EuroSTAR Software Testing Community

Automation through the Back Door (by Supporting Manual Testing)

Seretta Gamba

case study

AUGUST 2013

Foreword

After I had been asked to contribute to the new EuroSTAR library of testing case studies, I immediately thought about the chapter that I had written for the book by Dorothy Graham and Mark Fewster (Experiences of Test Automation, January 2011). Since then though, I have been working on a new project, a book about Test Automation Patterns that was actually inspired from that very book. After reading the chapters of the other contributors, I realized that all of us had used some of the same "patterns", that is, used similar solutions to solve similar problems. Since I didn't find any book on the topic, I decided to write one myself.

Fast forward to 2013. I have been able to involve Dorothy Graham and Mark Fewster in my book project and we have already catalogued a good number of Test Automation Issues and resolving Test Automation Patterns. So what I want to accomplish with this case study is to show how I had tackled the problems described in my chapter in the book without patterns and how I would address them now, with my new knowledge of patterns.

Introduction

In my chapter (21, Automation through the Back Door), I first describe how we started with test automation, which strategies we applied, and then how we solved the progressive stagnation by supporting manual testing. I will follow the same structure here, but I will first introduce the issues that we had to address and then the patterns that we applied or that we should have applied. Note that we used the patterns not knowing that we did. I will briefly explain issues and patterns as we need them, but for more details, you will have to wait for the book! Or contact me, see Conclusion for details.

Background for the Case Study

Around 2001 my company decided that we needed test automation and I was charged with its introduction and maintenance. Starting with test automation is one of the first issues that we catalogued (NO PREVIOUS TEST AUTOMATION[1]). To solve it we suggest doing a number of patterns:

1. SET CLEAR GOALS: This pattern is critical. It must be applied at the beginning of any big or small automation effort
2. MANAGEMENT SUPPORT: Also critical. Automation efforts that are driven only by a lone hero are much more apt to stall because nobody else knows about it or can use and maintain what has been developed
3. DEDICATED RESOURCES: This pattern is especially important at the beginning of a new automation effort. Depending on the size of your automation you can later slacken its use
4. RIGHT TOOLS: This pattern is essential for long lasting automation
5. AUTOMATION ROLES: Use this pattern to fill the roles you need to develop the automation testware. If possible use a WHOLE TEAM APPROACH.
6. PLAN SUPPORT ACTIVITIES: Don't forget to apply this pattern if you want to be able to keep your schedules. Missing support from specialists can ground a project pretty effectively!
7. MAINTAINABLE TESTWARE: Apply this pattern from the very beginning if you want your automation effort to be long lasting and your maintenance costs low
8. AUTOMATE WHAT'S NEEDED: This pattern shows you how to select the features most worthy to be automated
9. TAKE SMALL STEPS: This pattern is especially important in the beginning, but it should always be kept in mind
10. UNATTENDED TEST EXECUTION: This pattern gives you a goal to work towards in your automation

Did we use them all? Well, not completely and that was actually one of the reasons for stagnation later on. Let's examine what went on.

- SET CLEAR GOALS: we defined what we wanted to automate first, the regression tests for our registration product[2]
- MANAGEMENT SUPPORT: the project was initiated by management because it was recognized that we could not afford to lose customers just because our testers were overwhelmed by regression testing
- DEDICATED RESOURCES: this is one pattern that we completely underestimated. I was doing all the automation work on the side-lines of my regular work! Not good!
- RIGHT TOOLS: we selected a Capture-Replay tool, QARun, which could easily drive our application. We were warned by the sellers though that a new product was being developed and it would not be compatible with the old one
- AUTOMATION ROLES: this was also a pattern that we ignored. I was tester,

[1] *Issues are shown in ITALICS CAPS and patterns in CAPS*

[2] *My company develops standard software for insurances. We offer products for registration with the German finance authority (BaFin) and to manage assets or portfolios for an insurance company*

automator, developer, project leader…all in one!
- PLAN SUPPORT ACTIVITIES: another missed opportunity. In the beginning I didn't need very much support, so I didn't care to plan for the future and it came back later with a vengeance
- MAINTAINABLE TESTWARE: since I come from development this was the pattern that I applied best
- AUTOMATE WHAT'S NEEDED: this worked out pretty well (at least at first)
- TAKE SMALL STEPS: yes, I used this one too
- UNATTENDED TEST EXECUTION: and of course this was our target (and we reached it)

To recapitulate: with no knowledge of patterns, we did quite a lot right, but what we ignored came back later to haunt us.

I think the most critical pattern at this point was MANAGEMENT SUPPORT. Since the project had been started by management (my boss) I believed management support a no brainer. But there is more to this pattern. Let's examine in detail some of the pattern suggestions:

- Build a convincing TEST AUTOMATION BUSINESS CASE. Test automation can be quite expensive and requires, especially at the beginning, a lot of effort.
- A good way to convince management is to DO A PILOT. In this way they can actually "touch" the advantages of test automation and it will be much easier to win them over.

We didn't develop a business case. I didn't realize then how important it was and since my boss was bought-in on test automation, he also believed it wasn't necessary. The problem is that without a business case there is no real commitment from management to continue to support test automation when for instance development projects come to need the same resources. Just as for testing in general, test automation doesn't come first!

The pattern DO A PILOT also offers quite a number of important suggestions:

- First of all SET CLEAR GOALS: with the pilot project you should achieve one or more of the following goals:
  - Prove that automation works on your application
  - Chose a test automation architecture
  - Select one or more tools
  - Define a set of standards
  - Show that test automation should deliver a good return on investment
  - Show what test automation can deliver and what it cannot deliver
  - Get experience with the application and the tools
- Try out different tools in order to select the RIGHT TOOLS that fit best for your Software Under Test (SUT), but if possible PREFER FAMILIAR SOLUTIONS because you will be able to benefit from available know-how from the very beginning.
- Do not be afraid to MIX APPROACHES
- See that you get the people with the necessary skills right from the beginning (AUTOMATION ROLES).
- TAKE SMALL STEPS, for instance start by automating a STEEL THREAD: in this way you can get a good feeling about what kind of problems you will be facing, for instance check if you have TESTABLE SOFTWARE
- Take time for debriefing when you are thru and don't forget to LEARN FROM MISTAKES
- In order to get fast feedback adopt SHORT ITERATIONS

We performed most of the suggested patterns (again without knowing it), but I think that the most grievous error was that we never stopped to look back at what had been achieved, what could have been done better and so forth, that is we forgot the debriefing and just went on as before. In this way, as I already mentioned, we missed some important patterns (the most important was AUTOMATION ROLES) and didn't even notice it.

Our Technical Solution

Since it is important to understand how we did our automation, I will describe in more detail the pattern MAINTAINABLE TESTWARE. In order to develop automation that will be long lasting this pattern suggests applying some other patterns[3]:

- DOMAIN-DRIVEN TESTING
- GOOD DEVELOPMENT PROCESS
- GOOD PROGRAMMING PRACTICES
- OBJECT MAP

Applying the pattern GOOD PROGRAMMING PRACTICES came naturally to me (as I already mentioned I come from development and I still work as a developer). It suggests that since scripting is a kind of programming, you should use the same good practices as in software development.

[3] *The patterns are in alphabetical order and not necessarily in order of importance*

The recommended patterns are:

- DESIGN FOR REUSE
- KEEP IT SIMPLE
- SET STANDARDS
- SKIP VOID INPUTS
- Separate the scripts from the data: use at least DATA-DRIVEN TESTING or, better, KEYWORD-DRIVEN TESTING
- Apply the DRY Principle (Don't Repeat Yourself). Also known as DIE (Duplication is Evil)

Even without recognizing the patterns for such, we applied every one of them.

Another pattern that I applied, even if it is not suggested by MAINTAINABLE TESTWARE, was GET TRAINING: since I had never before worked with test automation, I got all the available books; I searched all possible internet forums and studied how to work with our chosen tool. In this way I came "automatically" to develop a variation of the pattern KEYWORD-DRIVEN TESTING[4], which in fact is one of the patterns suggested for the implementation of DOMAIN-DRIVEN TESTING. This variation we called Command-Driven Testing.

Command-Driven Testing

Since we wanted to reduce to a minimum what we had to implement in the language of the test tool (we knew from the beginning that it would be changed in the near future) we decided to use keywords that were not domain terms but just simple commands (like INPUT or SELECT) and that thus didn't need any application specific implementation. To change the tool we would need only to implement these "primitive" commands in the language of the new tool. An important side benefit was that we could use the same interpreter scripts for all our products, even if they belonged to other domains and were implemented in different environments. Another payback for this system is that we can use different tools concurrently since only the internal interpreter changes, not our functional scripts.

[4]KEYWORD-DRIVEN TESTING tells you to add to each line of test data a keyword that drives how it should be processed. You then need a framework that provides the expected functionality for each keyword. This is usually implemented with libraries in the script language of the test tool. For every line of data that has to be executed the framework must call the library script that implements the keyword. Generally the keywords are domain specific terms that imply some more or less complicated processing.

We didn't stop there. I had studied the advantages of the pattern DATA-DRIVEN TESTING[5] and devised a way to get the same benefits: we split our command scripts into a DRIVER part and a DATA part. In the DRIVER-File the variable data is replaced with placeholders and in the DATA-File the placeholders are substituted with the actual data. In this way, as with DATA-DRIVEN TESTING, we can have any number of DATA files to one DRIVER file.

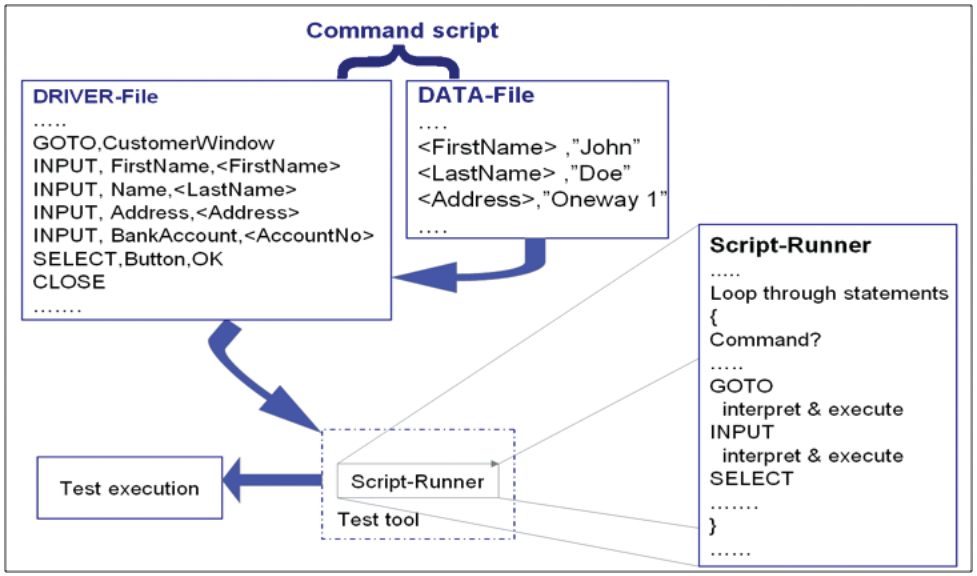

Figure 1[6]

Figure 1 details how this works out in practice: the interpreter script (Script Runner) executes the DRIVER script sequentially. Every time it finds a placeholder, for instance <LastName> it looks for the data in the corresponding DATA file. In this case it would substitute it with the word "Doe". To make the scripting even more flexible the interpreter script ignores statements that contain placeholders that are not to be found in the corresponding DATA file[7]. After this substitution all remaining commands (for instance INPUT) are translated in the internal scripting language of the tool and executed.

I realized pretty soon that in order to implement Command-Driven Testing I would need an appropriate framework (yes, this is a pattern too: TEST AUTOMATION FRAMEWORK). I wrote it first as a small utility program, but, as more and features were added, it evolved into a full-fledged framework (ISS Test Station). One of the first features implemented

[5]With DATA-DRIVEN TESTING you extract the variable data from your scripts, so that you can use the same script for any number of similar test cases. The data is usually stored in some external file that is read by the script
[6]This figure has been taken from my chapter in the book Experiences of Test Automation
[7]Btw this is also a pattern, SKIP VOID INPUTS

was an automatic translation of captured scripts into our DRIVER / DATA formalism. In this way the scripts can be developed quite efficiently:

1. The capture functionality of the tool is activated
2. A tester performs all possible actions on a window of the application, that is touches all the GUI-elements on it
3. With modern tools (for instance TestComplete) the tool automatically maps the objects while recording. With older tools they must be registered in the proprietary object map[8] of the tool beforehand.
4. The capture functionality is deactivated and the script (if necessary) exported to a text file)
6. To create new tests one just has to copy the template to a new DATA file and adjust the data accordingly (that is modify what has to be changed and eliminate what is not necessary for that particular test case)
7. The scripts are created window-wise (another pattern, SINGLE PAGE SCRIPTS) and are subsequently assembled together to form a test case
8. Similar test cases finally are grouped together to form test suites

The framework also supports development and maintenance of automated test cases and starts the script interpreter in the selected tool to perform and log the tests.

Developing this framework was thus done using (without knowing it) even more patterns, for instance CAPTURE-REPLAY, DATA-DRIVEN TESTING, GOOD PROGRAMMING PRACTICES, KEYWORD-DRIVEN TESTING, TOOL INDEPENDENCE and DESIGN FOR REUSE.

You probably noticed that some patterns show up repeatedly. This is one of the charms of patterns: you don't need a lot of complicated ones, but just a number of simple ones that you can combine to describe quite complex solutions.

Day by Day Experiences

After all this had been implemented, I stopped doing the automation myself and started to train and coach colleagues from the various projects. Here also we implemented some patterns and, not knowing better, left some important ones out.

[8] *The name for the object repository can change from tool to tool, but every tool provides the means to register the objects. Giving objects the same names in every used tool is a precondition for tool independence (pattern OBJECT MAP).*

One issue that we disregarded was INADEQUATE TEAM. We did use AUTOMATION ROLES as suggested and, again as suggested, used GET TRAINING, but we could not select the team and had to manage with colleagues who were not needed for development. The pattern (AUTOMATION ROLES) warns that one needs special skills for the different automation roles and we had completely disregarded it. Of course it should be just sensible to choose the right people, but in my experience one often thinks, that it will work out somehow.

If you have a pattern, that at least confronts you with the problem from the beginning. You cannot say afterwards that you didn't think about it.

Another pattern that we more or less ignored was FULL TIME JOB. As I already mentioned I did most of the automation work on the side-line, actually because I found it really interesting, I just couldn't let it go. But in this way I apparently let management believe that everybody would be doing it just like me and we lost a very good and experienced colleague, because she was not willing to sacrifice all her free time for the job.

In one project we got a developer turned tester that knew her application perfectly and had the "developer" mind-set that is just perfect to develop good automation. However she automated her manual test cases as is, so that her test cases turned out to be long and complicated scenarios where it was difficult to locate the problem once they failed. Here it would have been better to use patterns like KEEP IT SIMPLE, ONE CLEAR PURPOSE or EASY TO DEBUG FAILURES.

In another case we had somebody who did know the application, but had no idea and no aptitude for development. And to make things worse, he had obviously never heard of a pattern called ASK FOR HELP! When he found a problem, he never said a word and just waited until somebody would ask him how it was going. Only then he would tell that he hadn't been able to do anything for x days because of some problem!

Of course we also had a newcomer that already had experience with KEYWORD-DRIVEN TESTING and her automation was impeccable. She used all the right patterns (DESIGN FOR REUSE, KEEP IT SIMPLE, ONE CLEAR PURPOSE, READABLE REPORTS etc.), again without knowing that she did!

One pattern that we applied with success was GET ON THE CLOUD. Our company is based in Hamburg, Germany, but many colleagues live in other cities (like Berlin or Cologne) and all work from home regularly. We put our automation on the cloud and in this way it makes no difference where you work. Another advantage is that two persons can

connect to the same virtual machine and can in this way PAIR UP[9] even if they are located in different cities. Being on the cloud is also cheaper so one can have an extra machine for every operating system or database. In this way it's also easier to have DEDICATED RESOURCES.

Another pattern that we applied even though we didn't realize that it was a pattern is SET STANDARDS. We defined one set of standards that is applicable for the whole company, but gave the different projects the liberty to choose project internal standards. This also has the advantage that in our periodic test management meetings we can hear how it works out for them and the other projects can profit from their experience (another pattern SHARE INFORMATION).

One of the most important standards that we defined was that every test case had to be independent from all others (and of course this is a pattern too, INDEPENDENT TEST CASES) and had to prepare its own setup (FRESH SETUP). In a few cases we allowed exceptions, but by and large all projects complied with this rule.

Another standard that has proven valuable was to automate each window independently (yes, another pattern, SINGLE PAGE SCRIPT). In this way when our test cases move thru several windows we have a standardized modularization.

As I mentioned before I have written a framework (TEST AUTOMATION FRAMEWORK) to support our automation (ISS Test Station). Having an in-house solution has the advantage that when we had problems automating some part of the SUT where the developers had been particularly inventive[10], we could tweak the framework as needed. To this day we have managed to automate everything we had planned.

Stagnation

Having implemented such a sophisticated framework, we expected the ratio of automated to manual tests to steadily grow. Instead we faced stagnation in one of our most important products. What happened? This product was way too successful! It sounds crazy, but the fact is that for the small company that we are, what is outstanding for marketing can be deadly for test automation. The reasons are simple:

- INADEQUATE SUPPORT: Developers and testers were needed for customer projects. Guess what our management decided when confronted with the choice between earning money now and expanding test automation for better regression tests in the future!

- NO INFO ON CHANGES, INADEQUATE COMMUNICATION: The application was updated again and again. The automation was not informed about the new features and needed help from the testers. They on the other hand didn't have time to support the test automation project that would have helped diminish their own testing effort (yes, a typical catch-22 situation!)

Looking for a solution

Having understood the issues (note that we hadn't categorized them yet) I started to look for possible solutions. The main problem was how to get the needed information from the testers without forcing them to "loose" time with automation problems.

I guessed that a way could be to get this information as a side-product of what the testers had to do anyhow, manual testing To verify this hunch I had to extract from the testers how they prepared and executed their manual tests. So I implemented, again without knowing it, the pattern, SHARE INFORMATION. Here is what it suggests to do:

- ASK FOR HELP when you have a problem or a question: you should never ponder too long on some issue, other people may have already solved just the same question

- Listen to testers or developers. Ask why they do something and why they do it as they do. If you find out what they really need, you can support them even better than you were planning

Let's take a look at what our issues would have suggested to tackle this kind of problems.

- INADEQUATE SUPPORT:
  - o MANAGEMENT SUPPORT: I did try to apply this pattern, but, as mentioned above, without success
  - o PLAN SUPPORT ACTIVITIES: this should have happened much earlier, at this point I didn't have a chance

[9]The pattern PAIR UP tells you do pair automating (like pair programming in agile development)
[10]This issue we called HARD-TO-AUTOMATE

- NO INFO ON CHANGES:
  - o SHARE INFORMATION: without knowing it, I had chosen the right approach

- INADEQUATE COMMUNICATION:
  - o SHARE INFORMATION: see above
  - o WHOLE TEAM APPROACH: this pattern works in agile environments. At the time we were not really agile enough for it to be applicable

Most of the patterns suggest getting information on the problem. At this point they cannot be more specific, because the solution will depend on what you will find out.

After coaxing or cheating the testers to answer my questions, I had a good idea of how they were working. They had, for every application area, template spreadsheets with the test case specifications. To perform the tests they would copy the template and perform the tests as specified. They would write the results or eventual updates directly in this new document. For failed tests they were also supposed to open a correction issue in the development control system.

There were several issues that could be improved in this process, for instance:

- Since the testers would write not only the results, but also changes or enhancements to the test cases on their own private spreadsheet, it was later necessary to merge such updates to the original template, a task that was not only boring, but also prone to errors
- When creating a defect issue in the development control system, testers would be pulled out of the test flow
- Often, in order to report the failure correctly, they had to repeat the test.
- If they didn't detail the failures well enough, then the developers had problems locating the bugs
- Metrics combining test execution with defect status had to be compiled manually, because the information was stored in different systems

I decided that the problem could be solved by adding a manual testing feature to our test automation framework. This new application would have to offer substantial advantages to any tester using it (otherwise why take the trouble to change) and at the same time harvest the test execution information for later automation

The next step consisted in examining what features our framework already contained (see Table 1) and what would have to be implemented to support manual testing (see Table 2).

Table 1[11] – Available features

Manual testing issue	Framework Solution	Importance
Maintain test cases	Test maintenance features: add, update, copy and remove objects at every level of the test suite or suite list hierarchy completely integrated configuration management	Critical: this feature must offer the same comfort as in the current process, although in a different format
Overview of the available test cases	Test suites group the test cases in a hierarchical structure that is displayed in tree form. Related test cases can be displayed by expanding the tree nodes. Suite lists group related test suites	Critical: the new process must offer at least the same comforts as the old one. No tester would do without
Logging of test results	Logging information is automatically recorded in separate logging files.	High: separates update functionalities from the logging functionalities.
Result reports or overviews	Result reports, overviews and cross-reference lists can be displayed (or printed) with different contents and different granularities: Command script Test case Test suite Suite list	Critical: by offering statistics that include not only the completion status of test execution but also the defect tracking data, we offered an important incentive to use the framework
Prioritization of the test cases	Priority feature: test cases can be easily prioritized in order to specify which ones are to be executed in a particular test session. This feature allows us to use the same test suites for regression and smoke tests.	High: prioritization was a feature that they valued and would certainly not want to miss.
Defect reporting to development control process	Integration with development control process: New defect-items can be created directly out of the result reports. All known data is automatically transferred, so that testers in most cases have nothing else to do.	High: enables testers to report defects to the development control system with the push of a button

[11] *Tables 1 and 2 are taken from my chapter in the book*

Table 2 - Features not currently available in our framework

Manual testing issue	Solution	Importance
Support for manual test execution	The test cases that should be executed (selected by priority) must be displayed sequentially. The tester must be able to see simultaneously both the test specifications and the application under test (paperless execution) The tester can perform any of the following actions: • Perform the test case and set the test status • Move to the next planned test case • Skip the test case • Interrupt the test session • Continue the test session • Finish the test session • Change the current test case specifications • Create a new test case • Eliminate the test case • Create a defect item • Start test recording • Interrupt the recording • Restart the recording • Stop recording	Critical: if testers don't get this kind of support, there is no way they will ever switch over Note that the recording of the test was both by the test execution tool and a video clip (see point below about what is given to developers).
Import test-template sheets to framework suites	The original test cases must be imported from the current spreadsheets (csv-files) to the test suites in the framework format. The import functionality should be driven by an external table in order to support the different formats of the test-template sheets. The table pairs the original columns with the target fields in the framework suite.	Critical: testers must be able to continue to use their current test cases, although in the new suite format. This step is also crucial to test automation, as it generates nothing less than the structure for the future automated test suites.
Detail test execution for developers	Capture-facility must be integrated into the framework. It must be possible to generate not only a script for later automation, but also some kind of recording (screen-shot or film) that a developer can view without needing the expensive capture-replay tool license The tester can decide if and when to capture the test and for how long. Starting and stopping must be performed directly from the framework without having to call external tools. The recording produced will be attached to the defect tracking item automatically. The script produced is made available to the test automation team for further processing.	High: testers currently do without, but this feature could be a great incentive to actually switch over to the framework. This feature is also crucial for test automation: both the recorded screen-shots (and film) and the captured scripts illustrate exactly the test execution that we want to automate. These can be attached to a defect item for the developer.
Support the creation of input data or compare files	Integrate SQL-scripts that extract data from the database to build initial conditions or compare values for test cases.	High: would enable testers to save the inserted data without breaking the flow of test execution Essential for test automation, because in this way the testers would not only reveal the test case specifications but also deliver the necessary data to create the test preconditions, the expected results or both.

The new manual features

The first thing that got implemented was a feature (written in PLSQL) to extract table contents from an Oracle database. For manual just as for automated testing this was a great help because now one could create a specific condition, export it (with a recognizable name) and use it (that is re-import the data in the database) every time a test case needed such a condition. A typical example was to create partners with specific characteristics. By doing this we were applying (again without noticing) the pattern MIX APPROACHES. It suggests, among other things:

- For every task use the tool that fits best
- PREFER FAMILIAR SOLUTIONS: if possible chose tools that are already in use in your organization and that are well known. In this case we used PLSQL because the development team was using it all the time
- AUTOMATE WHAT'S NEEDED: decide individually for each application if it's better to use test automation, manual tests or a mixture of both. What we automated here was just a kind of utility but it proved extremely helpful for all kinds of tests

Then I implemented the manual testing features described in Table 2. Since, as usual, I was adding these features in my free time, it took some months to get them all running. Actually I did it piecemeal, adding new functions as they were needed. In this way we could already give value even before everything was completed. This again means using a pattern, TAKE SMALL STEPS. One of its suggestions is exactly this: In some cases it can be most rewarding to do just some automation in order to support the testers right away. This will help to produce much interest and support for a later extension of the automation effort.

In this phase I had to select the RIGHT TOOLS to capture the scripts and to do recordings of the tester's actions. Since it was still a kind of experiment, I had to limit the choice to freeware. The other important condition was that the tools had to offer a command line interface so that they could be called from the framework

Last but not least I implemented the feature to migrate the spreadsheets to the framework suite format. Having adjusted the import definition tables in just a couple of hours, the transition was completed in less than a day for all the needed documents. From that moment on the test cases had to be used and maintained in the framework. Now the new manual test process could be deployed.

Test execution with the new manual test process

After migration the test suites looked something like Figure 2[12] :

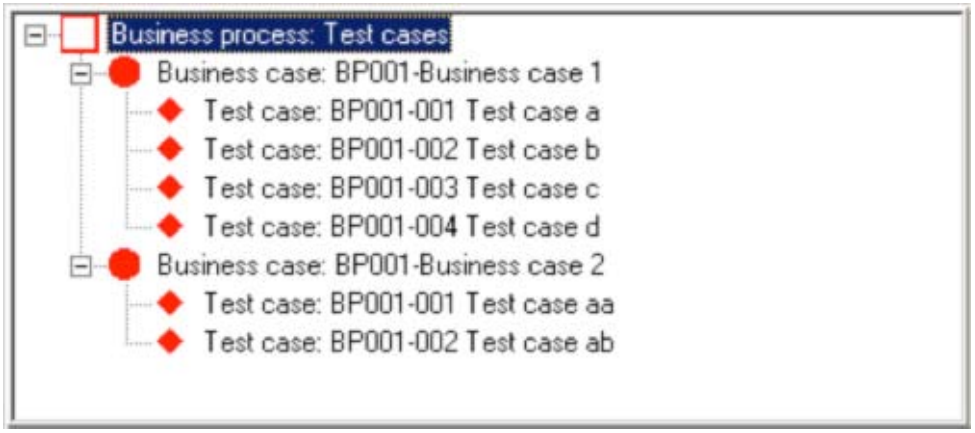

Figure 2

The test execution process now goes like this:

1. A tester starts execution by selecting which test cases are to be performed.
2. A window like Figure 3 pops up with the first (next) test case to be executed. Note that this window has been designed especially small in order to enable testers to see the application they are testing and the test specifications side by side.
3. If the preconditions are not met, they must be set up and the tester has the possibility to export them to external files for later use. To make reference easier the files are automatically given names that contain the actual Test-ID. Note that this is useful to the tester but even more so for later test automation
4. The tester starts the recording functionality (the tester can decide if he needs the recording for the particular test case. We gave the rule that this was mostly necessary only once for a family of test cases)
5. The tester performs the manual test.
6. Afterwards the tester can export the current state of the application. These files can then be used as input for subsequent test cases. And, as a useful side effect (from the automation point of view), they constitute the expected results that will be needed to check the actual results when the test is run next time (if the test has passed).

[12] *Figures 2 and 3 are taken from my chapter in the book*

7. If the test case is completed without defects the tester checks that it's OK (■) and goes on to the next test. The process continues from step 2.
8. If the test case discovers a defect, then the tester can classify it as a failure (■) and create a defect-item in the development process system to have it resolved. Available information is automatically attached to the item, also any recording, so that the tester has to step in only in exceptional cases (to fill in fields that can't be filled out automatically and aren't usually needed). Afterwards the tester can select to perform the next test and the process continues from step 2.
9. The tester finishes the test, and the test results are displayed automatically.

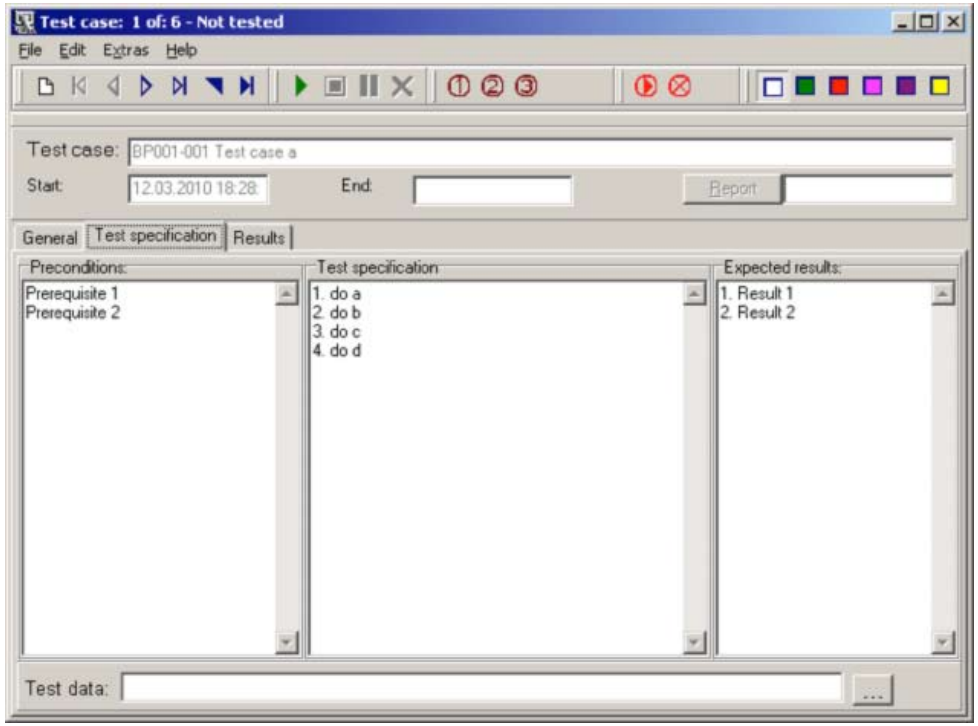

Figure 3

Automating the manual tests

As an example on how automation worked after the manual tests were executed with the framework I want to describe how we set up near shore automation.

A couple of years ago we decided to try to outsource the development of our automation. Since the team from Poland was new to our product, to the tools and to Command-Driven Testing they had to stay at our Hamburg office for about 3 months to get trained. To prepare their training one of the testers migrated a new set of test cases to the framework (see above) and performed them all, exporting previous and resulting conditions. She also recorded the tests as scripts and as videos. In this way the two new "colleagues" would get as much information as possible regarding the application to be automated.

The training schedule was first to get to know the product, at least in the areas where they were supposed to work. Then they were taught how to use the tools and the framework. We gave them an overview of the various features, but they were supposed to learn the details by doing (pattern GET TRAINING). They were also informed about the standards they were supposed to follow (pattern SET STANDARDS).

Finally they could start with the automation work. Here are the steps that had to be done:

1. The manual test suite was split up so that one guy would automate the first half of the test cases and the other one the second half
2. The captured scripts were used mainly to see what had to be done, together with the videos. A more experienced automator (one of us) could have created the DRIVER and DATA_TEMPLATE scripts for all the needed windows in one go, but in order to learn it was better for them to capture the single windows again (pattern SINGLE PAGE SCRIPTS). This was also because the GUI objects had to be mapped and it was easier to use the automatic mapping offered by the tool while recording (pattern OBJECT MAP)
3. The recorded scripts were converted to DRIVER and DATA_TEMPLATE scripts
4. The DATA_TEMPLATE scripts were copied to DATA-Scripts and following the instructions (text, recorded scripts or videos), they had to adapt the data accordingly.
5. The scripts to prepare the preconditions or to check the results were already available, they just had to insert the correct file names
6. In this way they could build up modular test cases. Once the first ones were developed, creating others was much faster as more and more of the building blocks were already available
7. Finally they had to test each test case before going on

Once they got the feeling for it they could work mostly alone so that we could spare our testers time.

Still we had problems, but they originated mainly from the colleagues themselves

(INADEQUATE TEAM) and at the end that also killed this experiment. One of the two had no real aptitude for development, and automation has lots to do with development. The other was good, but incredibly slow so that we could do the automation in house in a fraction of the time he needed. Since they would be paid by time and not by completed test cases it was ultimately a no brainer to kill the experiment.

Still this example shows how valuable support of manual testing can be for test automation

Conclusion

To conclude I return to the use of patterns: We did a lot right, but if we had known about patterns before, we could have been spared quite a lot of pain!

If you are interested in testing our patterns, just write a mail to me (srttgmb@yahoo.com) or Dorothy Graham (info@dorothygraham.co.uk) and we will invite you to our Test Automations Patterns Wiki. In return we would like you to write down your experiences using them whether positive or not. The best contributions will be inserted in the future book (of course acknowledging your merit)

Thank you

References

Dorothy Graham and Mark Fewster (2012) Experiences of Test Automation, Case Studies of Software Test Automation, Addison –Wesley ISBN 978-0-321-75406-6

Hans Buwalda, Dennis Janssen, Iris Pinkster (2002). Integrated Test Design and Automation using the TestFrame Method. Addison-Wesley ISBN 0-201-73725-6

Elfriede Dustin (2002). Effective Software Testing. 50 Specific Ways to Improve Your Testing. Addison Wesley ISBN 0-201-79429-2

Elfriede Dustin, Jeff Rashka, John Paul (1999). Automated Software Testing. Addison-Wesley ISBN 0-201-43287-0

Mark Fewster, Dorothy Graham (1999). Software Test Automation. Addison-Wesley ISBN 0-201-33140-3

Seretta Gamba, Command-Driven Testing, A step beyond Key-Driven Testing. EuroSTAR 2005 TE7

Tim Koomen, Leo van der Alst, Bart Broekman, Michiel Vroon (2006). TMap Next for result-driven testing. UTN Publishers ISBN (10) 90-72194-80-2

Biography

Seretta Gamba started programming for her physics thesis back in 1972 and found this kind of work so satisfying that she never started a scientific career.

Instead she worked as an IT-specialist first in her home country Italy and since more than thirty years in Germany. She has experience in very different branches of IT such as furniture warehousing, semiconductor manufacturing, automotive process controlling and insurance. Presently she works full time as developer, test manager and automation lead at Steria Mummert ISS in Hamburg, Germany.

She has developed Command-Driven Testing and the framework ISSTestStation.
Seretta Gamba has spoken at different international conferences (EuroSTAR; TestKit; Belgian Testing Days; IQNITE) .

Steria Mummert ISS GmbH, Hans-Henny-Jahnn-Weg 29, 22085 Hamburg, Germany
seretta.gamba@steria-mummert-iss.de

Join the EuroSTAR Community...

Access the latest testing news! Our Community strives to provide test professionals with resources that prove beneficial to their day-to-day roles. These resources include catalogues of free on-demand webinars, ebooks, case studies, videos, presentations from past conferences and much more...

Follow us on Twitter @esconfs
Remember to use our hash tag #esconfs when tweeting about EuroSTAR 2013!

Become a fan of EuroSTAR on Facebook

Join our LinkedIn Group

Add us to your circles

Contribute to the Blog

Check out our free Webinar Archive

Download our latest eBook

Download our latest Case Study

www.eurostarconferences.com