steps automation benefits (focused on automatic execution and output data grabbing). In this one we will take automation even further extending it to evaluate the steps outcome.

Until now, it was up to the tester, to observe all the steps output, looking for clues regarding the system well/malfunction. Frequently, testers find themselves looking after the same common patterns when tests are being executed.

So, the point here is — when applicable — define an oracle [8] which performs this "observation" for the tester and stamp a pass/fail outcome on the step's execution. We will further discuss this mechanism on section II-B.

Like all previous path stages, this one can also be taken progressively. Being tests composed by reusable steps, when an oracle is defined for one step — each and every related test benefits from it. Moreover, this oracle can (and should) be refined along the way — tightening the evaluation criteria — making the test set even more effective on its SUT evaluation process.

Due to the automation degree achieved somewhere near by this stage, the test set is ready to allow (fully automated) regression testing. This is actually a huge step towards the delivery of better quality software products.

Our experience shows that this last stage is probably the steepest one, since sometimes its far from trivial to figure out a way to describe formally all the SUT's expected outcomes. Despite that, the trend we have seen so far, is that our users' testing sets are pursuing and steadily approaching the full automation, at least in a given subset of their requirements base.

C. The path's side effects

At this point, it should be clear how — step-by-step — and in a feasible way, we gradually evolved from a point where only some ad-hoc/exploratory testing was performed, to a point where we have a test set handy to scale — in its testing strength — up to our needs and available resources.

Besides all the above mentioned benefits, following this path also brings along:

- a more controlled and deterministic testing process
- automated regression testing
- a safety net under which we cannot fall
- a "de facto" SUT functional specification
- a non neglectable peace of mind sense regarding the way our system is evolving
- a set of useful and always up-to-date software quality metrics

II. The framework scaffolding

Given our field experience, in this section we will share some of the distinguishing features that — in our perspective — a software testing tool has to provide (and ours hopefully does) in order to overcome some of the challenges presented by testing software applications.

One of the main concerns when deploying this kind of tools in an organization, derives from their potential to create some entropy. We managed to overcome this issue creating some features which mimic the typical workflow of a tester when he does not have a specialized tool to help him design and execute tests. Some of them will be briefly described here:

A. Steps heterogeneity

As an example, when a tester needs to test a "create new user" requirement in a web application meant to make the users provisioning in a information system, a possible test for this requirement could composed by the following steps:

- step 1: use firefox user agent to navigate to the provisioning page and create a new user
- step 2: run a database client to check that there is a new record in the users data table
- step 3: collect the application server and database logs

Each of these steps are in their essence very distinct from each other.

Every step type should be eligible for composing a test case, e.g., steps meant to query a database may run simultaneously with steps drawn to collect a log via a tail command executed via a ssh connection while concurrently, there can be steps performing actions on a Graphical User Interface (GUI).

Since this is what actually happens on a hand testing session, we believe that a testing tool should allow that too.

B. Steps evaluation

If we think a bit about how a tester "manually" evaluates a test, we take out that in most of the cases, he relies on textual content to draw his conclusions about the steps outcome.

Since virtually all SUT's properties (even colors, dimensions, etc.) can be serialized into text, Regular Expressions [9] — with their all mighty power to describe text patterns — seem to be the best option to perform this task.

Therefore, all steps fail/pass conditions are asserted by means of regular expressions.

Despite the terrifying comments sometimes heard when these are mentioned, it should be noted that, regexes are as much complex as the pattern one is trying to match[5].

C. Abstracting steps

Often, we have to perform the same set of actions with slightly different inputs to assert about distinct SUT properties.

Instead of create one step for each of these scenarios — that would lead us to steps maintenance nightmare — we should be able to abstract them as much as we need to.

Therefore, test steps (their scripts, pass and/or fail criteria, etc.) should be parameterizable in a way that, the very same step may be "extended" to perform several validation scenarios. E.g., the same step on one test can validate if a record field in a given table is null, while in another test, it can be parametrized to assert that the same field has a value greater than 5.

D. Piping steps

On some occasions, part of the information needed to perform some steps execution and validation in an SUT is only generated at its runtime. When we do not have a way to know these values upfront, we must be able to grab these from the SUT and feed the following steps with them (e.g. when we need to get a primary key ID from DB record that will be created in runtime).

To overcome this problem, we managed a way — based on a similar mechanism above described on section II-B — to give to the step designer the ability to define a criteria set specifying the content to be grabbed from the SUT. This content may feed the following steps using the mechanics mentioned on the previous section (II-C).

E. When black is too dark

Usually in functional testing — regularly rendered as black box testing [3] — testers should only care about the inputs (actions) and verify the expected outcomes, i.e. they're not worried about the system's inner details.

Despite that, when testers are asked to dive into the systems' guts, they should be able to do that. Therefore, we should provide them the ability to create test steps to drill down as deep as possible in the SUT.

These steps usually take the form of DB assertions, log files screening, etc. and end up to be valuable resources on the debugging process.

F. Looking for the absence of...what?

On manual testing, testers almost always observe the SUT to infer if its execution outcome is the expected one *and*...if nothing unexpected happened.

James Bach states that "Hand testing and automated testing are really two different processes, rather than two different ways to execute the same process. Their dynamics are different, and the bugs they tend to reveal are different" [5] and also that "The exploratory tester must watch for anything unusual or mysterious" [4].

A human tester may be able to detect hundreds of problem patterns and distinguish the harmless anomalies. Despite that, it is indeed an arduous task to define an unambiguous criteria to enable a machine to detect unexpected behavior. Mimic this (intrinsic human) characteristic turns out to be everything but an easy task. How can we specify a criteria to match with "anything unusual or mysterious"?

One way to at least approach and mock this behaviour, is to gradually collect "error fingerprints" and append them to the steps fail criteria. Some of the best candidates are clues like: uncaught exception trace messages, services error signatures (like the famous 404 page on a browser), kernel panic messages, etc..

G. Extensibility

By integrating third party test automation software, testers can combine multiple tools, widely available and some without costs.

For example, one can use Selenium [10], Watir [11], etc. for cross-browser functional tests with sikuli's advanced GUI recognition [12]. Web tests can also run distributed in multiple servers. In fact, the possibilities are so immense that a tester may use it's favorite Load and Performance Test tool along with the best-pick for Web Site Security tool or even just a simple HTML Link Checker.

Communities around the integrated software also provide good support and low-cost

[5]e.g. the regex to match the word Telbit on a given text is...Telbit — How scary is that?

training. It's increasingly common to find people with former experience in these technologies, thus making them a valuable resource. Integration also allows reusing existing tests without throwing them away before acquiring a test framework.

Existing tests that once were only run as unit tests or in a continuous integration environment now become bridged to project's requirements and defects.

H. The Pesticide Paradox

The Pesticide Paradox [7] applied to software testing states that: "Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual."

Since tests are composed by several — and reusable — steps provides our test set an additional resilience degree regarding this issue. Whenever we need to check a SUT's new property/behaviour, we can just tighten the few necessary steps' oracles and after that, all their related tests are — by its nature — adapted to this new circumstances.

I. Mostly Harmless

Given the above stated, we can see that a test is just a bunch of "programs" running in parallel with the SUT, constantly asserting some of its properties.

Therefore, it is not very hard to fall in the trap of adding (unnecessary) complexity to the steps — and their entanglement — in a way that it becomes uneasy to analise what it does...*begging to be debugged*! When these symptoms start, a bell should ring on the testers mind and immediate measures should be taken to prevent this handicap.

One of the (hard) choices we made to minimize this threat — which we took very seriously — is to deny forks in the steps workflow avoiding cyclomatic complexity.

Another symptom we usually notice is when misconfigured tests start to interfere in the SUT's behaviour.

III. Field Results

This section presents more than three years of data created by TeStudio deployment on a subset of our users. These customers develop software mainly to other software development organizations (regarding SDLC tools) and also for the telecom industry.

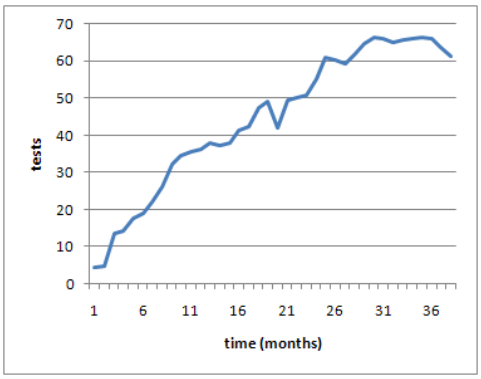

Fig. 1. Average number of tests where a step is (re)used

The figures here presented are based on an sample of one hundred and fifty thousand tests, created and edited by approximately sixty users.

Figure 1 shows the evolution of steps reuse along time. We can see that the steps reuse rate has been increasing and by now, on average, each step is part of more than sixty tests.

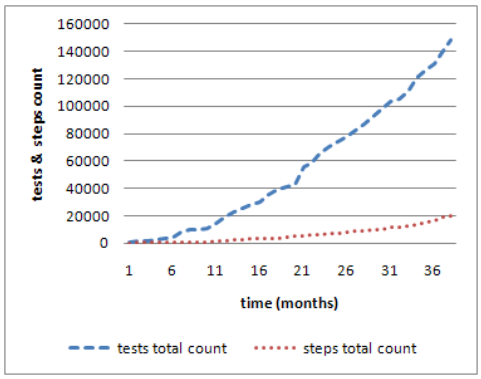

Fig. 2. Tests total count VS steps total count

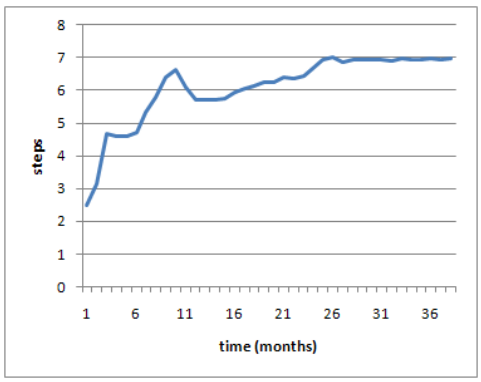

Fig. 3. Average number of steps by test

On Figure 2, it is shown that the rate at which steps are created is almost one order of magnitude lower than the test creation rate.

Since the average number of steps per test has not been decreasing — as can be drawn from Figure 3 — this is an empirical proof that our approach favoring the tests composition from a step library was successful.

It should be noted that, although steps are the most expensive item to design and automate, each bit of investment on these testing assets will be divided by each and every test where they are part of.

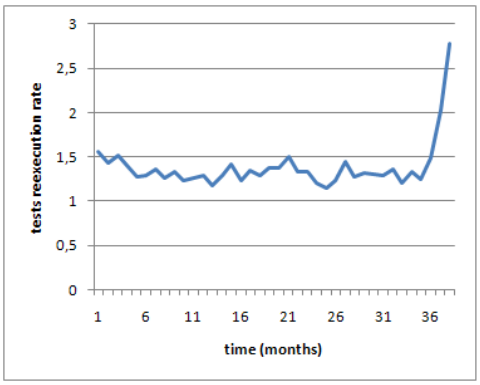

Fig. 4. Average test re-execution rate

The chart on Figure 4 shows the evolution of the average number of times each test is executed. We can see a breaking point near by the 34^{th} month. We can relate this huge turning point to a serious investment made on steps automation — and consequently — on tests fully automation.

There is actually a very strong correlation between these fully automated tests and their higher re-execution rate values.

IV. Future directions

Despite all the achievements already made, we continue our efforts to develop an even more mature tool suitable to ease the testing pain. In a short/medium term, our focus will be:

- minimize the tool-mastery burden — decreasing the required skill sets to effectively design, automate and evaluate test steps
- improve the information system facilities in a way to ease the steep learning curve for untrained testers
- provide users more guidance about what's happening at tests runtime

V. Conclusion

Unless a Silver Bullet is meanwhile found — which seems unlikely — software testing is not *the* antidote for the mismatch between what the system is built to do and what it actually does.

This mismatch eradication seems to be an arms race between the always rising software systems complexity and the degree to which we are able to maintain their development under control.

Systematic testing provides a way to steer and keep the development process inside safer trails in a cost effective way.

In the end, the main purpose is, by all means, *not to* copycat *Sirius Cybernetics Corporation*[6], which among other devices, built the famous *Nutrimatic* drink dispenser machine. This device when asked for a cup of tea, produces a beverage described by its consumers as *"almost, but not quite, entirely unlike tea"*.

Acknowledgments

We cheer Telbit's mindset which enables us to spend some of our time on the "not so run-of-the-mill-daily" tasks and provide us the opportunity to Think! about far more than the ordinary issues.

We must also thank our clients, partners and end users for their valuable nurture and feedback, which provides us a fertile soil for the sustained growth and maturity of our products.

References

[1] Brooks, F. P., Jr. *No Silver Bullet: Essence and Accidents of Software Engineering*, IEEE Computer (20:4), April 1987, pp 10–19.

[2] Boehm, Barry W. "Software Engineering Economics"; IEEE Trans. Software Eng., Vol.10 No.1, Jan.1984, pp.4–21.

[3] Myers, G.J. *The art of software testing*, Second Edition revised and updated by Tom Badgett and Todd M.Thomas with Corey Sandler, John Wiley & Sons, Inc. 2004

[4] Bach, James, *Exploratory Testing Explained*, The Testing Practitioner, E. van Veenendaal, ed. UTN Publishers (2003)

[5] Bach, James, *Test Automation Snake Oil*, 14th International Conference on Testing Computer Software, Washington, USA.

[6] McConnell, Steve, *Best practices: Daily build and smoke test.* IEEE Software, 13(4):144, 143, July 1996.

[7] Beizer, Boris *Software Testing Techniques*, 2nd Edition 1990.

[8] Baresi, L. and Young, M., *Test oracles*, Technical report, Dept. of Comp. and Information Science, Univ. of Oregon, 2001. http://bit.ly/9fkDwJ (as of 2010 July)

[9] Friedl, Jeffrey E. F. 2006. *Mastering Regular Expressions*, 3rd edition. O'Reilly

[10] Gheorghiu, Grig *A look at selenium*. Software Quality Engineering, 7(8):38–44, October 2005

[11] P. Rogers, B. Pettichord, and J. Kohl. *Watir: Web Application Testing in Ruby. Technical report*, 2005.

[12] Yeh, T., Change, T.-H. and Miller, R.C. (2009). *Sikuli: Using GUI Screenshots for Search and Automation*. Proc. of the ACM Symposium on User Interface Software and Technology (UIST 2009), 183-192.

[13] Nuseibeh, B. *Ariane 5: Who Dunnit?* (1997) IEEE Software, 14(3):15–16.

[6]Douglas Adams (1978), *The Hitchhiker's Guide to the Galaxy*