Tool Rules

When we speak about Test Automation, our first association is of a major integrated system: An execution controller, scripts, monitors, logs, reports, a user-interface and a database that supports the whole thing. Something big and complex.

In truth, there is nothing in the definition of "automation" that mandates this image. James Bach, for example, defines test automation as "any use of tools to aid testing"[1] .

Here are some examples of tools that I either wrote myself or asked someone to write for me:

- Expansion of data from a concise format to a verbose format
- Aggregation of many csv files into a single csv file
- Computation of the height of a 3D body model
- Extraction of metadata from a video file with a proprietary format
- Scaling of a 3D object model by a given factor

Considering my whole team, I'd guess we create more than a hundred such tools every year. Since I don't think we are that unique, I must assume that this happens in other test teams as well. Given that writing small automation tools is such a common activity, it is worth considering how to best manage this activity.

Problem statement

The thing that troubles me most with automation tools is that there is a lot of waste associated with writing them. The waste is a result of the very little re-use these tools enjoy within the organization. The lack of re-use is due to a number of reasons:

- In a large organization (or even not that large: it's enough that you have a few teams and a few separate labs) there is no simple way for a tester to know if someone in another team developed a tool that directly or with a small modification can provide a functionality she needs
- Even when the tester did find that there is an appropriate tool, it's hard to use. In many cases the tool was written for the sole use of the tool's developer. That person did not have any reason to write usage instructions for the tool. When a new user tries to use the tool, the lack of documentation immediately generates frustration, a feeling of "this is not what I need" and abandonment of the tool.
- If the new user need to make changes to the tool, it turns out that it's not easy to find the source code. When the code is found, it does not compile cleanly for various reasons. Once again frustration ensues with the result of giving up the attempt at reuse.
- It's fun to write a new tool. It's a relatively short task (although always longer than initially planned) with an immediate gratification. It can also be proudly reported in the weekly report: "I developed a tool that does…" .

  The end result is that the same tools are written again and again by different people in different teams. The overall sentiment is that it will be faster to write the needed tool rather than search for it, waste time learning how to use it or fight with code that won't compile.

[1] What is Test Automation? James Bach, http://www.satisfice.com/blog/archives/118

I believe it is possible to improve this situation significantly by adopting a number of rules for tool development. The rules are directly targeting the reasons for low reuse that I listed above.

1) Look around

The first rule is that no one starts writing a tool before investing a tiny effort to search if the tool already exists. First, check Google. There is very little new under the sun and most likely someone already solved your problem. Even if you need to pay something for a license, it's a good deal. Trust me – it will cost you more to develop it. Of course this can succeed only if you have a very efficient system for approving the purchase of low-cost software utilities (say, up to $100). If buying a tool means wasting days in a bureaucratic quagmire, looking for the right budget line and going through a long thread of mail exchanges with a buyer, your testers will prefer to waste these days writing the tool they could (should!) have bought.

2) Create an internal tool list

Looking around is fine, but Google will only find public tools – not tools that were developed internally in your organization. For this, you need to build a list of internal tools.

The list must include, as a minimum, a short blurb describing the tool's functionality, where to download it from and who wrote it. One caveat: my experience in building such a list shows that tool authors won't, generally, put the 10-15 minutes effort to add their tool to the list. They and their team have already ripped the benefit from the tool. The tool authors have no incentive to publish the tool to a wider circle and are concerned that publishing will generate requests for support or training.

For tools your team writes, you can just agree that publishing on the public list is a mandatory step in tool development. Since most teams anyway maintain an internal tool list, you can define the public tool list as your team's tool-list (with proper filtering) instead of maintaining a separate, private list. Developers in other teams can be encouraged to list tools by the usual stuff: small recognitions, a letter of appreciation to their manager, a raffle between tool authors who listed their tool in the last quarter, etc. It should also be allowed for authors to note next to their listing that no support is offered ("use as is and at your own risk").

3) Name it!

While on the subject of tool-listing: you should invest a bit of effort to select a good name for your tool. The name should hint to the tool's functionality, to help finding it when searching the tool list. It needs to be unique or you will soon find yourself with a host of tools with names that are all a not-very-creative variation on "AutoTest". A generic tool name makes communication trickier because you can't just call the tool by its name: "Use RunTool; Not Joe's one – use Rebecca's". The tool name is also a chance for the tool developers to immortalize themselves by choosing a name that has a special meaning to the author; by interleaving the author's initials or name into the tool's name; or by using an internal joke[2]… in short – this is an area for wild creativity which is also adding fun to the work environment.

4) Write a Readme

Each tool must have a short readme file that explains how to use it. Additionally, provide a working example so that whoever downloads the tool can see exactly what the tool does without spending time on assembling the proper inputs. If the tool requires some pre-requisites (e.g. have Visual studio runtime installed on the computer), it must be clearly indicated in the readme file. If it's a small number of dlls – just put these files in the same folder where the tool is. Your goal is that if someone showed an interest in your tool, they can try it with zero effort.

5) Make the code accessible and compile-able

This is so trivial we shouldn't be talking about it: The code of any tool you develop must be stored in a configuration management (aka code repository) tool. I won't insult you by explaining why – I assume you know this already[3].

In the organization-wide list of tools, add the location where the code resides. Alternatively, write this inside the readme file you create for each tool.

Protecting the code and providing access to it is the first step. The next step in encouraging re-use is to ensure that whoever downloads the code can compile it cleanly without losing time. The code must be without hard-wired paths. Dependencies must be based on system variables and a readme file must explain clearly what variables are needed and how to set them. Providing a batch file that will do it automatically is a good idea. If some dlls are needed, add them to the files that are in the code repository. Etc. The ultimate test is to download the code to a computer that was just re-installed from scratch with only the operating system, download the code, do the pre-requisites and compile. It should compile cleanly.

6) What about Having Fun?

If you are serious about gaining control over tool development, you must set a rule that no new tool is to be developed without explicit approval from a manager. The approval

[2] We once had a huge fight with the development team about a test tool we needed. We requested that they support "WebSight" – an internally-developed tool we used successfully in testing previous project. They were adamant that the original tool was lousy, must be redone and provide different functionality than what we said is needed. It took months of discussion to arrive to something that was suspiciously close to our original request. We called it JAWS: "Just Another WebSight".

[3] If you are really new in the business and don't know what I am talking about, search Google for "configuration management" to learn why it is important.

route should be super-fast. All you need to answer are two questions: Did you search Google? Did you search the organization-wide list? If the answer is No to any of these, then the tool development is not approved. If, as a result, the tester found an existing tool – give a positive feedback. If the tester decided to write the tool without approval, it will still be fun, but the other half of the incentive to write a tool, namely pride and the ability to report it as an achievement, become irrelevant: you can't publish that you did something without approval, when everyone knows what the rules are. It won't solve the problem (some people will keep developing skunk-work tools) – but it will improve the odds for re-use.

How to Get Started
To help you in gaining control over your tool development, I created a checklist of rules for internal tool development.

A word of warning: It is not trivial to comply with **all** the requirements in the list. For example, the requirement that tools should be accompanied with a minimal set of tests (Wow! What an unconventional thought... a software tool must be tested!) requires a significant effort. Code review is time consuming and slow. On the other hand, the list contains a number of requirements that are very easy to implement with minimal effort. The list is not "all or nothing". Any item that you choose to do will increase the chances for re-use by a bit. As a team, you can decide which rules are absolutely must and you will adhere to zealously, which are highly recommended and which ones are nice to have. You can remove any item you don't think is a good idea and of course add rules I did not think of (and will be happy to hear about!).

Good luck!