Manual means using your hands (and your head)

01 March 2012 · automation, heuristics, tablet, testing

I recently purchased a Samsung Galaxy Tab and an iPad2. Unlike many of my previous gadget purchases, these new gadgets have become very much part of the way I now work and play. One thing I like about them, is their tactile nature. You have a real sense that their is less barrier between you and what you want to do. If you want to do something - you touch it - and it 'just' does it. I don't have to look at a different device, click a couple of keys or move a box on a string (http://en.wikipedia.org/wiki/Mouse_(computing)) to get access to what I can see right in front of me.

Features such as the haptic (http://en.wikipedia.org/wiki/Haptic_technology) feedback provide a greater feeling that you are actually working with a tool, rather than herding unresponsive 'icons' or typing magic incantations into a typing device, originally conceived 300 years ago (http://en.wikipedia.org/wiki/Typewriter#Early_innovations).

The underlying software systems used in these devices is a UNIX variant, just like the computer systems that underpin the majority of real world systems from the internet to a developer's shiny Apple Mac or Linux workstation. UNIX was initially developed 40 years ago (http://en.wikipedia.org/wiki/Unix#1970s) while it has been re-written, ported and improved over the years, as far as these devices are concerned its the stable platform upon which the magic happens.

Part of that magic is variously called the 'interface', GUI or more vaguely the 'experience'. There has been an increasing availability of devices with improved interfaces for many years. The introduction of the command line itself - a little more friendly than punch cards. Graphical menus and keyboard shortcuts. The windowing system, making managing those command line terminals and applications a little easier, especially if you invested in a box on a string (http://en.wikipedia.org/wiki/Mouse_(computing)). Affordable, portable and powerful computers with touch screens and software that uses these features are just another example of this.

From a testing perspective these are exciting new areas to expand our skills and of course challenges to overcome. We get to learn about these tools and toys and how people use them. We also need to grasp how they work - and what they can't do. Yet as I mentioned the underlying technology is conveniently similar. They are still UNIX, they probably speak to other computers using TCP/IP just like your desktop computer does. Much of server-communication under-the-hood probably uses HTTP just like your GMail.

As the 'interface' gets more human oriented. More like the other 'real' tools in our lives they get easier to use. But this of course means they are more removed from what computers

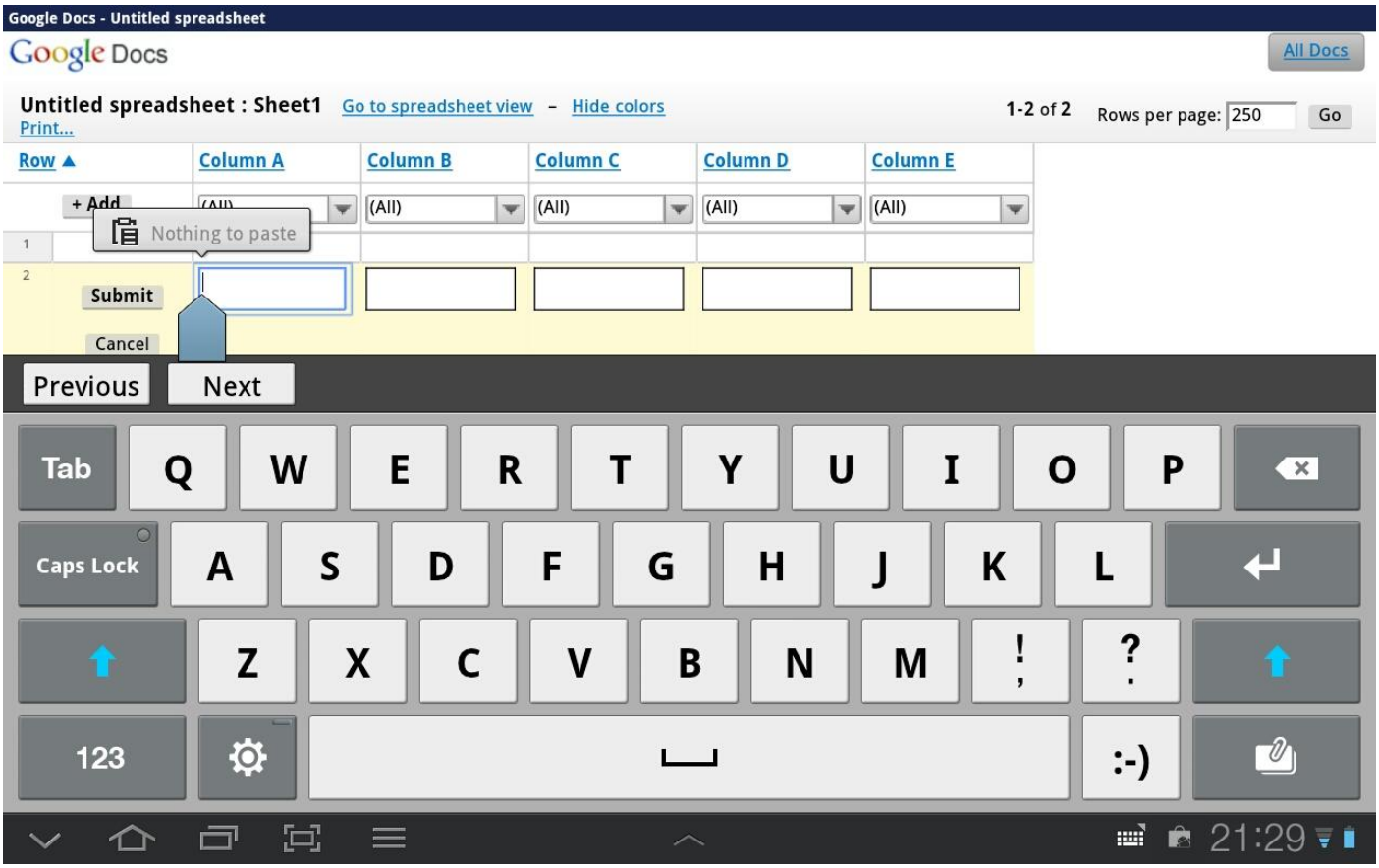
themselves are able to work with. My computer doesn't really have a concept of what 'touch' is: just a way of handling such events. It can't sense a slightly clunky window drag and drop. When we write software to try and test a window drag and drop, we can make it reliably apply the correct events in the right sequence, and check another sequence of events and actions took place. Beyond that we have little knowledge about what is going on as far as our user is concerned.

We often kid our selves that we are testing, for example drag and drop, but in reality we are checking that a sequence of events happened and were received and processed appropriately. Thats fine, and probably a good idea - but its not actually seeing how well drag and drop works. It might not 'work' at all for our user.

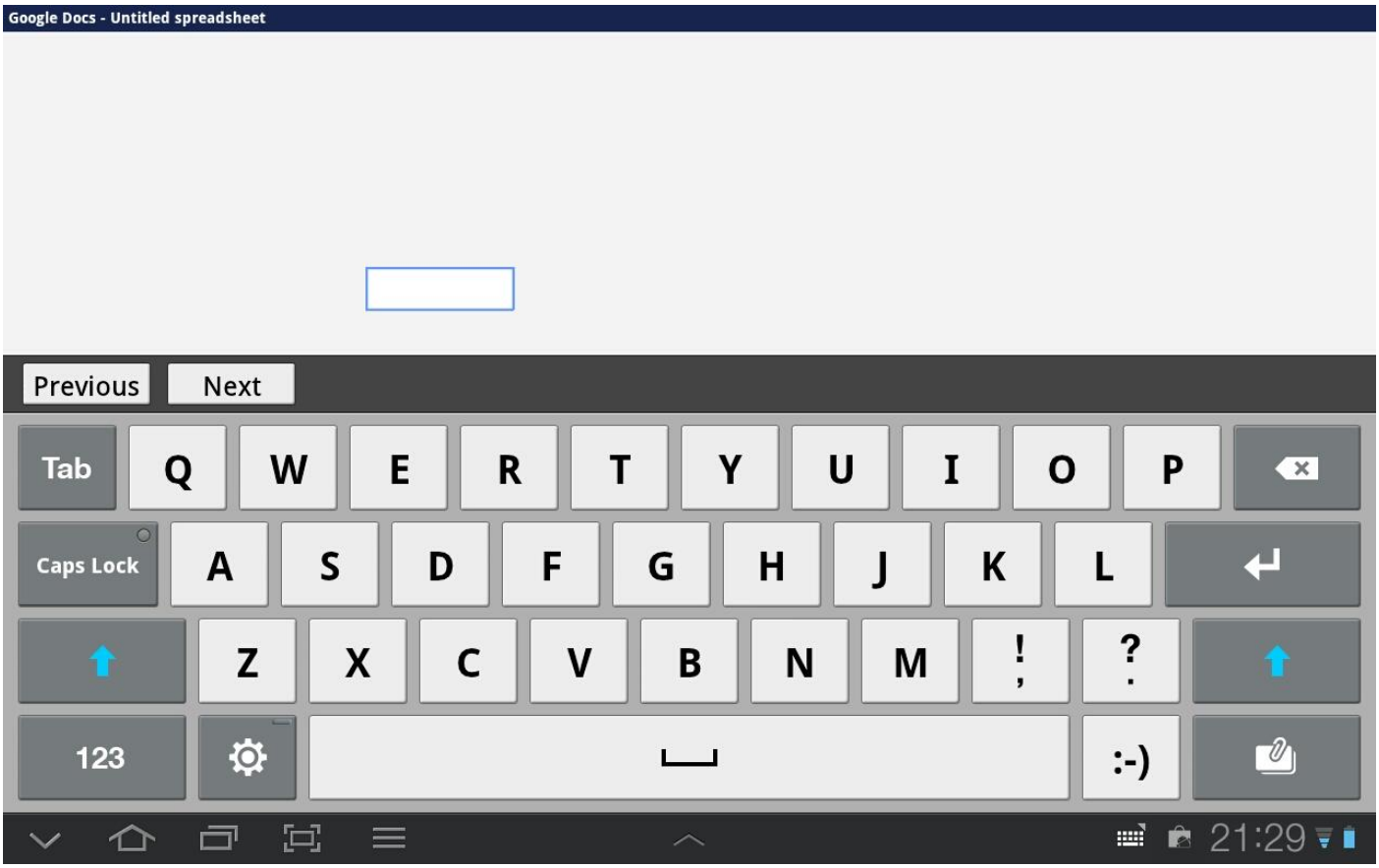
I once worked on a project that included a complex web navigation menu system. Multiple configurations were possible, and depending on the users context various menu configurations and styling would be displayed. A great deal of effort had been spent on test automation to 'test' this menu-ing system. Yet shortly before a release the CTO took a look at the system and noticed the menu was missing, he was not impressed. A recent code 'fix' had inadvertently altered the page layout and the menu was entirely invisible. The test automation was oblivious to this, even if it had checked the menus 'visibility' setting, it would not have detected the problem. The menu was set to 'visible', but unfortunately another component had been placed on-top, obscuring the menu.

This situation is a classic GUI-test-automation mistake. It highlights the common problem with test automation that tries to 'play human'. The test automation couldn't see the page and its missing menu. Why do we keep trying to get our computers to do this? The final arbiter of whether a feature is visible is the user's brain, not Selenium's object model.

I recently took a look at the Google Docs app on my new Galaxy Tab. I created a new blank spreadsheet and just started to click on the cells to enter some numbers. I use spreadsheets on Google Docs frequently and felt testing that I am able to do this on my tablet would be worthwhile. My expectations were high, I'm using the Google Docs App, on [Google] Android software on a market leading Android-supporting hardware. I click on a cell in the spreadsheet, and I feel the haptic response, I know the tablet-knows I've done something. I see the cell become editable.



I wanted to see the range of menu options available, for the cell. As is normal for touch screens and some desktop software I press and hold on the cell. (thats pretty much the equivalent of the right mouse button) Oops. The whole spreadsheet disappears...



I try various similar manoeuvres, all typical tablet interface commands (us tablet kids call them gestures). These tend to give varied results from ‘expected’ behaviour such as displaying the option to Paste - to causing parts of the spreadsheet to disappear. This is a high impact flaw in the application, one that a human tester would find in seconds. (SECONDS I tell you!)

This exposes again a lack of what I mean by manual testing. That is actually using your hands to test things. Literally: your hands. I found that if ‘pressed and held’ for a certain amount of time the spreadsheet would not go blank. But that ‘press-but not too quick and not for too long’ technique was obviously useless for normal usage. This defect is a blocking issue for me, I do not use Google Docs App on my Tablet. I use Polaris, an app that comes free and installed: it has no such issues - and allows me to upload files to the Google Docs server, or email the files etc.

The big ‘A’ Agile crowd and waterfall/v-model die-hards alike fall into a polarised debate about the need for ‘manual [X]OR automated’ testing, but really they are not grokking the need for testing, and testing using the right tools in the right places. Those test tools might be:

- a logging or monitoring program/script running on those underlying UNIX systems.
- a fake Google Docs server that lets you check the client app against a server in ‘known states’.

- a fake Google Docs client app…
- a javascript library that exposes the details of the client application or interacts and triggers events as required.
- a tool that creates random / diverse spreadsheet data - and checks for problems/errors in the server or client etc.
- a tool that can apply load and measure system performance of those HTTP calls.

(Notice the pattern, test automation is good at doing and checking machine/code oriented things in ways that people are not.)

Or even:

- Your hand(s): The haptic feedback doesn't work for XYZ, The tablet can still be hard to use one handed - can we fix that? Why can't I zoom-gesture on this Sunday Times magazine?
- Your eyes, e.g.: observing instantly the HTTP traffic in a monitoring tool…
- Your ears: The screen brightness adjusts to ambient light levels, but the speaker does not adjust to ambient noise levels…

All these tools need to be considered in conjunction. E.g.: Now we know the application is prone to these 'disappearing' tricks - how can we (1) stop it happening? (2) detect when it does? and discuss the merits of doing either or both. Sometimes it makes sense to divide our resources and write test automation and write the fix - for example when you suspect you haven't caught all the causes of the issue. But that inevitable drag away from more testing means you don't find the next bug because your team is still coding the code-fix and test for the code-fix (or the maintenance of both). This is why the decision, on how to proceed at that point, is always context sensitive.

Its not that one should use manual or automated testing, its a question of asking What am I trying to do? What tool do I need? For example: If development has not started, then the best tools might be the tester's brain and a white board rather than a bloated java framework. If you don't know what you need to test - then your hands and eyes will quickly give you valuable feedback as where to go next. For example: the application seems sluggish - we need to check performance and network latency. Or the spreadsheet disappears! - We need to be able to automatically generate those events - and reliably check the visibility.