Exploration Principles for Non-Graphical Systems

By Shmuel Gershon

Abstract

It is easy to grasp the idea of test exploration for software that has graphical user interfaces. Exploration and discovery are results of engagement of senses, and our day-to-day software running on modern operating systems and web pages present testers with visual guidance and easily identified inputs. But how does one explore embedded, remote, or web service systems, which reside built-in inside other systems, far away from the testers’ eyes and hands? How does one solve the problem that testing programmatic interfaces demands developing test code, which is hard to do in the same fast cadence of test-learn-change-test used in exploratory testing of GUI systems?

We can solve this riddle through innovation in the testing method and in the testing tools. Fundamental patterns of exploration can be applied to non-GUI systems: identifying the ways to acquire data and to cause change; being in charge of the tests and their interpretation; and modeling the software testing tools in a way that allows quicker interaction.

Index

1 Introduction
   1.1 Exploratory Tests
   1.2 The non-GUI problem
2 Patterns of exploration
   2.1 Exploring a hidden world
   2.2 Exploring another hidden world
3 Exploration Principles for Non-Graphical Systems
   3.1 Identify Probes
   3.2 Nurture Verbs
   3.3 Stay in Control
   3.4 Be Cautious in Judgment
   3.5 Cut out the Middleman
   3.6 Traduttore, Tradittore
   3.7 A condensed summary
4 Putting it into Practice
   4.1 At your workplace
   4.2 Test Automation Layers
   4.3 Utopia in Practice
5 Conclusion
   References

1 Introduction

Software Testing is the act of discovering and exposing quality related information about a software system. Gathering indiscriminate information (any type of data about any part of the software) is too vast and overwhelming and of little use; so it is common for testing activities to focus on finding software bugs and identifying usage problems, as risks to the product’s success provides the most effective data to take action on and is the they seek it. The thoroughness of the definition is often useful when faced with the dilemma of trying to understand the relationship of different project activities with testing.

---

3 A more detailed definition is given by Cem Kaner: “Testing is an empirical, technical investigation of a product, done on behalf of stakeholders, with the intention of revealing quality-related information of the kind that
Exploration Principles for Non-Graphical Systems

To that end, testers tune their work to the areas of highest risk and relevance, and this is done by a combination of following predetermined steps and of defining steps as needed during test execution. Tests combining different mixtures of prescribed steps and on-the-spot steps are often said to fall within a software-testing continuum between two extremities: scripted testing (prescriptive steps) and exploratory testing (improvised steps).

1.1 Exploratory Tests

More than a type of test, this exploration is a mindset, a way we think while we perform a test and create variations from a plan.

To see the effects of visual elements in our ability to improvise tests, let’s do an exercise that is representative of popular Exploratory Testing Exercises.

Please test and provide test ideas for the following software (Figure 1):

- Try the logout/login flows, with multiple application instances, and with/without internet connection
- Fill non-numerical characters in one field
  - After this step an error dialog pops up referring you to the help page. This can prompt you to:
    - Test the contents of the help page
    - Study the modality of the dialog
    - Analyze which type of invalid data triggers a dialog and which does not

You can probably go on and on with more cases to test than these, though the only information provided was a picture. You feel confident in your ability to explore your way around this application even before other explanation or context was provided to help determine the value of individual tests or objectives. Our affinity and experience with graphic interfaces, and the fact that (as the name implies) these interfaces expose their functions and constraints visually, create a fruitful ground for exploration, for deciding our way as we progress, for designing and adjusting tests as we learn from their execution.

Software like this, be it a web page, a desktop application, or a mobile app, figuratively “talks” to us. It creates a dialog.

1.2 The non-GUI problem

However, not all software has a graphical interface. A myriad of programs have no visual exposure, for example coding libraries, most embedded software, web services, and a multitude of console applications.

Testers perceive systems like these as harder to explore, and there are at least two reasons for that:

Our affinity with graphic interfaces create fruitful ground for exploration
1.2.1 Lack of visual stimuli

If we repeat the previous exercise but this time with a different software (Figure 2), you might feel more frustrated about the tests you can come up with. This time the software does not reveal as much information about itself, and valuable additional context (usage, purpose, design, inputs, outputs...) needs to come from sources external to the software presentation. The visual elements are not a condition to be critical of an object’s fit for purpose (we will learn in this article how to prepare to do just that), but they are certainly a catalyst for the critical process and questions.

1.2.2 Programmatic interfaces lead to programmed automation suites

When the product’s inputs and outputs (or the product itself) are a programmatic API, like in the case of a web service, the natural way to reach the product and test it is by implementing programs using this API. We plan one test, and write a program that executes the steps intended for that test. Before we know it, we have a huge list of tests run by a machine, and the testing team spends much of its effort maintaining the automation tests and getting them to run autonomously.

This is a position of disadvantage: The tester does not feel he is in an exploration scene to notice opportunities to adapt his test by taking advantage of hints the software execution shares with him. The tester’s analytical empowerment is not encouraged, and a team in this situation ends up focusing more on the setup and programming skills of its members than on the investigative and critical thinking skills.

As long as humans are in control of a test, every test is in some degree exploratory because testers cannot avoid adding human-variability and are naturally rigged to identify inconsistencies in patterns. A large emphasis on automation endangers this: A limitation of unmanned execution is that it is the epitome of tests placed at the absolute scripted side of the testing spectrum mentioned earlier, and takes no advantage of learnings during test execution nor of the tester’s problem detecting skills.

So how do we break this spell? How can we engage a team in exploration of systems that are not visual, that are embedded or hidden inside other systems?

2 Patterns of exploration

To solve this conundrum, we will look at some examples and try to extract from them ingredients that enable an approach

---

2 The importance of the sight sense is intensified in the software realm, as software abstractions rarely appeal to any of the other senses (smell, balance, taste...).

3 API stands for Application Programming Interface, a set of functions and parameters which enable an application program to communicate with services provided by a separate software.

4 Sometimes a test accounts for more adaptability and sometimes for less, and as long a human is in charge of the test execution, there is variability and judgment in play and the difference is not one of the essence of the test but one of the degree of freedom in it. An unmanned automatic test, on the other hand, lacks this natural adaptability and becomes a different test in its essence for it does not respond to circumstances.
of exploration. The examples are from generic and relatable domains.

2.1 Exploring a hidden world

Once upon a time computer games were more rudimentary and lacked the resources modern games have. World exploration games today show a 3D rendering of the world, a map, location of enemies, and other visual clues that (for reasons explained above) put the player into a natural exploration mode.

Games from the 80's did not have this privilege. We had to explore a world and avoid its perils without ever seeing it once.

Here is a transcript of a play of Zork, one of the most popular text-based adventure games:

C:\> dungeon.exe
You are in an open field west of a big white house with a boarded front door. There is a small mailbox here.
> read mailbox
How can I read a mailbox?
> open mailbox
Opening the mailbox reveals:
   A leaflet.
> read leaflet
   Welcome to Dungeon!
   Dungeon is a game of adventure
   <content snipped for brevity>
> go west
You are in a forest, with trees in all directions around you.
> go east
You are in a dimly lit forest, with large trees all around you. One particularly large tree with some low branches stands here.
> climb tree
You are about ten feet above the ground nestled among some large branches. The nearest branch above you is beyond your reach. On the branch is a small birds nest. The birds nest contains:
   A jewel-incrusted egg.

> take egg
Taken.
> climb down
Forest
> go east
You are in a clearing, with a forest surrounding you on the west and south. There is a pile of leaves in the ground.
> kick leaves
Kicking a pile of leaves isn't very interesting.
> Take leaves
Taken.
A grating appears on the ground!

Feel that? The software was in constant dialog! It educated the player on the correct gameplay commands; it informed him when an endeavor was of low value, and about interesting landmarks of note. It compensated for the lack of visuals with clever devices to empower exploration during our quest:

- Abundance of understood action verbs for commands
- Immediate reactions for the actions performed

Gamers that continue the quest are able to explore the world to an amazing level of detail, reaching impressive understanding of the terrain and its inhabitants. You can search the web for maps created out of players' heads (Figure 3).

![Map of Zork](Figure 3)
Paradoxically, these maps are at the same time faithful representations of the Zork world and completely different from each other. That is because each is a model built as an externalization of a common internal cognitive model.

2.2 Exploring another hidden world

Imagine you are in charge of testing (uncovering information from) a remote system not directly reachable (not unlike an embedded system). The hard-to-reach condition makes tester’s interaction with it expensive. You are told the product will be ready in five years, during which available knowledge will be shared with you; the results of your work will be published worldwide; and you have USD2.5 billion to create an automatic testing apparatus.

How would the system you create look? Perhaps you’d list every single piece of information you want to extract from the product, then produce a large automation suite with tests for each entry on that list so you are ready to fire up all the tests once you get access to the product and collect the result logs.

Now let’s see how NASA tackled the challenge above when given the task to deploy an information gathering (testing) system in Mars. This is an image of the Curiosity Mars Rover that roams the planet (Figure 4).

Rovers on celestial bodies cannot be easily remote-controlled since the radio signal speed is slow (a signal from Mars to Earth takes between 3 and 21 minutes). Rovers are capable of operating autonomously with little assistance from ground control for data acquisition, but require human input for identifying promising targets, and determining how to position itself. Even though each test conducted by an engineer is costly (in time and money), NASA takes into account that they will know more on what is important to learn about Mars as they proceed in their tests execution, and that step decisions are better done by a human engineer than by a machine (no matter how sophisticated). For this, they have a few clever mechanisms:

- Probes in place to gather as much information as possible for the engineers
- The engineer remains in charge of the test procedure
- Engineers are the ones who interpret results and reach conclusions, not the rover

When exploring a planet, or testing software, it is imperative to account for new learnings during tests – for if there wasn’t anything to learn there would be no need for performing a test at all.

3 Exploration Principles for Non-Graphical Systems

There is a lot we can learn about software testing from these examples. We will see six principles distilled from them and from experience testing software. We will relate each one to examples above and give guidelines on how to work with them, including how to identify (feel) that you are in the right track. The principles are:

1. Identify Probes
2. Nurture Verbs
3. Stay in Control
4. Be Cautious in Judgment
5. Cut Out the Middleman
6. Traduttore, Tradittore

Figure 4
3.1 Identify Probes

Probes are the instruments and mechanisms by which we gather information about a system or entity. When we test software we are looking for information, so we will benefit from having sophisticated methods for that. The more probes we have into our software (informing us of status, of state, of function usage...) the better equipped with information we are to identify bugs, expose risks, raise technical or usage-related questions, or even – a recursively exponential benefit – devise more probes that we need to have in place to improve the depth or types of data available.

On planetary rovers, the probes are easy to identify as many are physical probes, prongs, sensors and antennae. The Zork game has data probes in many forms, like the description of the scene, changes on it after every move and the ability to peek at the ‘inventory’ which gives an account of the player’s status and wealth, among others.

A software-testing probe should feel like having eyes or CCTV cameras in every important part of our system under test: being able to know what the software is doing, whether an error or limit is reached, usage statistics...

Probes can take the form of network sniffers, logs, status messages, return values to functions or operations, LED colors or beeps. We can also work with the product manager and programmers to have data and logs tailored for us, specifically for the testing needs.

Despite the product tested being non-visual, when preparing a testing suite many of these can become visual elements, like a dialog popping up when a specific state is reached, a tail command constantly displaying last entries of a log or even audio alerts when a log contains certain word6. Think of it as your own dashboard for the product.

3.2 Nurture Verbs

Verbs are the methods by which we command the product to behave in a specific way.

With tests being an empirical investigation as well as an active performance, much valuable knowledge is gained by executing the software and operating its functions. That is why we are at an advantage when we have more methods to command the software.

Every piece of software has at least one verb: ‘run’, ‘load’, ‘start’, its own “let there be light” – the one that makes it execute; and some software will also have more operations and options, which are exposed to users and are part of the product, and will be used to control the product’s state during tests.

Often software also sports unexposed commands that customers do not see but can be used to trigger a status, configure the system, or help the programming testing team. Cherish those; a large list of secret commands can be a treasure in your testing toolbox and upgrade your team’s skills!

Verbs are perhaps the most distinguishable feature of the Zork game example. It understands a large amount of action commands and offers full control of the player’s position and reaction to the available to us”. Bach also uses a sight metaphor (observation). I named the principle ‘probes’ trying to convey that data sources do not need to be visual in their original form.


6 Secret commands are a treasure in the toolbox and upgrade your team’s skills
environment. For a text-based program, the image it gives is so vivid because it lets us operate the digital world as we would a real physical world (only with monsters). NASA’s Mars rover responds to many remote commands, too. That is what allows their engineers to determine how to reposition, without these commands the engineers’ exploration hands would be tied.

Having enough verbs feels like having your hands inside the software, being able to act every time a “what if” case comes to mind, or finally feeling confident that you can comply when someone outside of your team asks you to do something that no one had thought about with the product.

The verbs in software take various forms: buttons, toolbar commands, command line arguments etc. Less noticed verbs are present in the configuration: Each setting in the configuration of our product acts as a modifier to one or more commands, effectively multiplying the number of verbs we have! Think about software you know that has large setting dialogs (many open-source products are like this) and notice how each configuration transforms what buttons do.

Another source of verbs is the environment your product runs on and the way it is affected by it. As such, if your product interacts with a database, you can make it act by modifying the database contents. If it shares data or memory with other code, you can force action by operating this other software or manipulating the contents of the memory. If it has temperature sensors, control the software by changing room temperature.

When preparing a testing suite, pay attention to the numerous paths existent to command the product under test. Some verbs may need to be added especially for your team’s needs as part of a testability effort.

When considering the verbs available or required for testing an application, it can be useful to classify them for transitivity as we do in grammar: knowing which of them are intransitive verbs and require no subject or object to act on (‘run’, ‘exit’, ‘refresh’, ‘print’ can be examples); and which are transitive and apply to entities that can also be used to control the product (for example ‘save - to file’, ‘connect - to server X’, ‘load - calendar template’).

Consider grouping the verbs together in a visual application even for a non-visual product, creating what can be a control panel for your testing applications.

3.3 Stay in Control

Staying in control is the ability to direct the test as it proceeds. When our attitude in a test is exploratory, we redesign the test and its steps as we go – having the reigns to do so is paramount.

The principle of staying in control suggests that more control on the progress of a test is advantageous and worth seeking.

To draw from our previous examples, Zork showed an obsessive restrain from doing any action on behalf of the player, even when the intentions of the player were known (‘read mailbox’). Letting the player in charge of how and when to perform these actions is part of what makes the world explorable and approachable. The Curiosity rover of NASA Testability: “To test, we must be able to visit the behavior of the product. Ideally we can provide any possible input and invoke any possible state, combination of states, or sequence of states on demand, easily and immediately.”

---

7 Open Source communities have a natural tendency to value flexibility over usability, and sometimes this leads to high-testability software with easy access to verbs.

8 The principle is related to the controllability concept in the Heuristics of Software Engineering

---
is carefully engineered to allow the space scientists to control the rover at a moment’s (or 21 minutes’) notice based on its constant probe information. Not having this ability could mean missing an opportunity to learn or wasting time in a non-valuable test.

How can we identify if we are keeping control of the tests in the test tools we prepare? One way is to notice when we do not feel (or feel less) out of control. If there is no way to act on a whim, if we find ourselves having to wait before changing something important to run the test again... we may not be as much in control of the test progress as we would like to.

A good example for seeing the contrast between a software yielding control to the user and taking it from her we will find in installer applications. Some installers are extensively controllable and let us decide on the smallest details of the installation (what components to install, the directory, which shortcuts to create, and so on). Others are autonomous and decide everything for the user, installing as soon as he presses ‘install’, for the user’s comfort (and often to their chagrin...').

When building your test toolbox, analyze where you would benefit from controlling what the software does and being able to correct course at any point, and where you would prefer quicker runs that are more autonomous and less controllable. Some level of autonomy is healthy, for example to reach common setups or to reset to a known state or default configuration.

3.4 Be Cautious in Judgment

Being cautious in judgment means taking into account the need for a human to deliberate and decide what is a bug, a feature, an issue.

Often we see test suites in which (seemingly) autonomous tests decide when the test is a ‘Pass’ (no bug detected) or a ‘Fail’ (a predefined bug was identified). If the entire testing effort is contained in this automated suite we miss many problems that cannot be predetermined but humans would recognize (human nature is conditioned to detect trouble, inconsistencies or risks; and skilled testers hone this aptitude).

In addition to that, the essence of testing is in the surprises found during test performance, as in a hypothetical world where it was possible to predict all problems and defects in order to automate their tests, we would be better fixing them directly instead of testing for them.

It is valuable therefore to have a testing system that allows the tester to judge and interpret results (based on his knowledge from probes and verbs).

Our explanation of the NASA rover operation makes it clear the engineer is the one in position to analyze results and reach conclusions. That was not a task delegated to the machine.

I like to remark humorously that if NASA were to automate the results interpretation (as we sometimes try to do with tests and bugs), they would have programmed the rover to look for the known form of extraterrestrial life form, entities that are skinny, green-gray, big-headed, and with large eyes. The rover would wander the surface of Mars and find a body that is large, black and white, four-legged and mooing, to automatically dismiss it as “Not an alien, just a cow”.

A method to ensure having a non-judging testing tool is to construct this tool not by “Great Assembly” for generations of judges to come, stressing that more than clear-cut rules for judgment there is place for human deliberation.

9 The principle name comes from a passage in the Ethics of the Fathers book (a compendium of rabbinical teachings from the 2nd century). This appears as the first advice given by the
implementing tests, but implementing the functions needed to test; the tool is then specific in the technology it interacts with, while at the same time generic in the tests we can do with it. For tools that implement specific tests and testing suites with automatic tests, your feeling of how much of the effort goes to analyzing false-positives or adding new tests for bugs that were found elsewhere can give you an indication of the level of judgment you are abdicating or holding.

Many of the testing tools we use are already non-judgmental like this, and it is valuable to assess how your tools fare on the judgmental scale. For example, in the web pages domain, the popular BrowserStack tool allows one to render a page in different browsers, OSs and configurations but at the end displays the page results for the user’s judgment. Other tools, like the W3C Validator are judgmental and provide a Pass/Fail result (with a list of problems attached). Your testing tool can steer a middle course in which it does not take a final judgment, but together with the entire information provide alerts or warnings for suspicious behaviors to aid the tester in making decisions.

3.5 Cut out the Middleman

Cutting the middleman means removing intermediaries between action applied to the product and reaction returned from it, and even more, removing intermediaries between having a test idea and being able to perform that test immediately without losing the train of thought.

In an activity like testing where one is busy looking at results, comparing bugs and relating the system’s behavior to our mental model of the product, the amount of variables kept in mind is large, and cognitive interruptions can break an otherwise great test.

Zork keeps a direct connection between player and game, and that is not because it is a simple game: most games of any generation make an effort to have a direct and quick command cycle because without that the gamer loses his cognitive engagement with the game (much as a tester with a test activity).

The NASA example has more distance (pun intended) between action and reaction, because the answer to every new test or command takes several minutes or hours, if not because of the numerous and complex machinery they are likely to use.

Having no (significant) middleman feels like being able to operate the product in a short time and/or few steps. If a testing tool needs to be modified when you want to do a test (as an example, an automated test that opens 10 connections and needs to be rewritten and compiled if you want to experiment with 15 connections) then it may be hindering the tester from immersing in the testing activity by breaking his line of thinking.

The example mentioning compilation gives us a good simile for compiler software as middlemen when compared to interpreted languages. Programming in a compiled language requires many steps from the moment the code changes until it executes (cleaning, linking, compiling, waiting, building...) while interpreted languages are ready to execute as soon as the code change is saved.

When building a testing tool suite, it is worth to try to remove extraneous steps. Sometimes that may be attained by simplifying the method, but often it will be

---

There are many parallels between the optimal environments for gaming and for work. One study defines the concept of gameplay as “interacting with a game design in the performance of cognitive tasks, with a variety of emotions arising from or associated with different elements of motivation, task performance and completion,” similar to how we could define expectations in the workplace.
achieved by hiding steps behind the scenes in an automatic form (*the healthy level of autonomy mentioned earlier*). One good heuristic to try and maintain a non-mediated mode of work takes into account the first principles: If you are able to collect all (or most) of the verbs and probes into a single tool or panel, it is likely that the end result requires little mediation – especially because had the command required intermediate steps we would have had to hide it when adding the verb to the consolidate panel.

It is valuable to achieve perceived immediacy by efficiently hiding the medium elements away – they are there, but one need not think about them.

### 3.6 Traduttore, Tradittore

A loose translation of this Italian aphorism is “every translator is a traitor”, once one creates a new form to expose an original this new form is not the same as the source despite how high-fidelity it looks. It comes as a caveat that a testing suite with verbs and probes under perceived immediacy and control is now a translation of the original system being tested, a model built for our convenience.

The engagement generated by quick response and full control is such that in our minds the real invisible product is replaced by our representation of it, a model of our own creation. In testing this can be dangerous because we may end up dealing with the model’s limits and abilities instead of the originals – without even noticing.

The effect of translation was clear in our example of the Zork games. A Zork player creates a map in his mind and often on paper that is not the world of Zork (which is a digital abstraction) but a physical representation (*of this abstraction*). The fact that two different players drawing equally accurate maps will produce distinct results corroborates this point.

If we pay attention we will notice this representation effect in much of the software we use. Mind Maps, for example, are referred to as an expression of a person’s understanding. But Mind Maps are limited at least by this person’s ability to scrutinize her mind, by her skill articulating the thoughts, by the medium presented (*flat Mind Map format*) and by the Mind Map app used. Mind Maps will be as useful after this realization as before, but this insight makes them safer to use.

When organizing your testing methods, there is no need to try to avoid all forms of translation – software is an intangible abstraction and any interaction with it requires translation. Every test activity is based on models, and modeling your software under test into the form of another software is a good way to expose

---

11 The Italian cliché was brought to my attention by the memoirs of Gregory Rabassa, who spent his life translating Latino-American authors such as Gabriel Garcia Marques, Jorge Amado and Julio Cortazar. Alluding to the proverb and to his devotion to the translation craft, he titled his memoirs “If This Be Treason”. Surprisingly, notwithstanding his dedication to accurate translation, Rabassa explains that even accurate translations miss the original meanings and intentions: “languages are the products of culture”, “what *dog* connotes for me, is probably different for what *cão* suggests for Antonio Lobo Antunes”.

12 The notion of models as treason appears also in the title of the famous Magritte painting “The Treachery of Images”. The pipe painting is not a pipe (*or we would be able to tobacco it and puff from it*). The concept of treason is apt, because it implies there is blind trust in the representation until we notice its restrictions. This does not render the model useless; if we are aware of its limitation, the pipe depiction can be useful to discuss pipe matters, to learn about pipe parts and practices, or other uses one may find for a pipe painting (*such as arguments on meta-language*).
your mental model to scrutiny and empiric evidence, with the added advantage that the knowledge is at that point executable and easily updatable.

3.7 A condensed summary

These are six principles that, if taken into consideration by a testing team, will enhance the team abilities to explore even hard to reach or see products.

I have used the same six points in a testing strategy document at work, and the best way we found to convey them to the team during our testing tools programming efforts was the motto “it’s all about creating an automated system for manual tests”, repeated as a mantra; in contrast to a natural tendency to create an automatic tests suite.

4 Putting it into Practice

Creating an automated system for manual tests has easy parts as well as harder parts. Some of the difficulties are technical, yet some pertain to human dynamics in the team.

4.1 At your workplace

To some extent, implementing such toolbox means working with what you have now: classifying existing verbs, mapping available probes, organizing these in a control-favorable form and keeping in mind the limitations of this toolbox. This stage is relatively simple because the change may have less noticeable impact in perceived plans, timelines and deliverables. However after this categorization stage, when it comes to the time of enhancing capabilities in each one of the principles, changes may be necessary in the product (for example to improve probes and verbs and testability aspects of it) and in the process (to account for work with non-judgmental tools).

Apart from a change-adverse response to technical modifications, the team and managers may be wary of investing efforts in programming and automating tools without directly covering tests with them.

Yet an exploration environment puts the testing team in good shape to create automatic procedures for specific tests: the team is uniquely positioned to learn about the environment, product and bugs in order to select the most valuable steps to automate and how to do it. As we will see, a suite that learns from these principles is two thirds of the way to having a robust automation suite.

4.2 Test Automation Layers

Starting your suite with generic verbs and probing capabilities provides an interaction layer that can be used for both manual tests and automatic ones.

This division is powerful. An anti-pattern of automated tests is that starting from a list of tests ends up creating a non-layered construct in which test steps and technology functions are intertwined and interdependent, and as tests grow it becomes monolithic and inflexible.

A more efficient way to organize an automation is by separating different needs into different layers: A basic layer that provides generic services (execution, logs, results consolidation), a second layer that deals with the product’s specific technology (and implements verbs, probes etc.), and a third separate layer for test steps which are assembled with the building blocks of the technology layer (Figure 6).
Notice how once you are prepared to explore and test manually, the technology layer is all set and ready for use in automation scripts as well. This method is refreshingly robust: as the product’s interfaces evolve the tech building blocks in the 2nd layer are reprogrammed while the test scripts have a good chance of remaining unchanged; and when you need to change a script’s steps, no change or impact is expected at other scripts or the technology layer.\(^\text{13}\)

4.3 Utopia in Practice

In my career I have been witness to the testing of different kind of products, ranging from visual macros in Excel to invisible embedded software, and I have been part of the programming efforts of these products or their testing tools. All (the ones that succeeded and the ones that failed) stand to me as evidence of the effect of these six principles in the form the tests were conducted: As available verbs, probes and control diminished, exploration was less fluent. Most of them did not add any additional input or outputs (verbs or probes) other than the product’s original functions, and had no special care with yielding control and judgment. Some of them, however, were strikingly welcoming to exploration in spite their embedded unreachable nature.

I am especially proud of two testing tools effort (though I did not participate in ideating or programming them); in which the tools support exploration (and automation) by following the six points we have learned – either intentionally or subconsciously.

One of them is a testing tool for a network-oriented embedded system whose protocol implements a collection of classes and objects closely tied between themselves. The tool displays all the instances of the classes with their internal data, hyperlinks the related objects, and allows modification of any value, refreshing the display to show the effect of any action.

The other is a tool used in the test of a hardware network controller. Few systems I have worked on are as distant and out of reach as this component, which is not only limited by the poor conversation skills of hardware (bits, signals and registers) but operates under a barrier of software and drivers. The tool, as if by magic, has a visual representation of all the relevant registers and signals, reached through a special-access driver built especially for testability. These variables are not only read in real-time but can also be set and written, while all the rest of data updates accordingly.

Both examples try to maximize verbs and probes, do not implement any test (leaving judgment and control to the tester) and provide quick reactions. Both examples make it feel as if they embrace the product complex (as a testing system that controls a system with a large amount of possible states will tend over time to have at least as many states as the controlled program).

\(^\text{13}\) It is a bit naïve to present these layers as being completely disconnected and independent. Reality will end up adding relationships and basic assumptions between them, and the interfaces will be invariable
under test in such a tailored fit, that the tester could confuse the tool model with the real product. Both examples empower testers to explore as much or more as any typical graphical user interface applications would allow.

5 Conclusion

The most effective wrap up of this thesis is that for programmatic interface products, an efficient way to enable and foster exploration is “creating an automated system for manual tests”.

If you desire to enter into this challenge, and wish to enhance the exploration abilities and skills of your team, I recommend using that motto as a reminder during conversations and the six principles as guiding aids during work.

Even if you do not expect to do any change in tools or methods, knowledge and awareness of the concepts presented are by themselves beneficial. You will be better equipped to understand, explain and defend your current practice. If you intend to make or change a work practice, these principles will help you pave the way and communicate expectations.

More than a technical challenge, this will be a personal adventure. Exploration requires bravery from individuals and teams, and is a catalyst for creativity, renewal and trust. Good luck in your journey!

References and more information

I “Exploratory mindset is focused on questions like ‘What if...?’”. Michael Bolton http://bit.ly/ETStChck