Programming “systems” deserve a theory too!

Joel Jakubovic
School of Computing
University of Kent
Joel.Jakubovic@kent.ac.uk

Abstract

It is comparatively easy to find new *languages* and derivative work within academic publishing, but somewhat harder for more general programming "systems" or "environments" which encompass more than that. This is a shame, since some of these have a dedicated following and have influenced programming and computing at large. We concentrate on some examples of such that, for various reasons, do not have much material written about them. We suggest some reasons to expect this, and draw attention to some characteristics (both "cognitive" and "technical") marking them worthy of further study. We conclude with a sketch of further steps to make the most of these software artefacts from a research perspective.

1. Introduction

Programming language theory, implementation and evaluation is an expansive and well-established field. Even entire applications, “environments” or “systems”, developed *for the purposes of research*, contribute to progress by being discussed and evaluated within a certain academic scope (for example, HCI). However, there exist innovative, influential or otherwise noteworthy systems that nevertheless fall outside of these realms. As we expand on later, this is in part due to originating outside of the academic environment, and it is also pushed against by the nature of the publishing medium. We will lead in with three such systems to illustrate our later points.

1.1. Smalltalk

Smalltalk could be described as a self-sufficient desktop environment programmed in a language of the same name. It was influential on the rise of object-orientation and graphical user interfaces, and was itself intended to be widely embraced. However, it did not manage to achieve this goal, and lives on in its own niche on a par with other programming language ecosystems.

Whenever a language or IDE feature spreads across the mainstream, Smalltalkers have a reputation for pointing out that Smalltalk was able to do the same thing back in the ’80s. This is a plausible claim, owing to the deliberate high-level, flexible and totalising architecture of the system. Noting its following by a community of devotees, this is evidence of something special and valuable to learn from and build upon. However, despite having a literature presence in virtual-machine optimisation and related *implementation* technologies, insight into Smalltalk’s *design* seems to be mostly scattered around historical magazine articles (*BYTE Magazine Volume 6, Number 8*, 1981), web pages and blog posts.

Without a well-structured design discussion, it is only clear that there is “something about Smalltalk” that is worth improving on. In other words, it is difficult to distinguish which specific aspects of Smalltalk are essential to its value and which are incidental. We can identify characteristics such as *meta-circularity* and *self-sufficiency*, by which the software within lives in a world where “everything is Smalltalk”. Another identifying feature is that instead of transient “applications” and manual saving to “files”, the entire system state is persisted as a continuously evolving “image”. Smalltalk also embeds its programming language within a larger prescribed context of graphical interface and device utilities, encompassing the same roles (and scope) of an entire Operating System. Such prescription of a graphical interface is characteristic of what we are referring to as “systems”, but this last “encompass the world” aspect is more unusual.

1.2. HyperCard

HyperCard was a platform for exploring and creating interlinked sets of multimedia pages, often regarded as a precursor to the Web. Explicitly designed with the goal of end-user empowerment, with a slogan of "programming for the rest of us", HyperCard was a popular platform both to share and to author among teachers, businesspeople, and other non-programmers. This was no doubt in part due to its (initial) default inclusion in Macs. However, the existence of other default apps (such as Terminal) *not* enthusiastically adopted by end-users means that HyperCard itself still stands on its own merit.

HyperCard applications were organised into "stacks", with typical UI elements (text, pictures, buttons) added and edited by direct manipulation. Every such element could have a script attached to it in HyperTalk, a language designed to resemble English. HyperCard also distinguished itself in having separate "beginner" / "advanced" layers, allowing novices to make their way through straightforwardly *using* and maybe editing stacks superficially, before digging into more advanced authoring and scripting.

Unfortunately, despite its popularity and originality, HyperCard fell out of use due to corporate and marketing decisions and subsequent declining technical support, eventually falling out of compatibility with newer versions of the Mac OS. As HyperCard was primarily a *commercial* product, not much has been written about its design, but its popularity suggests that it is worth studying and possibly adapting for future software systems. Sample design characteristics for HyperCard include its layered design allowing progression from *using* to *developing* card stacks, and the fact that such developing is not considered something *far away* but instead occurs in the same user interface.

1.3. Flash

Adobe Flash was a widely popular browser plugin providing "rich" content (video, audio, and games) for the primitive early Web. Such applications were created using Flash Builder: an advanced graphical workspace for various forms of vector graphics and animation, scripted with an event-driven language called ActionScript.

Flash gave rise to an explosion of animations, games and websites across the early Web, often by individuals and within online communities. Because it was used so widely for websites and games, Flash can be considered a success case, to some extent, of "end-user" software development. Unfortunately, Flash was thought to be too insecure and poorly-optimised for the emerging mobile platforms of the 2010s. Apple's decision not to support Flash on the iPhone ultimately spelled its decline and planned official demise at the end of 2020.

Following from our previous two examples, what are the interesting design characteristics of Flash? Its success may be attributed to the fact that it merely *filled a gap* and achieved rapid ubiquity as a result, even defining a new *medium* of interactive Web content. This is certainly true, but suggests this role could have been filled by any other provider of rich content support. Flash Builder, though, does seem to be an effectively designed tool for its audience. It is a graphical editor centering around a WYSIWYG "stage" metaphor, surrounded by many specialised sub-interfaces (for example, managing the broad and narrow details of keyframe animation), and with an attached scripting sub-editor. Are there more specific design principles hiding here than merely "use the right interface for the job"?

2. What to do?

2.1. Rational Reconstruction

The main point of focusing on such systems is that they exist outside of the academic literature, having been commercial or educational ventures rather than research artefacts. This means that theoretical analysis or design rationale, if it exists at all, may have never been published or be scattered around documents for internal use. In some cases, there may not be any such written material or what existed has been lost.

This means that it is currently hard to do rigorous follow-up work on these artefacts. If we wish to build on their successes and learn from their failures, in a scholarly way, there needs to be more than the artefact itself and documentation. They need to be made "legible" to academic literature by *reconstructing* what their design and theoretical analysis *would* have been in such a counterfactual world.

2.2. Cognitive and Technical Dimensions

One component of such a “rational reconstruction” could be an analysis in terms of a common vocabulary of named characteristics. For example, the Cognitive Dimensions of Notation framework (Green & Petre, 1996) suggests a number of “dimensions” along which a user interface or “notation” may be measured, along with linkages and tradeoffs between different dimensions. This is certainly useful, but its emphasis on the *cognitive* aspect of *interfaces* might leave uncharted other aspects – what we suggest to be *technical* dimensions – of the “rest” of a software system.

For example, a programming language in which “code” is expressed in the same form as “data” is said to exhibit “homo-iconicity”. We could extend this to systems-in-general, perhaps, as a measure of how much *automating* a task resembles manually performing it. Similarly, “self-sufficiency” measures the extent to which a system can be improved and evolved from within, without relying on external tools. Smalltalk was designed to be quite high on this measure (intended to “obsolete itself”), while HyperCard and Flash (along with most other software) facilitate creation of *separate* artefacts (HyperCard stacks, Flash applications) and can only be changed via in-application preferences or by editing their source code.

Our desire for “technical” dimensions comes from a feeling that there are also properties of “the system itself” rather than merely its *interface*. If *cognitive* dimensions are about how a system is *received* or *experienced* by users, then *technical* dimensions are about how the system was *intended* or *designed*. However, this does rest on an assumption that separating the “interface” from the “system itself” is a meaningful or useful thing in the first place, and we invite further discussion of this.

2.3. The Medium is the Message

Another important thing to note is that academic publishing is by and large a paper (virtual or physical) medium; as such, the ease with which one can publish varies according to the medium of the subject matter. Programming languages, being strings of characters equipped with syntax and semantics, lend themselves very well to describing in a paper. However, our systems of interest is that tend to be graphical, interactive *systems* rather than *languages* alone. They define much more of a given computing environment than languages; they additionally state how they are graphically presented and manipulated. So, in order to properly present or discuss these systems in full, one needs to at least provide pictures and ideally provide a running version.

While videos or runnable artefacts can be part of academic submissions, they are still seen as “supplementary material” to accompany a written paper. It is plausible that this makes it less attractive to present general “systems” in academic publishing, and the non-academic nature of our examples is easier to understand in light of this. This is recognised in (Edwards, Kell, Petricek, & Church, 2019), which begins a conversation about how format, submission and peer-review ought to look for such interactive systems and work on their *design*. Such a model would support a “gallery of interactions” with (possibly simplified) re-creations of systems as one possible corrective.

3. Conclusions

In summary, there exist interesting holistic “systems” which seem to have been ignored in published research. To be able to do follow-up work on them, we need a deeper and more rigorous understanding. But their typical commercial or educational nature means there was no initial body of literature to seed such a process. Additionally, the (virtual) paper medium is not well suited to “systems” in the first place. We propose to re-create the design analysis in light of innovations like the Cognitive Dimensions framework, and taking advantage of more appropriate media like video or interactive essays. A Smalltalker might say that “everything was already invented in the 1970s”; we agree that there was little explicit building upon the visionary work of the past. To remedy this, we need to take a fresh, academically rigorous look at such old work on programming systems.

4. References

Byte magazine volume 6, number 8. (1981). Retrieved from `https://archive.org/details/byte-magazine-1981-08/`

Edwards, J., Kell, S., Petricek, T., & Church, L. (2019). Evaluating programming systems design. In *Proceedings of 30th annual workshop of psychology of programming interest group.*

Green, T. R. G., & Petre, M. (1996). Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *JOURNAL OF VISUAL LANGUAGES AND COMPUTING*, *7*, 131–174.