$(ECHO ECHO) ECHO $(ECHO): COMMAND LINE POETICS

FLORIAN CRAMER

DESIGN

Most arguments in favor of command line versus graphical user interface (GUI) computing are flawed by system administrator Platonism. A command like `cp test.txt /mnt/disk` is, however, not a single bit closer to a hypothetic *truth* of the machine than dragging an icon of the `file.txt` with a mouse pointer to the drive symbol of a mounted disk. Even if it were closer to the *truth*, what would be gained from it?

The command line is, by itself, just as much an user interface abstracted from the operating system kernel as the GUI. While the *desktop* look and feel of the GUI emulates real life objects of an analog office environment, the Unix, BSD, Linux/GNU and Mac OS X command line emulates teletype machines that served as the user terminals to the first Unix computers in the early 1970s. This legacy lives on in the terminology of the *virtual terminal* and the device file `/dev/tty` (for *teletype*) on Unix-compatible operating systems. Both graphical and command line computing are therefore media; mediating layers in the cybernetic feedback loop between humans and machines, and proofs of McLuhan's truism that the contents of a new medium is always an old medium.

Both user interfaces were designed with different objectives: In the case of the TTY command line, minimization of typing effort and paper waste, in the case of the GUI, use of – ideally – self-explanatory analogies. Minimization of typing and paper waste meant to avoid redundancy, keeping command syntax and feedback as terse and efficient as possible. This is why `cp` is not spelled `copy`, `/usr/bin/` not `/Unix Special Resources/Binaries`, why the successful completion of the copy command is answered with just a blank line, and why the command can be repeated just by pressing the arrow up and return keys, or retyping `/mnt/disk` can be avoided by just typing `!$`.

The GUI conversely reinvents the paradigm of universal pictorial sign languages, first envisioned in Renaissance educational utopias from Tommaso Campanella's City of the Sun to Jan Amos Comenius illustrated

school book “Orbis Pictus”. Their design goals were similar: *usability*, self-explanatory operation across different human languages and cultures, if necessary at the expense of complexity or efficiency. In the file copy operation, the action of dragging is, strictly seen, redundant. Signifying nothing more than the transfer from a to b, it accomplishes exactly the same as the space in between the words – or, in technical terms: arguments - `test.txt` and `/mnt/disk`, but requiring a much more complicated tactile operation than pushing the space key. This complication is intended as the operation simulates the familiar operation of dragging a real life object to another place. But still, the analogy is not fully intuitive: in real life, dragging an object doesn't copy it. And with the evolution of GUIs from Xerox Parc via the first Macintosh to more contemporary paradigms of task bars, desktop switchers, browser integration, one can no longer put computer-illiterate people in front of a GUI and tell them to think of it as a real-life desk. Never mind the accuracy of such analogies, GUI usage is as much a constructed and trained cultural technique as is typing commands.

Consequentely, platonic truth categories cannot be avoided altogether. While the command line interface is a simulation, too – namely that of a telegraphic conversation – its alphanumeric expressions translate more smoothly into the computer's numeric operation, and vice versa. Written language can be more easily used to use computers for what they were constructed for, to automate formal tasks: the operation `cp *.txt /mnt/disk` which copies not only one, but all text files from the source directory to a mounted disk can only be replicated in a GUI by manually finding, selecting and copying all text files, or by using a search or scripting function as a bolted-on tool. The extension of the commmand to `for file in *; do cp $file $file.bak; done` cannot be replicated in a GUI unless this function has been hard-coded into it before. On the command line, *usage* seamlessly extends into *programming*.

In a larger perspective, this means that GUI applications typically are direct simulations of an analog tool: word processing emulates typewriters, Photoshop a dark room, DTP software a lay-out table, video editors a video studio etc. The software remains hard-wired to a traditional work flow. The equivalent command line tools – for example: `sed`, `grep`, `awk`, `sort`, `wc` for word processing, ImageMagick for image manipulation, groff, TeX or XML for typesetting, ffmpeg or MLT for video processing – rewire the traditional work process much like `cp *.txt` rewires the concept of copying a document. The designer Michael Murtaugh for example employs command line tools to automatically extract images from a collection of video files in order to generate galleries or composites, a concept that simply exceeds the paradigm of a graphical video editor with its predefined concept of what video editing is.

The implications of this reach much farther than it might first seem. The command line user interface provides functions, not applications; methods, not solutions, or: nothing but a bunch of plug-ins to be promiscuously plugged into each other. The application can be built, and the solution invented, by users themselves. It is not a shrink-wrapped, or – borrowing from Roland Barthes – a *readerly*, but a *writerly* interface. According to Barthes' distinction of realist versus experimental literature, the readerly text presents itself as linear and smoothly composed, "like a cupboard where meanings are shelved, stacked, safeguarded".(1, p.200) Reflecting in contrast the "plurality of entrances, the opening of networks, the infinity of languages",(1, p.5) the writerly text aims to make "make the reader no longer a consumer, but a producer of the text".(1, p.4) In addition to Umberto Eco's characterization of the command line as iconoclastically *protestant* and the GUI as idolatrously *catholic*, the GUI might be called the Tolstoj or Toni Morrison, the command line the Gertrude Stein, Finnegans Wake or L.A.N.G.U.A.G.E poetry of computer user interfaces; alternatively, a Lego paradigm of a self-defined versus the Playmobil paradigm of the ready-made toy.

Ironically enough, the Lego paradigm had been Alan Kay's original design objective for the graphical user interface at Xerox PARC in the 1970s. Based on the programming language Smalltalk, and leveraging object oriented-programming, the GUI should allow users to plug together their own applications from existing modules. In its popular forms on Mac OS, Windows and KDE/Gnome/XFCE, GUIs never delivered on this promise, but reinforced the division of users and developers. Even the fringe exceptions of Kay's own system – living on as the *Squeak* project – and Miller Puckette's graphical multimedia programming environments *MAX* and *Pure Data* show the limitation of GUIs to also work as graphical programming interfaces, since they both continue to require textual programmation on the core syntax level. In programmer's terms, the GUI enforces a separation of UI (user interface) and API (application programming interface), whereas on the command line, the UI is the API. Alan Kay concedes that "it would not be surprising if the visual system were less able in this area [of programmation] than the mechanism that solve noun phrases for natural language. Although it is not fair to say that 'iconic languages can't work' just because no one has been able to design a good one, it is likely that the above explanation is close to truth".(2, p.25)

MUTANT

```
CORE CORE bash bash CORE bash
```

```
There are %d possibilities.  Do you really
wish to see them all? (y or n)

SECONDS
SECONDS

grep hurt mm grep terr mm grep these mm grep eyes grep eyes mm grep hands
mm grep terr mm > zz grep hurt mm >> zz grep nobody mm >> zz grep
important mm >> zz grep terror mm > z grep hurt mm >> zz grep these mm >>
zz grep sexy mm >> zz grep eyes mm >> zz grep terror mm > zz grep hurt mm
>> zz grep these mm >> zz grep sexy mm >> zz grep eyes mm >> zz grep sexy
mm >> zz grep hurt mm >> zz grep eyes mm grep hurt mm grep hands mm grep
terr mm > zz grep these mm >> zz grep nobody mm >> zz prof!

if [ "x`tput kbs`" != "x" ]; then # We can't do this with "dumb" terminal
        stty erase `tput kbs`

DYNAMIC LINKER BUG!!!
```

Codework by Alan Sondheim, posted to the mailing list ärc.hiveön July 21, 2002

In a terminal, commands and data become interchangeable. In `echo date`, `date` is the text, or data, to be output by the ëchocommand. But if the output is sent back to the command line processor (a.k.a. shell) – `echo date - sh` – `date` is executed as a command of it own. That means: Command lines can be constructed that wrangle input data, text, into new commands to be executed. Unlike in GUIs, there is recursion in user interfaces: commands can process themselves. Photoshop, on the other hand, can photoshop its own graphical dialogues, but not actually run those mutations afterwards. As the programmer and system administrator Thomas Scoville puts it in his 1998 paper *The Elements Of Style: UNIX As Literature*, "UNIX system utilities are a sort of Lego construction set for word-smiths. Pipes and filters connect one utility to the next, text flows invisibly between. Working with a shell, awk/lex derivatives, or the utility set is literally a word dance."(3)

In net.art, jodi's *OSS* comes closest to a hypothetic GUI that eats itself through photoshopping its own dialogues. The Unix/Linux/GNU command line environment is just that: A giant word/text processor in which every single function - searching, replacing, counting words, sorting lines - has been outsourced into a small computer program of its own, each represented by

a one word command; words that can process words both as data [E-Mail, text documents, web pages, configuration files, software manuals, program source code, for example] and themselves. And more culture-shockingly for people not used to it: with SSH or Telnet, every command line is *network transparent*, i.e. can be executed locally as well as remotely. `echo date - ssh user@somewhere.org` builds the command on the local machine, runs it on the remote host somewhere.org, but spits the output back onto the local terminal. Not only do commands and data mutate into each other, but commands and data on local machines intermingle with those on remote ones. The fact that the ARPA- and later Internet had been designed for distributed computing becomes tangible on the microscopic level of the space between single words, in a much more radical way than in such monolithic paradigms as *uploading* or *web applications*.

With its hybridization of local and remote code and data, the command line is an electronic poet's, codeworker's and ASCII net.artist's wet dream come true. Among the poetic *constraints* invented by the OULIPO group, the purely syntactical ones can be easily reproduced on the command line. *POE*, a computer program designed in the early 1990s by the Austrian experimental poets Franz Josef Czernin and Ferdinand Schmatz to aide poets in linguistic analysis and construction, ended up being an unintended Unix text tool clone for DOS. In 1997, American underground poet ficus strangulensis called upon for the creation of a *text synthesizer* which the Unix command line factually is. *Netwurker* mez breeze consequently names as a major cultural influences of her net-poetical *mezangelle* work *#unix [shelled + otherwise]*, next to *#LaTeX [+ LaTeX2e]*, *#perl*, *#python* and *#the concept of ARGS [still unrealised in terms of potentiality]*.[1] Conversely, obfuscated C programmers, Perl poets and hackers like jaromil have mutated their program codes into experimental net poetry.

The mutations and recursions on the command line are neither coincidental, nor security leaks, but a feature which system administrators rely on every day. As Richard Stallman, founder of the GNU project and initial developer of the GNU command line programs, puts it, "it is sort of paradoxical that you can successfully define something in terms of itself, that the definition is actually meaningful. [...] The fact that [...] you can define something in terms of itself and have it be well defined, that's a crucial part of computer programming".(4)

When, as Thomas Scoville observes, instruction vocabulary and syntax like that of Unix becomes *second nature*,(3) it also becomes conversational language, and syntax turns into semantics not via any artificial intelligence, but in purely pop cultural ways, much like the mutant typewriters in David

[1] Yet unpublished as of this writing, forthcoming on the site `http://www.cont3xt.net`

Cronenberg's film adaption of *Naked Lunch*. These, literally: buggy, typewriters are perhaps the most powerful icon of the writerly text. While Free Software is by no means hard-wired to terminals – the Unix userland had been non-free software first –, it is nevertheless this writerly quality, and break-down of user/consumer dichotomies, which makes Free/Open Source Software and the command line intimate bedfellows.

[This text is deliberately reusing and mutating passages from my 2003 essay Ëxe.cut[up]able Statements", published in the catalogue of ars electronica 2003.]

`http://www.digitalartistshandbook.org/node/13`

LITERATUR

[1] Roland Barthes. *S/Z: An Essay*. Hill & Wang, 1991. 3

[2] Alan Kay. User Interface: A Personal View. In Brenda Laurel and Joy S. Mountford, editors, *The Art of Human-Computer Interface Design*, pages 191–207. Addison-Wesley, 1990. 3

[3] Thomas Scoville. The Elements Of Style: UNIX As Literature, 1998. [Online; accessed 18-January-2009]. 4, 5

[4] David Bennahum. Interview with Richard Stallman, 1997. [Online; accessed 28-April-2011]. 5