Critique of 'files as directories: some thoughts on accessing structured data within files' (1)

Philip Tchernavskij

Critique of 'Files as Directories: Some Thoughts on Accessing Structured Data within Files' (1)

Philip Tchernavskij
LRI, Univ. Paris-Sud, CNRS
Inria, Université Paris-Saclay
Orsay, France
ptcher@lri.fr

ABSTRACT

In this critique of *Files as Directories* (FAD) by Raphael Wimmer, I argue that FAD as presented applies primarily to traditional programming tasks, consider FAD as a broader subversion of app-like software, and speculate about the hypothetical design space of FAD beyond programming.

CCS CONCEPTS

• **Human-centered computing** → **Interaction paradigms**;

KEYWORDS

Programming, Interaction Paradigms, File Systems, APIs

ACM Reference Format:
Philip Tchernavskij. 2018. Critique of 'Files as Directories: Some Thoughts on Accessing Structured Data within Files' (1). In *Proceedings of 2nd International Conference on the Art, Science, and Engineering of Programming (*Author version *<Programming'18> Companion).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3191697.3214324

1 SUMMARY OF THE PAPER

Files as Directories by Raphael Wimmer[1] presents and discusses the concept of file-as-directory (FAD), i.e. representing files as hierarchically structured data that can be navigated and manipulated with the same tools as folders. The paper sketches how FAD could extend the Unix shell scripting paradigm from its current domains to those covered by graphical applications. It argues that this paradigm shift would provide control and configurability relative to the desktop computing paradigm, and improve on current end user programming paradigms by supporting modular, integratable tools over domain-specific APIs.

2 FAD AS A SUBVERSION OF THE APP MODEL

Wimmer proposes that i) a set of modular, configurable, integratable tools can support tailorable workspaces and workflows, ii) the Unix shell scripting environment provides such a set of tools, and iii) FAD could extend that environment to the kinds of tasks we currently use domain-specific applications for.

This characterization of the Unix scripting paradigm emphasizes how it subverts the model of software exemplified by desktop, mobile, and web applications, in which most data takes the form of files that can only be modified inside the closed environments of apps. Since each app generally encompasses the complete life cycle of the particular type of content it deals with, users must adapt to the workflow of the app, rather than adapt the tools provided by the app to individual use situations. If two apps offer related or similar features, it is generally inelegant or impossible to make them work together, i.e. in the best case, they can be used in sequence by passing a file from one to the other.

Olsen has similarly praised Unix as an explorable and customizable work environment [10]. He focuses on the design choice of unifying Unix commands around the "human-centric" intermediate format of ASCII text, which makes the effects of commands generally comprehensible, and enables economic combination of commands.

Conceptually, software populated by many small tools operating in a shared environment lets people summon the capabilities they need when they are needed. Given mechanisms for organizing and combining tools, e.g., packaging sequences of commands in scripts and wiring them together with pipes, users can tailor their environment to fit their needs. By contrast to the difficulty of making apps work together, it is possible to use scripts created by different authors in combination, or to customize them for one's own needs.

However, the details of how tools are operated, organized, and combined matter to who make effective use of them, and for what. Next, I summarize some inherent limitations of the Unix shell scripting as a replacement for graphical user interfaces (GUIs).

3 ALGORITHMS AND INTERACTION

The proposed practical implications of FAD are that it would become possible to manipulate domain-specific data stored in files in the Unix shell scripting paradigm. Wimmer writes

> "A major benefit of FAD is that it allows end-user programmers to interactively explore the data they work with, use their preferred tools and programming languages, and focus on the algorithmic aspects of a problem instead of learning how to use specific APIs."

It is taken as a given that "focusing on the algorithmic aspects of a problem" is positive. However, algorithms are a specialized method of problem solving with inherent trade-offs.

Figure 1 recreates one of Wimmer's examples of a FAD interaction. In this case, the user modifies the color in an image by combining iteration over a list with a command that applies to single pixels, and applying the composed command to the directory `picture.jpg/pixels/`.

[1] Appearing in this volume

Author version
<Programming'18> Companion, April 9–12, 2018, Nice, France
2018. ACM ISBN 978-1-4503-5513-1/18/04…$15.00
https://doi.org/10.1145/3191697.3214324

```
philip at computer.local in [~]
16:36:23 > ls picture.jpg

cols rows pixels formats metadata

philip at computer.local in [~]
16:36:23 > for px in picture.jpg/pixels/*;\
do subtract.sh 40 "$px"; done
```

Figure 1: Files-as-directories as a Unix scripting extension. The user is manipulating a picture by navigating its internal structure and changing its internal values.

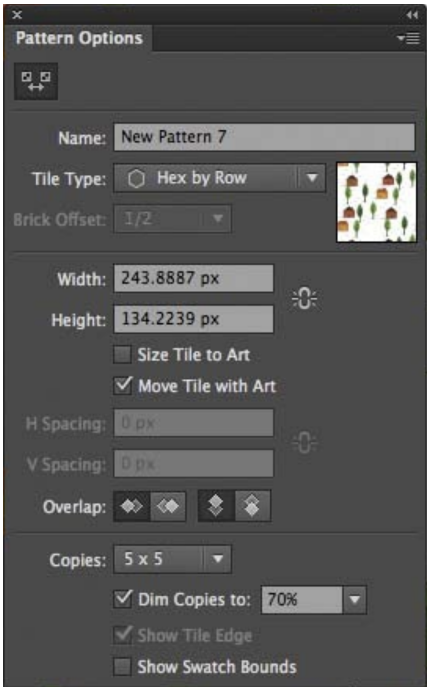

Figure 2: One version of Adobe Illustrator's pattern creation tool.

This example task is chosen to align well with the expressive possibilities of the shell scripting language. Other kinds of graphics editing tasks go against the grain of imperative languages built to deal with lists, strings, and numbers. To a human user, it is simple to highlight and annotate interesting elements in an image by free-hand drawing on top of it. Theoretically, this task consists in changing pixel values in an image, just like the example in figure 1. However, expressing it as a program requires developing routines to identify interesting elements, to determine the shape of the area to highlight, to pick appropriate spots in the image for annotations, etc. By contrast, performing it with a stylus or mouse simply requires looking and drawing, perhaps typing with a keyboard.

Jalal and Tziova have investigated how expert professional designers manipulate visual properties in software [5, p. 63]. In a study where 12 designers recreated two posters in Adobe Illustrator, they analyzed the tools and tactics applied by the designers. The posters were specifically chosen to support powerful programming-adjacent tools, i.e., they contained structurally repeating elements that were excellent use cases for Illustrator's pattern creation tool, which repeats shapes in a pattern, and is operated by configuring a dialog box (figure 2). The overwhelming majority of designers stuck with direct manipulation tools over indirect, cognitively heavy tools such as the pattern tool.

This distinction between programming and direct manipulation tools brings to mind Suchman's distinction between the cognitive assumptions of user interfaces into *planning* and *situated action* [12]. The former assumes that users come to the interface with a goal and a corresponding sequence of actions in mind, while the latter assumes that users continuously re-evaluate short-term actions as their environment changes. Suchman documents in detail the failures of interfaces designed under the assumption of a planning user.

Shell scripting can theoretically support the tactics of situated action, but they require repeated "boilerplate work" in the command-line interface: data is only ever represented as static logs, so one must repeatedly query the working data with `ls`, `cat`, `grep`, etc. to monitor how it changes. Undoing actions may be impossible, so one must work on test data or do non-mutating trial runs of commands[2]. Anecdotally, my personal shell scripting practice is based on these situated tactics, despite the boilerplate work involved.

By contrast, the principles of Direct Manipulation [11] interfaces – continuous representation of the object of interest; operation by physical actions rather than complex syntax; and rapid, incremental, reversible operations with immediate feedback – guide the design of interfaces that do not require these compensatory tactics. Critically, interfaces can be graphical without taking advantage of Direct Manipulation. For example, Illustrator's dialog box-based pattern tool is similar to a powerful shell command that takes several parameters: It provides a general solution to a repetitive task, but it must be fully configured and executed to produce any feedback, and it may be difficult to predict the relationship between parameters and results.

The operational details of tools affect the barriers to effective use found by Ko et al. in investigating end-user programming tools [7]. Particularly the barriers of use (operating a tool correctly), understanding (predicting the results of an action), and information (getting feedback about what changes an action caused). Direct manipulation tools and corresponding representations help avoid these barriers by graphically guiding user operation, having small predictable effects, and continuously producing feedback.

4 FAD AS AN INTERACTION PARADIGM

I do not see FAD as inherently a programming paradigm. Broadly, files-as-directories can be interpreted to mean that the documents that populate an operating system have an internal structure[3] that

[2] Greenberg [4] gives an example of another tactic necessitated by the lacking undo functionality in the shell: typed commands may be unpredictably expanded to their full forms before execution, so one may insert `echo` in front of a complex command to see its post-expansion form before executing it.

[3] Or many possible internal structures, provided by virtual file systems in Wimmer's vision.

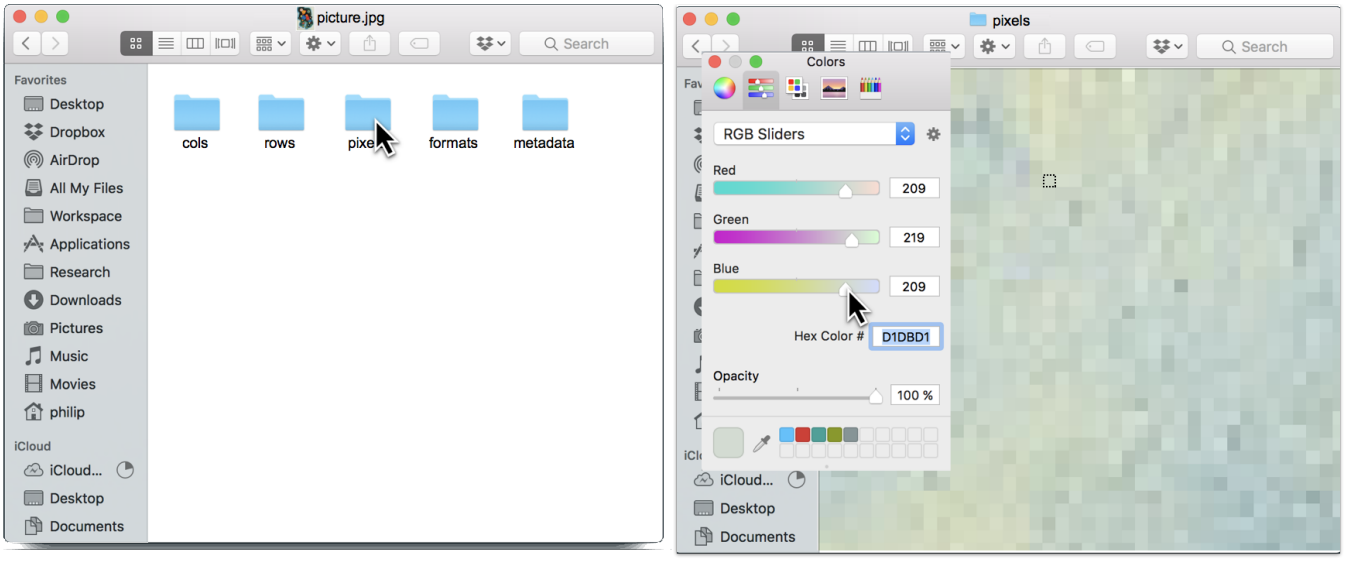

Figure 3: A mockup of files-as-graphical-directories. On the left, a picture's internal representations are shown as sub-folders in an Apple Finder-like interface. On the right, the pixels of the picture are shown as a grid inside the same folder interface, offering both folder-like actions, such as copying and moving, and graphical actions, such as modifying a pixel with the operation system-provided color picking tool.

can be exposed and manipulated through generic operations offered by the operating system.

In figure 3, I have speculated on how FAD might be interpreted as a graphical interaction paradigm. This scenario has the same starting point as the one in figure 1, but takes place in a Finder-like graphical file manager. The `picture.jpg/pixels/` folder inside an image has a graphical representation as a grid of individual pixels. The scenario user then applies the generic color picking and manipulation tool offered by OSX to individually adjust pixels. As shown, this scenario is not materially different from using a graphical editing app to change the image. That is the point: The right representations and tools make a task that would be cumbersome by programming trivial by direct manipulation.

This reinterpretation of FAD echoes Olsen's 1999 proposal that operating systems should be unified around five different human-centered data types; text, 2d images, audio, video, and 3d scenes [10]. Olsen argues that we need "a set of power tools" for these formats akin to those offered by Unix for ASCII text files. As with Unix commands, these tools should have small, self-contained functions, and be trivial to integrate with each other.

No particular set of data representations and tools could feasibly satisfy all users, or even one user all the time. Jalal et al. give color as an example of a data type that is conceptually basic, but is manipulated in diverse ways according to the visual design task it appears in [6]. Rather than one standard color manipulation tool, they argue that the diversity of color manipulation tactics motivates diverse *reifications* [3] – objects with representational and interactive properties – of color specialized for different design tasks, such as generating interesting palettes or leaving traces of activity.

Hence, we do not just need a set of power tools, but an open-ended set of diverse representations and tools. Creating environments in which many tools and representations can be manipulated, organized, and combined is a challenge for software architecture and user interface design. I am part of the five-year research project ONE [2][4], which takes up this challenge. We are developing a conceptual and technical basis for powerful tools and representations in the form of *interaction instruments* and *information substrates* [2]. Interaction instruments mediate physical action by users into specialized actions on domain objects [1]. Information substrates hold information, and apply constraints, transformations, and relationships to it. Crucially, substrates can be layered, so that what appears as a bar chart to the user may be manipulated as a bitmap picture, a set of shapes, or the numeric source data of the chart (figure 4). Wimmer's vision of pluggable virtual file systems that provide directory representions of files can be seen as one kind of – fairly generic – information substrate.

5 ALGORITHMS + INTERACTION

The scenarios in figure 1 and figure 3 are toy illustrations of the strenghts and weaknesses of programming and GUIs: Roughly, we can say that programming-like tools represent and manipulate processes, whereas GUIs are characterized by visual representations of domain objects and direct manipulation of those representations. There is a somewhat populated design spectrum between these two points, as noted by Maudet [9, figure 49][5] in her dissertation on the design of tools for designers. Nearer to programming, there are live and visual programming tools. Nearer to GUIs, there are macro

[4]Further information about this project is available at http://erc.one
[5]Dissertation available at http://www.designing-design-tools.nolwennmaudet.com/

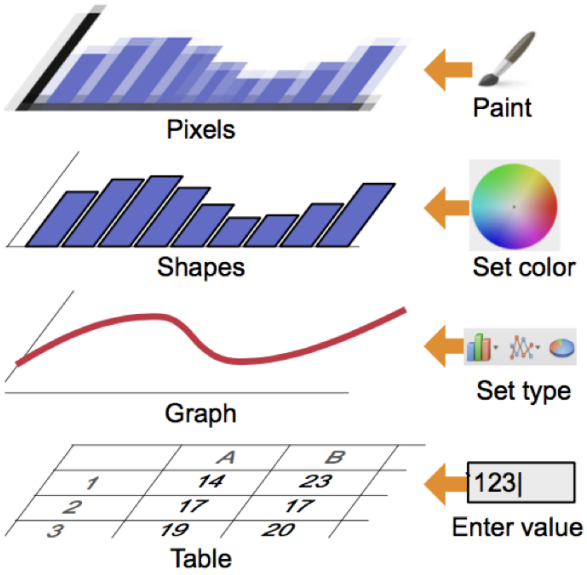

Figure 4: A rich substrate and associated instruments. Figure reused with permission from [2].

systems and interaction techniques such as surrogate objects [8] and macro recording[6], which respectively let users apply direct operations to groups of elements at once or store a sequence of operations as a repeatable command.

Beyond directly user-operated tools, computation enables semi-autonomous tools that can be configured to automate routine or continuous tasks. This brings to mind an aspect of Unix scripting that Wimmer does not consider: daemons. Daemons are background processes that provide some continuous service, e.g., the `cron` job scheduler can be configured to execute scripts according to a schedule. Combining the notion of daemons with FAD leads one to speculate about using background processes to augment arbitrary files with dynamic or interactive behavior. As a trivial example, one could reroute system logs into a bitmap file to create visualizations of system activity.

[6]Such as in Microsoft Excel office.microsoft.com/en-us/excel

6 CONCLUSION

Wimmer's vision of FAD has a strong conceptual starting point in seeking to expose files to diverse, modular tools as public, structured data. To go beyond this starting point, we must interrogate the cognitive and operational details of such tools and data representations, to determine whether they are likely to be helpful, and to whom. I have hinted at a larger interaction design space in which FAD is one point, which encompasses both programming, direct manipulation, and designs in-between. These analyses further motivate the research agenda for software infrastructure that can expose structured data to inspection and manipulation by freely coordinated tools, rather than locking data inside the walled workspaces of apps.

ACKNOWLEDGEMENTS

This work was partially supported by European Research Council (ERC) grant nº 695464 ONE: Unified Principles of Interaction.

REFERENCES

[1] Michel Beaudouin-Lafon. 2000. Instrumental interaction: an interaction model for designing post-WIMP user interfaces. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. ACM, 446–453.
[2] Michel Beaudouin-Lafon. 2017. Towards Unified Principles of Interaction. In *Proceedings of the 12th Biannual Conference on Italian SIGCHI Chapter (CHItaly '17)*. ACM, New York, NY, USA, Article 1, 2 pages. https://doi.org/10.1145/3125571.3125602
[3] Michel Beaudouin-Lafon and Wendy E. Mackay. 2000. Reification, Polymorphism and Reuse: Three Principles for Designing Visual Interfaces. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI '00)*. ACM, New York, NY, USA, 102–109. https://doi.org/10.1145/345513.345267
[4] Michael Greenberg. 2018. Word Expansion Supports POSIX Shell Interactivity. In *Companion to the Second International Conference on the Art, Science and Engineering of Programming (Programming '18)*. ACM, New York, NY, USA.
[5] Ghita Jalal. 2016. *Reification of visual properties for composition tasks.* Ph.D. Dissertation. Université Paris-Saclay.
[6] Ghita Jalal, Nolwenn Maudet, and Wendy E. Mackay. 2015. Color Portraits: From Color Picking to Interacting with Color. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 4207–4216. https://doi.org/10.1145/2702123.2702173
[7] Andrew J Ko, Brad A Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*. IEEE, 199–206.
[8] Bum chul Kwon, Waqas Javed, Niklas Elmqvist, and Ji Soo Yi. 2011. Direct Manipulation Through Surrogate Objects. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)*. ACM, New York, NY, USA, 627–636. https://doi.org/10.1145/1978942.1979033
[9] Nolwenn Maudet. 2017. *Designing Design Tools.* Ph.D. Dissertation. Université Paris-Saclay.
[10] Dan R Olsen Jr. 1999. Interacting in chaos. *Interactions* 6, 5 (1999), 42–54.
[11] B. Shneiderman. 1983. Direct Manipulation: A Step Beyond Programming Languages. *Computer* 16, 8 (Aug. 1983), 57–69. https://doi.org/10.1109/MC.1983.1654471
[12] Lucy A Suchman. 1987. *Plans and situated actions: The problem of human-machine communication.* Cambridge university press.