Unix, Plan 9 and the Lurking Smalltalk

Stephen Kell

Abstract High-level programming languages and their virtual machines have long aspired to erase operating systems from view. Starting from Dan Ingalls' Smalltalk-inspired position that "an operating system is a collection of things that don't fit inside a language; there shouldn't be one", I contrast the ambitions and trajectories of Smalltalk with those of Unix and its descendents, exploring why Ingalls's vision appears not (yet) to have materialised. Firstly, I trace the trajectory of Unix's "file" abstraction into Plan 9 and beyond, noting how its logical extrapolation suggests a surprisingly Smalltalk-like end-point. Secondly, I note how various reflection and debugging features of Smalltalk have many analogues in the fragmented world of Unix programming. Finally, I consider these how these two directions of change may yet be convergent within future Unix-derived systems, and what this might mean for programming languages.

Key words: Unix, Smalltalk, Plan 9, metasystem, composition, binding, integration, virtual machines, reflection, debugging

1 Introduction

For much of their history, high-level language virtual machines (VMs) have aspired to erase operating systems from view. Writing in the August 1981 "Smalltalk issue" of Byte Magazine (Ingalls, 1981), Dan Ingalls set forth various design principles behind the Smalltalk language and runtime (Goldberg and Robson, 1983), and addressed the issue of integration with the operating system as follows.

> An operating system is a collection of things that don't fit into a language. There shouldn't be one.

Stephen Kell
Computer Laboratory, University of Cambridge, 15 JJ Thomson Ave, Cambridge, United Kingdom, e-mail: stephen.kell@cl.cam.ac.uk

This article examines why this change has not come to pass, and what this might teach us about the respective roles of languages and operating systems, and indeed what distinctions can meaningfully be drawn between the two. Key witnesses will be the abstractions of "file" (in Unix-like operating systems) and "object" (in Smalltalk-like languages), which I'll argue have been on a convergent trajectory: for decades, the two fields have been tending towards the same end point from more-or-less opposite ends of a continuum. Meanwhile, we find a similar pattern in treatment of debugging, or more generally *reflection*, where again convergence is occurring, but this time with Unix siding more consistently with plurality and fluid abstraction boundaries—themes we now associate with a more *postmodern* approach to design, although likely arising in Unix as much from practical necessity as from conscious philosophy. This leads us to the conjecture that future advances could result from a post-hoc revisitation of Smalltalk's design goals, with a view to evolving Unix rather than replacing it.

2 Two origins

Unix and Smalltalk both have their origins in the late 1960s, and emerged in recognisable form in the early 1970s. According to Ritchie and Thompson (1974), the initial PDP-11 Unix became operational in February 1971, although a precursor on the much smaller PDP-7 had been written in 1969 (Ritchie, 1993). The earliest language bearing Smalltalk's name was implemented in 1971, although it was the next year's iteration (Goldberg and Kay, 1976) that provided the familiar windowed, bytecoded system—the first "real Smalltalk" in the words of its principal designer Alan Kay (1993).

2.1 Motivations for compositionality

Despite the similar timing, the motivations for the two systems could hardly have been more different. Smalltalk's lay in an outward-looking, futuristic vision of personal computing as an expansion of human thought, society and education—its inspirations including Engelbart's NLS, Sutherland's Sketchpad and Papert's LOGO, with an obvious emphasis on pedagogy and accessibility to children. It was concerned with lowering human thoughts down gracefully to a level executable by machine. Unix's aspirations were more prosaic—to create a powerful and efficient time-sharing system suitable for minicomputers of the day, borrowing those good ideas which could be salvaged from Multics without falling victim to the overambition, slow development and machine-level inefficiency that had led to Bell Labs' withdrawal from the latter project (Ritchie, 1984). The services required of the operating system were directed largely by the hardware and the immediate problem of efficiently time-sharing it—although details of the design were also guided by its creators' experience as programmers. That narrow conception of the system's would-be programmers contrasts markedly with Kay's concern for 'ordinary' people, children and the world at large.

Both systems were, however, united in seeking a coherent system of powerful abstractions for interactive use by programmers and end users, and in fact drawing very limited distinction between these two categories of human. Both had also latched onto the idea of *compositionality*: that the path to a flexible and powerful system lay in a few primitives which could beget a range of simple constituent parts amenable to combination, thereby servicing a very large space of user needs using relatively little new code. Later, Kernighan and Pike (1984) would summarise Unix philosophy as "the idea that the power of a system comes more from the relationships among programs than from the programs themselves", while Kay credited as Smalltalk's essence the idea that "everything we can describe can be represented by the recursive composition of a single kind of behavioral building block". Both Unix and Smalltalk can be seen as "grand designs" in a modernist tradition—seeking to contain and circumscribe particular visions of computation and the services supporting it. Keeping the system simple and comprehensible, such that it could be understood and "taken apart" by its programmers, was a shared goal—although that is overlooking a the huge difference in aspiration for who those programmers might include.

Indeed, the two systems' designers clearly differed in the relative grandness of their ambitions. Kay summarised his intention with Smalltalk's object abstraction as being to "take the hardest and most profound thing you need to do, make it great, and then build every easier thing out of it". This is aiming high and leaving little room at the top, instead seeking to *contain* from above. By contrast, Ritchie repeatedly emphasises the more restrained goal of building a *self-supporting* environment—that is, sufficient for running the tools necessary to develop the system itself, including ancillary tasks such as text-formatting of the system's documentation, but with limited concern for what might characterise application programming in general. That the system's core time-sharing and programming mechanisms could provide a usable foundation for more general applications was an aspiration, made reasonable by an evolutionary mindset and absence of truly fixed decisions, but not a critical initial part of the exercise.

2.2 Evolution and survival

These differences in reach brought consequences for how each technology could spread and evolve in the hands of its users—particularly as the explosive growth of computing saw the pool of such users and (potential) contributors grow from the dozens into the millions. Differing expectations about user-led change are embodied (but largely not stated) in the two designs. Notwithstanding its later commercial productisation and the compromises that entailed, Unix's essential attitude is characterised by an expectation of continuous change at all levels—enabled by self-conscious internal simplicity, and a tacit acceptance of imperfection. It embraces porous boundaries: it admits arbitrary and even divergent modifications, and makes little effort to hide system internals. In a traditional (pre-commercialisation) Unix system even the user/kernel boundary is not well guarded, since the system also includes the source code, compiler and other tools necessary to build a new kernel—even if, owing to the shared nature of the machines, not all users would have the privilege to deploy one. Meanwhile, its user-space environment consists of a deliberately simple set of abstractions, provided in a spirit of *laissez-faire*. This enshrines a *pluralist* attitude to many aspects of the system (as we will explore in due course), since it is a given that there is no optimal "right way". Smalltalk's more high-minded design is more strongly focused on a careful set of design elements put in place from above, hiding considerable internal complexity (particularly in the optimised implementations that emerged later). The boundary between a Smalltalk system's core implementation and its user is therefore harder than the equivalent in Unix: although a large fraction of the system resides in the bytecode "image" and may be inspected and modified from within, the core mechanisms that bootstrap this experience reside within the virtual machine; they lie behind the curtain of the implementation and are not exported to the user, at least not uniformly.

It is worth noting that commercial realities complicate the picture in both cases. On the one hand, it is significant that the Smalltalk-80 distribution consisted of an (executable, evolvable) image coupled with a (non-executable) *specification* for the virtual machine that would host it. Nevertheless, this did not reflect the PARC-internal experience of developing and using Smalltalk, where, just as with pre-commercial Unix, development of the VM 'kernel' and user-land 'image' were intertwined. Equally, commercial Unix variants would typically not include their own source code nor expect users to rebuild kernels, even though Unix's Bell Labs creators did so routinely. One can also argue that Smalltalk's efforts to push as much code as possible code out of the magic firmware and "into the image" expressed an ambition very similar to Unix's porous boundaries, albeit adhering to a higher-level but more opinionated set of abstractions. Of course, a key difference was that Unix programming remained a task done "from the outside", by editing and compiling and re-starting programs, in sharp contrast with the radical innovation of Smalltalk's image-based programming experience, As late as 1984, Rob Pike documented a visit to PARC noting that "the feel of the system is remarkable... more like sculpting than authoring" (Pike, 1984) (in a note dominated by grumbles about intrusive interactivity and the poor support for files and networks).

In these subtly different approaches to similar goals, we see a parallel with the distinction termed by Gabriel (1994) as "worse is better" (Unix) versus "the right thing" (here Smalltalk, playing the Lisp role). This is visible not only in the designs themselves, but also in what happens as the technologies evolve in the field—what Gabriel called "survival characteristics". This process of in-the-field spread and change, particularly against the backdrop of the mass marketisation of computing, the loss of authorial control over the system, and the replacement of single coherent narratives with multiple divergent views, shares much with the heterogeneity and loss of "big picture" identified by Noble and Biddle (2002) as a transition from modernist to postmodernist conceptions of programming. Meanwhile, Smalltalk's virtual machine paradigm, in which fundamental design elements—objects, messaging—were fixed up-front, represented a *quid pro quo*. In return for these impositions, strong and desirable properties would arise naturally within the population of user code. Code would be modifiable from within, remain inspectable and debuggable at all times, and retain a certain uniformity and (presumed) cognitive advantages from its adherence to the carefully constructed object-messaging abstraction. This idea of "control after decentralisation", observed as the property of "protocol" by Galloway (2004) in the context of networked systems, is shared by all high-level languages: the language's abstractions amount to a decentralised management discipline, and are in some sense inherently political or at least controversial. Unlike global networks, language-based "distributed management" systems are easy to opt out of, by choosing a different language; there is little imperative for a unique language to "win".

Although the minicomputers of the 1970s were far removed from the nascent personal computers such as the Xerox Alto, the inevitability of the personal microcomputer meant that these worlds were on a collision course. Soon after Alan Kay ceded the front-line design leadership of Smalltalk to Dan Ingalls in 1976, the system was rebased from the Alto onto commodity hardware, after Xerox (inexplicably in hindsight) opted not to take the Alto to market. This hardware included, notably, the Intel 8086, but also machines running Unix (with the emergence of Sun workstations around 1982) and, albeit indirectly, the "other" personal computing platform, the Apple Macintosh.[1] The convergence occurred in the other direction too: from the early 1980s, Unix was being run on commodity microcomputers (Hinnant, 1984), whose Intel-based successors would thoroughly take over server-side computing over the course of roughly a decade, starting in the mid-to-late 1990s. Plan 9, AT&T's putative successor to Unix, was designed well after this convergence became evident, affirming that the two distinct origins do not fully explain the persisting differences between Unix- and Smalltalk-like systems.

Unix is, infamously, a survivor—even satirised as "the world's first computer virus" (Garfinkel, Weise, and Strassmann, 1994). Its design remains ubiquitous: not only in its direct-descendent commodity operating systems (e.g. GNU/Linux), but as a key component of others (Apple's Mac OS) and a clear influence on the remainder. Smalltalk, by contrast, is easier to miss in modern systems. As a language, today it finds only niche interest. Its key programmatic concepts, namely classes and late-bound "messaging", have had an enormous influence on popular languages; this is clearest in highly dynamic class-based languages such as Python and Ruby, but is easily discernible in Java and C++, among many others. The rich user interface it presented to the programmer has also influenced countless modern "integrated" development environments. Despite this considerable influence, something seems to have been lost: anecdotally, enthusiasts are quick to point out that none of these

[1] This occurred not only from the abortive beta release of Apple Smalltalk in 1985, but from the 1979 demonstration of Smalltalk on Xerox's Dorado machine which would inspire the software for Apple's Lisa and later products.

contemporary languages or environments matches the simplicity, uniformity or immediacy of a Smalltalk system.

2.3 Languages and systems

When I write of "Unix" or "Smalltalk" in what follows, it will be important to distinguish their early, idealised conceptions from their later evolved forms—particularly in the case of Unix which, as we will see, has evolved considerably. The culmination of idealised early "Unix" was probably 1982's 4.2BSD, which retained the time-sharing flavour of the original but, by adding the mmap() call, tied a long-anticipated knot between files and memory. Smalltalk's evolution has been more measured, particularly after 1980; perhaps the most significant change has been the emergence of Smalltalks hosted *within* a wider operating system, as typified by Squeak (Ingalls, Kaehler, Maloney, Wallace, and Kay, 1997). By contrast, our notion of "Smalltalk", although flexible regarding finer details (such as metaclasses, added in the late 1970s), must be taken to mean a system occupying the entire hardware. This recalls the era of both systems' origins, where programming systems and operating systems were not as strongly delineated as at present. Kernighan and Pike (1984) described Unix as a "programming environment", whose primary languages were (implicitly) C and the shell. Perhaps the key distinction is that the Smalltalk system gives primacy to the Smalltalk language and its attendant concepts, to an extent not seen in the Unix system and its respective languages. High-level concepts, such as objects and messaging, inevitably draw competing proposals—dissenting voices to the cognitive (or political) theories proposed by the language. By contrast, although C is given special status in Unix, its status amounts primarily to engineering investment rather than a conceptual primacy; the fixed concepts (memory, instructions, and I/O interactions) instead come largely from the machine. In what follows we will explore the consequences of this "language-forward" approach of Smalltalk in contrast to the relative reticence of Unix.

3 How not to fit in

Let us return to Ingalls's statement of vision.

> An operating system is a collection of things that don't fit into a language. There shouldn't be one.

By 1981, the collision of Smalltalk-style personal computing and Unix-style time-sharing systems was well under way, and Ingalls's statement serves as a manifesto for how to resolve it. Elaborating on his "things that don't fit" characterisation, he notes that to invoke the operating system from a high-level language is "to depart from an otherwise consistent framework of description... [for] an entirely different and usually very primitive environment". Although not stated explicitly, we can infer that Ingalls' vision for there "not being" an operating system would include gradually pulling more and more system functionality (e.g. filesystems, network stacks, and perhaps isolated processes) into the Smalltalk runtime, where it could be exposed in the form of a higher-level message-oriented façade (e.g. as persistent and remote objects). This contrasts with the byte-streams and raw memory interfaces of operating systems in general, and Unix in particular—since Unix seems unquestionably one system to which his "very primitive" referred.

Over thirty-five years later, Smalltalk's influence has been felt strongly in certain ways, thanks largely to its mainstream successor, Java. However, its influence has been in relatively fine details: the popularisation of garbage collected runtimes and of class-based programming.[2] By contrast, the various "in-the-large" design points of Smalltalk to which Ingalls drew attention in the Byte Magazine article—its unified late-bound message-oriented worldview, with aspirations of supplanting the operating system—have failed to become mainstream (at least so far).

Was there a real benefit in whole-system design underlying Ingalls' position? Could such a design be realised in a contemporary context, and if it were, would there remain any need for a programmer- or user-facing operating system? I will make a case for answering these questions in the affirmative. Let us start by identifying the potential benefits of Ingalls' vision, and contrasting these with parallel developments in Unix relating broadly to the concerns of composition.

4 The Smalltalk wishlist

In saying that there "shouldn't be" an operating system, what benefits is Ingalls seeking? Clearly, the problem being addressed is that of *complexity* in software (the same article emphasises "management of complexity"), and that Smalltalk's approach is to provide well-designed abstractions which are *compositional* (which I take to be the essence of any programming language). Although the article does not list the intended benefits explicitly, we can infer that the following general benefits are probably included.

Programmatic availability

The Smalltalk programming abstraction is also available to system-level tasks. Programmers can write code "in the same way" against both user-defined and system-defined abstractions (e.g. processes, devices), also allowing the application of existing Smalltalk code (say, the famous collections library) to these new target domains. For example, maintaining a configuration file, generating a coredump or mounting a filesystem all cease to require the mechanism-specific code they would under Unix,

[2] I say "class-based" since "object-oriented" is arguably an inappropriate term to apply to mainstream styles of Java.

such as disk–memory marshalling, object file manipulations, or invoking the `mount` system call. Rather, they are simply rendered as (respectively) accessing a (persistent) configuration object, cloning a process object (likely stopping and persisting the copy), or pushing a new object into some delegation chain. Meanwhile, the late-bound semantics and interactive interface offered by Smalltalk allow it to subsume both programming and, no less important a kind of programmability, "scripting" similar to that offered by the Unix shell.

Descriptive availability

The pervasive metasystem of Smalltalk enables cheap provision of "added-value" services expressible at the meta-level, such as human-readability, visualisation, interactive data editing, debugging, or data persistence. Ingalls anticipates that extending the reach of these meta-level facilities to system-level state would amplify these benefits—when inspecting device state, debugging device drivers, persisting device configuration, and so on.

Interposable bindings

The late-bound, message-based interfaces of objects provide strong interposability properties: clients remain oblivious of the specific implementation they are talking to. In turn, this simplifies the customisation, extension or replacement of parts of a system, all of which can be rendered as interposition of a different object on the same client. Unix often talks about *redirection* instead of interposition; these are synonymous.[3] The concept of interposition presupposes a mechanism by which references to objects are acquired and transmitted. This process is *binding*. In Smalltalk there is one general mechanism for object binding, which is the flow of object references in messages. Binding is also prominent in Unix's design, as we will contrast shortly.

Although these three concerns—programmability, description and flexible binding—are integral to Smalltalk, they are not foreign to operating system designers either, whose work is often evaluated on its conduciveness towards *composition* as a means of user-level software development. We next consider Unix from the perspective of these concerns.

[3] "Redirection" sounds slightly stronger, since it seems to imply unbinding and eliminating whatever entity was previously connected; but consider that with interposition, too, there is no obligation for an interposing object to make any use of the (implied) interposed-on object.

5 Unix: the tick-list

Where do these three concerns—programmability, description and flexible binding—sit in Unix's design priorities? Let us consider firstly the 5th edition Unix described by Ritchie and Thompson (1974).

Programmability

Ritchie and Thompson wrote that "since we are programmers, we naturally designed the system to make it easy to write, test, and run programs". Indeed, Unix exposes multiple programmatic interfaces: the host instruction set (a large subset of which is exposed to the user via time-sharing processes created from `a.out` images), the various system calls (which embed into the host instruction set, extending it with operating system services), the shell (which abstracts the same interfaces in a manner convenient for interactive and scripting-style use) and the C language. These four cohere to some extent. The last of the four, C, is an abstract version of the first—both concern in-process "application" programming. Meanwhile, the shell can be considered an abstract version of the system call interface, since it specialises in file- and process-level operations. I will call the latter kind of programming "file-or-device" or just "device" programming. The remaining twofold distinction runs deep: there are application mechanisms and there are device mechanisms. Applications, aside from trapping into system calls, remain opaque to the operating system; device mechanisms, by contrast, are the operating system's reason for being. I will call this the *application–device split*.

Description

Unix was original in exposing diverse objects—program binaries, user files, and devices—in a single namespace, in a somewhat semantically unified way. This unifying filesystem abstraction includes names and other metadata for all such entities, along with enumerable directory structures. Although primitive, this is clearly a metasystem. For instance, enumeration of files in a directory corresponds closely to enumeration of slots in an object, as expressible using the Smalltalk meta-object protocol. However, Unix's metasystem is highly selective in coverage and content—the system predetermines what state is exposed to the filesystem, the metadata and operations are somewhat specialised for storage systems (sizes, timestamps, etc.), and the facility for exposing state at this meta-level is not extended to application code. While subsequent developments have integrated additional operating system state into the filesystem model, including processes (Killian, 1984; Faulkner and Gomes, 1991) and device state (Mochel, 2005), they have not changed this basic property that use of the filesystem abstraction, and meta-abstraction, is selective and pre-determined. Only some entities are exposed through it; those entities must be chosen in advance, and special-purpose code written to expose them.

Interposable bindings

Thompson and Ritchie stated as a goal for Unix the property that “all programs should be usable with any file or device as input or output”. This is a clearly an interposability property. It was successfully achieved by unifying devices with files—the famous “everything is a file” design. Note, however, its tacit characterisation of applications as having readily identifiable (and unique) input and output streams. The streams stdin and stdout are easily substitutable: they exist in every process, and the parent can bind them (using the dup() system call) to any file or device it can open.

Unfortunately, many other cases of interposition are not supported. One example is how user code cannot quite “be a file”, because only files may be opened by name. (By contrast, for programs using only parent-supplied file descriptors, pipe() serves for this purpose.) The same property means that programs accessing specific files or devices may only be redirected to *user-selected* files if the developers had the foresight to accept the file name as a parameter. Sometimes this foresight is lacking (as known to anyone who has resorted to recompiling a program just to replace a string like ”/dev/dsp”). In Smalltalk this foresight is not necessary, because this kind of definitive early binding is not possible.

(Note that the uniqueness of the “standard” input and output streams is not the limitation here, since in fact a parent process may dup() arbitrarily many descriptors before forking a child, and the child inherits the full set of descriptors. Rather, the limitation is that the set of streams must be enumerable by the parent in advance. This precludes cases where the eventual number or selection of I/O streams depends on program input.)

Contrasts

“Late binding everywhere” is one property which helps Smalltalk ensure interposable bindings, and which on Unix is left for the user to implement (or not). We can note several other contrasts. While the Unix filesystem is a primitive metasystem, it lacks any notion of user-defined “classes”, which in Smalltalk exists to describe commonalities between both user- and language-defined abstractions. In the Unix filesystem, explicit classes are unnecessary, since objects are always of one of three implicit classes: files, directories, or devices. (Later, symbolic links, named pipes and sockets would be added to this list.) Meanwhile, user-defined classes need not be supported because Unix remains pointedly oblivious to user code.

Another way of looking at this is that the operating system concerns itself with *large objects* only. Here we are crudely characterising files as large objects—in contrast to the small units of data that constitute, say, individual records in a file on disk, or indeed, program variables allocated on the process stack or by malloc().

The specification of the mmap() system call in 4.2BSD[4] and the advent of unified virtual memory systems (Gingell, Moran, and Shannon, 1987b) would cement a unification of files and memory objects, but *only* for the case of large objects. This owed primarily to the fact that their interfaces work at page-sized granularity, being neither convenient nor efficient for smaller objects. Of course, Unix filesystems certainly allow files to be small as well as large. "Large objects" is therefore our shorthand for "objects selected by the programmer to be managed as mapped files"—likely for their large size, but perhaps also to enable their access via interprocess communication, as with the example of small synthetic files in the /proc filesystem.

Interestingly, Alan Kay had already observed (and criticised) this preoccupation with large objects in the design of a time-sharing system roughly contemporaneous with (albeit more ambitious than) Unix, recalling as follows:

> "I heard a wonderful talk by Butler Lampson about CAL-TSS, a capability-based operating system that seemed very 'object-oriented'. The only problem—which the CAL designers did not see as a problem at all—was that only certain (usually large and slow) things were 'objects'. Fast things and small things, etc., weren't. This needed to be fixed." (Kay, 1993, p.524).

A consequence of offering only these large-object abstractions is that Unix is tolerant to diversity in how smaller objects are managed. Unix processes happily "accommodate" diverse implementations of language-level abstractions, albeit in the weakest possible sense: by being oblivious to them. By remaining agnostic to application-level mechanisms (in the form of programming languages and user-code libraries), Unix helped ensure its own longevity—at a cost of *fragmentation*. This included not only fragmentation of system- from user-level mechanisms, but also fragmentation *among* system-level mechanisms (noting the various binding mechanisms we have identified), and finally, fragmentation *within* opaque user-level code. Each language implementation must adopt its own mechanisms for object binding and identity, i.e. conventions for representing and storing object addresses. The result of all this fragmentation—which has only grown since Ingalls' article—is an endemic *non*-compositionality which is anathema to the "unified" ideal (held by both Smalltalk and, initially, Unix). It has the effect of ensuring that different software ecosystems are kept separate, and that logically sensible compositions are difficult or impossible to achieve. If Unix's diverse binding mechanisms were not enough fragmentation, the addition of independently developed protocols and data representations "in the small" adds huge impediment to composition.

We should counter, however, that Smalltalk itself has no compelling solution to fragmentation. Its solution is "don't fragment; use Smalltalk for everything!". Unix's lower aspirations serve better in surviving and supporting diverse, independently developed, mutually incoherent abstractions—by virtue of its obliviousness to them.

[4] Although specified in the 4.2BSD design, around 1982, and described in the Programmer's Manual of the 4.3 release in 1986, this interface would remain unimplemented in any BSD release until 1990's 4.3BSD-Reno.

6 From files to. . . : Plan 9 and beyond

Failures of compositionality in Unix have been remarked on since its inception, and often provoke developments which unify system interfaces or mechanisms. Since its initial design, a trend in Unix has been to unify around the filesystem abstraction, by opening it up to new and diverse uses. As noted previously, exposing processes as files (Killian, 1984) created a cleaner and faster alternative interface to process debugging and process enumeration, and this filesystem later evolved into a more general process control interface (Faulkner and Gomes, 1991). VFS (Kleiman, 1986), a kernel-side extension interface for defining new filesystems, later became a central feature of modern Unix implementations. Plan 9, Bell Labs' spiritual successor to Unix, embraced the filesystem to an unprecedented extent. Its design, pithily stated, is that "everything is a [file] server"—a system is a (distributed) collection of processes serving and consuming files, or things superficially like them, using a standard protocol (9P) that is transport-agnostic. Applications serve their own filesystems, and essentially all inter-process functionality is exposed in this fashion. To illustrate the design of Plan 9 and its conducivity to composition, Pike recounted[5] the following impressive anecdote about the design's properties.

> A system could import. . . a TCP stack to a computer that didn't have TCP or even Ethernet, and over that network connect to a machine with a different CPU architecture, import its /proc tree, and run a local debugger to do breakpoint debugging of the remote process. This sort of operation was workaday on Plan 9, nothing special at all. The ability to do such things fell out of the design.

The expanded use of files and servers allowed several simplifications relative to the Unix syscall interface. For example, gone are ioctl() and other device manipulations, process operations such as setuid() or nice(), and the host of Berkeley sockets calls (which had added yet another naming and binding mechanism to Unix). Replacing them are a generalised binding mechanism—essentially bind() by the server and open() by the client—and simple reads and writes to files, including to a selection of *control files*. These are files with arbitrary request-response semantics: a client writes a message, and then reads back a response. Arbitrary communication and computation can be expressed in this way; indeed, it is not-so-uncannily reminiscent of message-passing in Smalltalk.

As the filesystem's use has expanded, its semantics have become less clear. What do the timestamps on a process represent? What about the size of a control file? Is a directory tree always finite in depth (hence recursable-down) or in breadth (hence readable via readdir())? Although some diversity was present even when limited to files and devices (is a file seekable? what ioctls[6] does the device support?), semantic diversity inevitably strains a fixed protocol. The result is a system in which the likelihood of a client's idea of "file" being different from the file server's idea is

[5] in his 2012 SPLASH keynote; slides retrieved from http://talks.golang.org/2012/splash.article on 2017/5/1

[6] ioctl() first appeared in 7th Edition Unix, although calls including gtty() and stty() are its forebears in earlier versions.

ever-greater. It becomes ill-defined whether “the usual things” one can do with files will work. Can I use `cp -r` to take a snapshot of a process tree? It is hard to tell. The selection of what files to compose with what programs, and the fixing-up of any differences in expected and provided behaviour, becomes a task for a very careful user. Unlike in Smalltalk, semantic diversity is not accompanied with any meta-level descriptive facility analogous to classes.

The impressive compositionality of his anecdote Pike credits to the filesystem abstraction of Plan 9, i.e. the property that “all system data items implemented exactly the same interface, a file system API defined by 14 methods”. (Given the few semantics which are guaranteed to be ascribed to a file, 14 seems a rather large number.) Reading more closely, a different property of Plan 9—the network-transparency of server access—is at least jointly responsible. It is no coincidence that Smalltalk objects, like Plan 9 files, are naturally amenable to a distributed implementation (Schelvis and Bledoeg, 1988) and that Alan Kay has recollected how from a very early stage he “thought of objects being like biological cells and/or individual computers on a network”. [7] A Smalltalk-style notion of “object” corresponds closely to the notion of “entity” in the OSI model of networking (Zimmermann, 1988).

Proposals for applying Plan 9’s file-server abstraction still further are easy to find. One example is a replacement for shared libraries: Narayanan blogged[8] a sketch of a proposal for shared file servers replacing shared libraries, using control files to negotiate a precise interface version. In both this case and Pike’s quotation above, what is actually being articulated is the desire for three properties which, of course, Smalltalk already has: a network-transparent object abstraction (an unstated enabler of Pike’s composition scenario), a metasystem (bundled into the unifying API Pike mentions) and late binding (for addressing the versioning difficulties mentioned by Narayanan).

It now seems reasonable to declare “file” (in the Plan 9 sense) and “object” (in the Smalltalk sense) as synonymous. Both are equally universal, more-or-less semantics-free, and deliberately so. However, still distinguishing Smalltalk from Plan 9 is the former’s metasystem and inclusiveness towards objects large and small. Whereas Plan 9 applications must implement a 14-method protocol to reify their state as objects, Smalltalk’s objects have this by default. Moreover, the notion of classes allows at least some semantic description of an object, albeit not capturing those semantics in much detail.

Before continuing, it is worth noting that around the same time as Plan 9, research into microkernels and vertically-structured operating systems (or “library OSes”) brought new consideration of binding and composition in operating system designs (Rashid, Baron, Forin, Golub, Jones, Orr, and Sanzi, 1989; Bershad, Chambers, Eggers, Maeda, McNamee, Pardyak, Savage, and Sirer, 1995; Engler and Kaashoek, 1995; Leslie, McAuley, Black, Roscoe, Barham, Evers, Fairbairns, and Hyden, 1996). These systems were mostly designed with a somewhat object-oriented flavour. Indeed, a key consideration was how to replicate a largely

[7] Various sources on the web attribute this statement to Kay, although I have been unable to find a definitive reference.

[8] at http://kix.in/2008/06/19/an-alternative-to-shared-libraries/, retrieved on 2017/5/1

Smalltalk-like object- or messaging-based abstraction in the presence of the fine-grained protection boundaries—and how to do so with high performance. In at least one case, a dynamic interpreted programming environment was developed atop the core operating system, furthering this similarity (Roscoe, 1995). These systems' results are encouraging testament to the feasibility of acceptable performance in a system of fine-grained protection domains. More recently, Singularity (Hunt and Larus, 2007) is arguably a culmination of work on this topic, offering the radical solution of avoiding hardware fault isolation entirely and relying instead on type-based software verification. Like Smalltalk, these systems offer primarily a grand narrative on how software could and should be structured. Unlike Smalltalk, however, their programming abstractions were something of a secondary concern, lacking a true aspiration to influence the fabric and construction of user-level software. Accordingly, they have been the subject of substantially less application programming experience. For our purposes, protection and performance are both orthogonal concerns, so we avoid further discussion of these systems.

7 Reflections on reflection

We have seen how "object"-like abstractions occur in Unix, Plan 9 and Smalltalk, whereas meta-level abstractions, such as classes, are mostly the province of Smalltalk and are neglected by Unix. Unix, being pointedly oblivious at its core to the structure of user code and data, does not feature a metamodel, or reflective model, centrally in its design. That is, however, not quite the full picture. Clearly, it has long been possible to do *some* reflection in Unix, because programs can be debugged.

We define reflection as *metaprogramming against a running program*, and "introspection" as *self-reflection*.[9] Smalltalk is clearly designed around reflection, and its structured view of objects and classes provides a clear reflective metamodel. By contrast, and as we have come to expect, reflection in Unix has evolved over multiple stages, in a decidedly bottom-up fashion.

Machine-level debugging was supported since the earliest versions of Unix. Source-level reflection was also an early addition and since then has acquired extensive support. This has consistently been achieved using a division of responsibilities which departs considerably from most language virtual machines' (VMs') reflection or debugging systems. The principles of Unix reflection, and their contrasts with VM-style reflection, are summarised as follows.

- Unix requires no cooperation from the reflectee, which might equally be a "live" process or a "dead" coredump. By contrast, a Smalltalk VM actively responds to reflective messaging requests; a dead or frozen VM cannot be debugged or otherwise reflected on.

[9] This is standard, but has the confusing consequence that "reflection" includes the *non-reflexive* case.

- Unix supports *multiple reflected views* of the program: at least source-level and assembly-level views, and optionally others. By contrast, a Smalltalk VM offers a single reflective view, based on the conceptual vocabulary of the unique source language, namely Smalltalk.
- Unix keeps the compiler and reflecting client (a debugger, say) separate, communicating via well-defined interfaces. By contrast, a Smalltalk VM is packaged as an integrated runtime in which communication between these entities occurs by implementation-defined means, via shared data structures that remain logically private from client code. Unix's use of explicit interfaces here necessarily brings strong *descriptive* properties into the metasystem of Unix debugging, in which compiler-generated metadata is particularly crucial.

To realise "no cooperation", the client is given (by the operating system) direct access to the reflectee's memory and registers. Metadata generated by the assembler affords a somewhat symbolic view of these, in terms of named memory addresses rather than purely numeric ones. Metadata generated by the compiler goes much further, affording a source-level view of program state. The latter metadata is exemplified by the DWARF format (Free Standards Group, 2010), whose standardisation began in 1992. In short, debugging metadata provides a medium for compilers to document their implementation decisions as embodied in the output binary, allowing debugging clients to recover a source-level view without building in knowledge of specific compilers.

The metadata-based approach contrasts strongly with VM approaches to reflection, in which the reflecting client consumes the services of an in-VM reflection API and/or debug server. The VM-integrated approach is expedient, since the reflection system and debug server share code in the runtime, and need not describe the compiler's implementation decisions explicitly, making the compiler's code much easier to change. A VM debug server need never disclose the kind of addressing, layout and location information detailed by debugging metadata. But it cannot easily support the post-mortem debugging case, and tightly couples run-time support with compiler: we cannot use one vendor's debugger to debug code from another vendor's (in-VM) compiler. It becomes hard to implement reflection features not anticipated in the design of the reflection API or debug server command language. By contrast, Unix's metadata is open-ended and naturally decouples the distinct tools.

I am not the first to note the architectural significance of decoupling the debugger from the reflected-on program. Cargill (1986), describing his Pi debugger, remarked that "Smalltalk's tools cooperate through shared data structures... [whereas] Pi is an isolated tool in a 'toolkit environment'... interacting through explicit interfaces." In other words, the Unix approach entails inter-tool encapsulation, hence stronger public interfaces than a single integrated virtual machine. One such interface was the `/proc` filesystem (Killian, 1984), co-developed with Pi, which exposes a view of process memory images as files in the filesystem; another "interface" is the exchange of standard debugging metadata.

A couple of decades later, after many years of experience with Smalltalk- and (similar) Java-style reflection, Bracha and Ungar (2004) articulated the "mirrors" design principles which effectively rectified several shortcomings with these VMs' approaches to reflection. Intriguingly, even though these principles were conceived with VMs in mind, with apparently little influence from Unix-style debugging, Unix-style reflection adheres remarkably tightly to the very same principles, which we summarise as follows.

- **Encapsulation**, meaning "the ability to write metaprogramming applications that are independent of a specific metaprogramming implementation", holds that metaprogramming interfaces should not impose undue restrictions on clients, such as reflecting only on the host program (a weakness of Java core reflection).
- **Stratification**, meaning "making it easy to eliminate reflection when it is not needed", intends that reflection can be eliminated on embedded platforms or in applications which happen not to use it.
- **Ontological correspondence**, meaning that metaprogramming interfaces should retain user-meaningful concepts, encompasses both structural (e.g. preserving source code features in the metamodel) and temporal considerations (e.g. the distinction between inactive "code" and active "computation").

The Unix approach to debug-time reflection satisfies all of these principles either fully or very nearly; we discuss each in turn.

Encapsulation

Bracha and Ungar motivated the encapsulation property of mirrors via a hypothetical class browser tool, noting that the Java core reflection APIs bring an unwanted restriction: reflecting only the host VM, not a remote instance. This is a failure of encapsulation, not because it doesn't hide the VM's internals (it does!), but on criteria of *plurality*: clients may reflect only on one specific machine's state (the host machine's); they are provided with only a single, fixed view; and only one implementation of the interface may be present in any one program. Different mirrors offering distinct meta-level views are often desirable, as alluded to by Bracha's and Ungar's mention of "a functional decomposition rather than… leaving that decision to the implementation of the objects themselves". Coexistence of different implementations of the same abstraction is a key property of object-oriented encapsulation, as noted by Cook (2009) and Aldrich (2013). We can also see it as a hallmark of postmodernism in software—an instance of a concern for "many little stories", in opposition to a unique grand modernist narrative.

Unix reflection is very strongly encapsulated, and highly pluralist. The same client can reflect on programs generated by diverse compilers; it is easily extended to remote processes and can reflect on coredumps similarly to "live" processes. The use of metadata as the "explicit interface" means there is no need to fix on a command language, and the client is free to consume the metadata in any way it sees fit. Unix debugging information has a history of being put to diverse and unanticipated uses, such as bounds checking (Avijit, Gupta, and Gupta, 2004), link-time code generation (Kell, 2010) or type checking (Banavar, Lindstrom, and Orr, 1994).

This post-hoc repurposing of pre-existing facilities, or after-the-fact reinterpretation of them, is a similarly postmodern phenomenon.

Stratification

Unix reflection is strongly stratified. This follows from the decision to avoid run-time cooperation from the reflectee (which, indeed, might be dead), and from the decoupling of compiler and runtime. Programs that are not reflected on do not suffer any time or space overhead, yet debuggers can be attached "from the outside" at any point, loading metadata from external sources as necessary. In-process reflection can also be added late, via dynamic loading if necessary. In-process stack walkers are commonplace, found in backtrace routines or C++ runtimes, and it is no coincidence that they are often implemented with metadata also used by debuggers, which enables them to be "stratified" in the sense that code throwing no exceptions pays no time or space overheads.[10] This ability to "add reflection" extends even to source languages such as C which do not specify any kind of introspection interface.

Temporal correspondence

Bracha and Ungar illustrated temporal correspondence by considering the hypothetical desire to "retarget the [class browser] application to browse classes described in a source database". The correspondence refers to a distinction between "mirroring code and mirroring computation"—where "code" means *code not yet activated* (such as method definitions in source code) while "computation" means *code in execution* (such as method activations in a running program). The authors remark that having attempted to do away with this distinction, they found themselves recreating it, in the Self project's "transporter" tool. (This tool could be described as Self's linker and loader. It is significant that image-based systems, such as Smalltalk and Self, are defined by their lack of a batch linker analogous to Unix's `ld`. Rather, images come as whole units grown from a primordial blank canvas; they may not be divided or stitched together from pieces.) Unix exhibits temporal correspondence in the sense that the metamodel of Unix loader and debugger inputs (shared objects, executables, and the functions and data types they define) is separate from run-time details (function activations, data type instances, etc.). In DWARF debugging information, we find the latter are described distinctly, in terms of an embedded stack machine language encoding mappings from machine state (such as a register) to units of source program state (such as a local variable). Consumers of DWARF which care only for static structure can ignore these attributes, and DWARF metadata which omits them remains well-formed.

[10] This is the so-called "zero cost" exception handling design favoured by C++ implementations (de Dinechin, 2000).

Structural correspondence

As defined by Bracha and Ungar, structural correspondence requires that all features of source code are representable at the meta-level. DWARF and similar debugging metadata models a wealth of information from source code, including lexical block structure, namespacing features, data types, module imports, and so on. However, it does not undertake to model every feature—arguably falling short of structural correspondence. In fact DWARF actively *abstracts away* from source, in that its metamodel deduplicates certain language features. For example, a Pascal record and a C struct are both modelled as a DWARF structure_type. Bracha and Ungar envisaged that distinct source languages would offer "distinct APIs", hence that any one reflection interface need only model a single language. However, this one-to-one relationship between a reflection facility and a source language is not always desirable. One intriguing possibility enabled by a pluralist DWARF-style approach is for reflection which actively exposes multiple source-level views of the same objects.

Summary

We have seen how the mirrors principles, starting with the pluralist notion of "encapsulation", reveal a trend from the modernist (a single grand design can be adopted universally) to the postmodern (multiple overlapping designs must be allowed to coexist, imperfectly). Unix's "worse is better" approach, growing reflection facilities organically, allowing compilers and debuggers to co-evolve, and repurposing or extending existing abstractions (such as files themselves, and the earlier assembler-level metadata in object files), has shown a knack for "anticipating the unanticipated". It has addressed, with an air of straightforwardness, a host of problems which Bracha and Ungar (2004) worked hard to vanquish when starting from the virtual-machine approach.

Afterword

Unsurprisingly, the virtual machine tradition has gone on to eliminate the constraints I have described as characteristic of the Smalltalk-style approach. It is interesting to observe that they have done so in a consistently less pluralist fashion than Unix. For example, the Klein virtual machine (Ungar et al, 2005) provided the means to debug a 'dead' virtual machine image using an outside process which embodied the same implementation details. Meanwhile, systems such as the Maxine VM and its Inspector have gone beyond the 'single-level' reflective view, generalising to to 'multi-level' debugging (Würthinger et al, 2010), exposing the different logical layers within a metacircular virtual machine where each 'level' can be seen as a turn in a helical structure (Chiba et al, 1996). The metacircular approach relies on a common, cooperative base platform shared between debugger and debuggee, atop which high-level features are realised in unison (indeed by the same code). By contrast, in Unix the use of descriptive metadata allows these to be fully decoupled, at a gain in pluralism: the target program might embody code generated by many compilers, for many languages, written without mutual awareness. This comes at a cost in code duplication and loss of uniformity: the debugger's implementation of the source languages is separate and perhaps divergent from whatever compilers were used to build the target program. The same distinction is evident in the difference between a self-hosting language implementation and a metacircular one: in the former case, no part of the 'compiling compiler' need be shared with the 'compiled compiler', whereas the latter by definition involves a shared core atop which the remainder is bootstrapped.

8 The Lurking Smalltalk

Whereas it first appeared that reflection in Unix was absent, it turned out to be present in remarkably strong form. Similarly, it turns out, perhaps surprisingly, that the Smalltalk-style facilities we identified in Plan 9—a generic object abstraction, a metasystem (albeit primitive), and interposable late binding—are present in abundance in modern Unices too. However, they are to be found in Unix's characteristic fragmented form. Countless Unix implementations of languages, libraries and tools have grown mechanisms or recipes catering to various requirements for composition and/or reflection. I survey them here, arguing their existence is the sign of a "lurking Smalltalk". Unfortunately, their organic, "evolved" and hence fragmented nature renders them usable only by experts solving specific particular tasks—rather than with the natural generality that arises within a uniform "designed" system. Later I will briefly speculate on future ways out of this cul-de-sac.

8.1 Lurking programmability

Programmability is abundant in Unix ecosystems, but often in awkward-to-use forms. Aside from the shell, the C compiler and whatever other language implementations are available, many applications implement their own configuration language or other "mini-language". Why are these mini-languages necessary? Sometimes they are a domain-specific form optimised for the domain at hand. But in others, they are simply an expedient form of exposing "good enough" configurability or customisability, provided because a full embedded programming language (or several!), although desirable, would be too much effort to achieve. System administrators' jobs would often be easier if they could write configuration logic in a language of their choosing, rather than an idiosyncratic configuration file format.

This is a strong requirement, having no particularly general solutions as far as this author is aware. Perhaps the closest is the facility in Smalltalk-80 permitting a class to reference a non-default compiler object, which takes over responsibility for interpreting the remainder of the class's source-level definition down to Smalltalk bytecode. One limitation of this facility is that the choice of language remains with the class's author, not its client, so cannot be changed on a per-object or per-use basis.

8.2 Lurking metasystems

We saw earlier how the Unix tradition of synthetic filesystems such as /proc or Linux's /sys offers an ad-hoc grafting of specific subsystems' data onto the filesystem, and in so doing, augments them with its primitive meta-level facilities. In turn, these find use via introspection and iteration using standard file APIs, command-line tools, shell-style scripting, and so on. The lack of a metasystem is often apparent here too; for special files' structures are exposed only in documentation, not programmatically, making them impossible to code generically against them. (For example, it is impossible to iterate over all attributes of a stat file in Linux's procfs without writing specialised code that is effectively manually "generated from" the relevant documentation.)

Extensions to the basic Unix file metamodel can be found in the use of tools such as file, which classify files based on their content, or attempts such as MIME (Borenstein and Freed, 1993) at formalising such content. Such attempts so far are highly limited; in particular, the compositional nature of data encodings is not captured (as revealed by MIME types such as x-gzipped-postscript, apparently unrelated to application/gzip). Network services too are minimally and opaquely described, such as by the /etc/services, which defines a quasi-standard mapping from port numbers to protocol names (with implied semantics). Interestingly, an inability to describe the behavioural details of protocols, as opposed to structural information such as fields or methods within classes, is a weakness it shares with Smalltalk's metasystem.

8.3 Lurkingly interposable bindings

The Unix dynamic linker (Gingell, Lee, Dang, and Weeks, 1987a) offers a "preload" interposition mechanism which is commonly used to bootstrap many other feats of interposition by overriding bindings to the C library. For example, applying this to the sockets API enables transparent proxying of applications (as with tsocks and similar tools), and a similar approach may be taken with the filesystem (in tools such as fakeroot or flcow, which provide clients with somewhat modified filesystem behaviour). The composition of separate Unix shared libraries, as commonly implemented for the ELF binary format adopted by all modern Unix implementations (since approximately the mid-1990s), approximates a "mixin"-based inheritance model (Smaragdakis, 2002) similar to that used by Cook (1989) to model various styles of inheritance, including but not limited to that of Smalltalk. In short, although distinctly imperfect in realisation, shared-library mechanisms have (perhaps somewhat by accident) re-created a large space of the class- and delegation-based composition idioms anticipated in Smalltalk's design.

The shell makes a valiant attempt to complete unhandled portions of the Unix composition space we identified earlier (§5). For example, `bash` allows commands like `diff -u <( cmd1 ) <( cmd2 )` for providing pipe-backed file descriptors where a named file is required, or `/dev/tcp/<port>` for redirecting to/from sockets. These approaches are limited: the latter because the shell can only introduce these "magic" filenames if the filename is interpreted by the shell (i.e. for redirection purposes), not when supplied as an argument to a program, and generally because not all functionality is invoked from a shell. Some applications reimplement shell-like facilities in their file-handling code for the same reason, but this reimplementation is both patchy and undesirable. User-level file servers such as Linux's FUSE or BSD's PUFFS (Kantee and Crooks, 2007) provide a more available alternative for file redirection, effectively enabling a Plan 9-style server abstraction, albeit within a host system which does not use them so heavily to such great effect. Union mounts, a staple of Plan 9 namespace composition, are among many common use cases of these systems.

Two further late-binding mechanisms deserve mention. One is arguably the most powerful late-binding device in today's computer systems: the memory management unit. Aside from the program relocation problem it was originally designed to solve, the late binding it provides from virtual to physical addresses has enabled many other operating systems innovations, including the unified virtual memory system. Finally, one must not forget that bindings transmitted in message payloads are routinely rewritten with pipelined use of `sed`, `awk` (Dougherty and Robbins, 1997) or Perl (Wall and Loukides, 2000) as stream rewriters.

8.4 Undoing early binding

The examples we just saw all exploit inherent late-binding in the systems they compose. However, Unix applications can also bind *too early*, creating separate class of problem—"undoing" early binding. Again, many techniques for this have become mainstream. In early-bound programming languages, various dynamic update techniques have been devised (Neamtiu, Hicks, Stoyle, and Oriol, 2006; Makris, 2009). Trap instructions and memory protection, exposed by the hardware and re-exposed in abstract form by the operating system (including BSD's `mprotect()`), provide useful mechanisms for intercepting early-bound code and data accesses (such as respectively for breakpoints and watchpoints). Dynamic instrumentation systems have been used to patch bindings (Hollingsworth, Niam, Miller, Xu, Goncalves, and Zheng, 1997) or completely virtualise compiled code (Bruening, Zhao, and Amarasinghe, 2012). Similar instrumentation techniques can also implement breakpoints

(Kessler, 1990) and watchpoints (Zhao, Rabbah, Amarasinghe, Rudolph, and Wong, 2008) faster than trap-based approaches; these can be seen as interposing on bindings formed earlier, respectively of code-to-code and code-to-data kinds.

9 In conclusion: harnessing the lurking Smalltalk

Languages such as Smalltalk are typically viewed as "grand design", master-blending a host of syntactic and semantic aspects while also seeking to perfectly and completely abstract from the world outside it (other processes) and underneath it (the hardware). This grandeur comes across as overreach from the perspective of the "worse is better" Unix philosophy. Meanwhile, the various fragmentary techniques we have just seen suggest that building a programmable, late-bound, metasystem-enabled environment can be done *using* rather than *replacing* existing Unix-based software. If this is achieved, it will represent a re-imagining or post-hoc recovery of Smalltalk. Several aspects to this are worth noting.

Firstly, the most obvious overreach in the narrative of Smalltalk has been its failure—along with all other high-level languages—to achieve true dominance over its competitors, leading to a fragmented software ecosystem which undermines the intended uniformity of these grand designs. No matter how sublime the uniformity within a Smalltalk VM, or the VM of some successor, in a world fragmented by dozens of competing VMs, practical problems frequently require solutions that reside partly or even largely outside the VM and the language in question. These solutions very often consist partially of file-level scripts or low-level wrapper code, i.e. precisely in the world of the Unix-like operating system that each fragment attempts to escape from.

Secondly, we find the postmodern idea of a whole emerging from "found" parts. Plan 9 was an attempt, its designers wrote, "to build a Unix out of little systems… not a system out of little Unixes" (Pike, Presotto, Thompson, Trickey et al, 1990). An analogous inversion applies strikingly in the context of Unix and Smalltalk. Rather than running isolated Smalltalks (i.e. Smalltalk VMs), each trapped *within* a Unix, we seek instead a path towards a Smalltalk built *out of* the fragmented reality of today's Unix systems. This requires a mindset hardly envisaged in the 1960s and 1970s: to accept the complex reality of *existing* software, developed in ignorance of any particular grand design, and to shift our system's role to constructing *views*, including Smalltalk-like ones, of this diverse reality.

Thirdly, we observe that this is an exercise in *retrofitting*. The need to break apart and back-form a codebase without regard to pre-existing abstraction boundaries is explicitly enshrined in Unix's "worse is better" philosophy. This does not hold abstraction boundaries as sacred; instead it prizes overall simplicity, continuous operation, comprehensibility of all code, and hence the feasibility of invasive modification (across abstraction boundaries if need be) and continuous improvement. A traditional mindset blames the developer who misuses the available abstractions, fails to factor their systems appropriately, or simply uses the wrong system to begin with. A worse-is-better viewpoint instead sees the necessity in accommodating mistakes and abuses, recognising that systems often start life in a specialised form, to be generalised later, rather than the traditional reverse. “Worse is better” actively disclaims the need to “get it right first time”, or even to get it right, merely “right enough”.

Fourthly is the secondary status of language. An implicit goal is of our hypothesised recreation of Smalltalk is to unbundle the underlying machinery, including core abstractions and meta-level facilities, from the programming language itself. High-level languages come and go, and the longevity of Unix—likened by Gabriel (1994) to a “virus”—can be attributed not only to its own lack of a favoured high-level language, but also to its ability to host *many* such languages without prejudice. However, we note that *meta-level* facilities appear less susceptible to design-level churn than base-level language features, because they are one step closer to a relatively small set of recurring concepts (whose recurrence though Smalltalk, Unix and Plan 9 I have been documenting). These abstractions appear to sit comfortably as the “waist in the hourglass”, supporting diverse surface forms for languages above, and running on diverse hardware-supported “big objects” below.

These observations suggest a new programmer-facing role to the operating system, and a response to Ingalls’s position. Far from being replaced by an all-conquering programming language, one interpretation of the role of operating systems is as an infrastructure providing the mechanisms that allow languages to come and go, while maximising the composability of the software written using them. Languages themselves may then become “views” onto a space of objects that is managed by the operating system, rather than by a per-language VM. Ingalls’s perspective, as a language designer looking on operating systems, saw “things that don’t fit in to a language”, and concluded that “there shouldn’t be one”. The culmination of our relatively postmodern perspective on operating systems provides a converse view of languages themselves. A language is a collection of concepts that can be found and recognised within a larger system; there will be many.

Acknowledgments

I thank Michael Haupt for provoking me into writing on this topic, and Mario Wolczko for his thoughtful and detailed feedback at several distinct stages. The present version has been improved by a host of helpful comments over several years from many people, including Peter Kessler, David Leibs, the participants and anonymous reviewers of the Programming Languages and Operating Systems workshop at SOSP 2013, the audience at Carnegie Mellon University during a November 2013 talk, the anonymous reviewers of Onward! at SPLASH 2015, the anonymous reviewers of the History and Philosophy of Programming Conference 2016, the anonymous reviewers of this publication, and many other people who have generously indulged my opinions and arguments on this topic.

References

Aldrich J (2013) The power of interoperability: Why objects are inevitable. In: Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, ACM, New York, NY, USA, Onward! 2013, pp 101–116, DOI 10.1145/2509578.2514738, URL http://doi.acm.org/10.1145/2509578.2514738

Avijit K, Gupta P, Gupta D (2004) TIED, LibsafePlus: tools for runtime buffer overflow protection. In: SSYM'04: Proceedings of the 13th USENIX Security Symposium, USENIX Association, Berkeley, CA, USA

Banavar G, Lindstrom G, Orr D (1994) Type-safe composition of object modules. Tech. Rep. UUCS-94-001, University of Utah, Salt Lake City, Utah, USA

Bershad BN, Chambers C, Eggers S, Maeda C, McNamee D, Pardyak P, Savage S, Sirer EG (1995) SPIN—an extensible microkernel for application-specific operating system services. SIGOPS Oper Syst Rev 29(1):74–77, DOI 10.1145/202453.202472, URL http://doi.acm.org/10.1145/202453.202472

Borenstein N, Freed N (1993) RFC 1521: MIME (Multipurpose Internet Mail Extensions) part one: Mechanisms for specifying and describing the format of Internet message bodies. IETF Request for Comments

Bracha G, Ungar D (2004) Mirrors: design principles for meta-level facilities of object-oriented programming languages. In: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM, New York, NY, USA, OOPSLA '04, pp 331–344, DOI http://doi.acm.org/10.1145/1028976.1029004, URL http://doi.acm.org/10.1145/1028976.1029004

Bruening D, Zhao Q, Amarasinghe S (2012) Transparent dynamic instrumentation. In: Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments, ACM, New York, NY, USA, VEE '12, pp 133–144, DOI 10.1145/2151024.2151043, URL http://doi.acm.org/10.1145/2151024.2151043

Cargill TA (1986) Pi: A case study in object-oriented programming. In: Conference Proceedings on Object-oriented Programming Systems, Languages and Applications, ACM, New York, NY, USA, OOPLSA '86, pp 350–360, DOI 10.1145/28697.28733, URL http://doi.acm.org/10.1145/28697.28733

Chiba S, Kiczales G, Lamping J (1996) Avoiding confusion in metacircularity: The meta-helix. In: Proceedings of the Second JSSST International Symposium on Object Technologies for Advanced Software, Springer-Verlag, London, UK, UK, ISOTAS '96, pp 157–172, URL http://dl.acm.org/citation.cfm?id=646898.756984

Cook WR (1989) A denotational semantics of inheritance. Tech. rep., Brown University, Providence, RI, USA

Cook WR (2009) On understanding data abstraction, revisited. In: Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, ACM, New York, NY, USA, OOPSLA '09, pp 557–572, DOI 10.1145/1640089.1640133, URL http://doi.acm.org/10.1145/1640089.1640133 de Dinechin C (2000) C++ exception handling. IEEE Concurrency 8(4):72–79

Dougherty D, Robbins A (1997) Sed and Awk. O'Reilly Media, Inc.

Engler DR, Kaashoek MF (1995) Exterminate all operating system abstractions. In: HOTOS '95: Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V), IEEE Computer Society, Washington, DC, USA, p 78

Faulkner R, Gomes R (1991) The process file system and process model in UNIX system V. In: Proceedings of the Usenix Winter 1991 Conference, Dallas, TX, USA, January 1991, USENIX Association, pp 243–252

Free Standards Group (2010) DWARF Debugging Information Format version 4. Free Standards Group

Gabriel RP (1994) Lisp: Good news, bad news, how to win big. AI Expert 6:31–39

Galloway A (2004) Protocol: How Control Exists After Decentralization. Leonardo (MIT Press), Books24x7.com, URL https://books.google.co.uk/books?id=7ePFIE5oo7kC

Garfinkel S, Weise D, Strassmann S (eds) (1994) The Unix-Haters Handbook. IDG, San Mateo, CA, USA

Gingell RA, Lee M, Dang XT, Weeks MS (1987a) Shared libraries in SunOS. In: Proceedings of the USENIX Summer Conference, pp 375–390

Gingell RA, Moran JP, Shannon WA (1987b) Virtual memory architecture in SunOS. In: Proceedings of the USENIX Summer Conference, USENIX Association, pp 81–94

Goldberg A, Kay AC (1976) Smalltalk-72 Instruction Manual. Xerox Corporation

Goldberg A, Robson D (1983) Smalltalk-80: the language and its implementation. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

Hinnant D (1984) Benchmarking UNIX systems. Byte Magazine 9(8):132–135,400–409

Hollingsworth J, Niam O, Miller B, Xu Z, Goncalves M, Zheng L (1997) MDL: a language and compiler for dynamic program instrumentation. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, IEEE, pp 201–212, DOI 10.1109/PACT.1997.644016

Hunt GC, Larus JR (2007) Singularity: rethinking the software stack. SIGOPS Oper Syst Rev 41(2):37–49, DOI 10.1145/1243418.1243424

Ingalls D, Kaehler T, Maloney J, Wallace S, Kay A (1997) Back to the future: the story of Squeak, a practical Smalltalk written in itself. ACM SIGPLAN Notices 32(10):318–326

Ingalls DH (1981) Design principles behind Smalltalk. Byte Magazine 6(8):286–298

Kantee A, Crooks A (2007) Refuse: Userspace fuse reimplementation using puffs. In: Proc. of the 6th European BSD Conference (EuroBSDCon)

Kay AC (1993) The early history of Smalltalk. In: The Second ACM SIGPLAN Conference on History of Programming Languages, ACM, New York, NY, USA, HOPL-II, pp 69–95, DOI 10.1145/154766.155364, URL http://doi.acm.org/10.1145/154766.155364

Kell S (2010) Component adaptation and assembly using interface relations. In: Proceedings of 25th ACM International Conference on Systems, Programming Languages, Applications: Software for Humanity, ACM, OOPSLA '10

Kernighan BW, Pike R (1984) The UNIX Programming Environment. Prentice Hall Professional Technical Reference

Kessler PB (1990) Fast breakpoints: design and implementation. In: Proceedings of the ACM SIGPLAN 1990 conference on Programming Language Design and Implementation, ACM, New York, NY, USA, PLDI '90, pp 78–84, DOI 10.1145/93542.93555, URL http://doi.acm.org/10.1145/93542.93555

Killian TJ (1984) Processes as files. In: USENIX Summer Conference Proceedings, USENIX Association

Kleiman SR (1986) Vnodes: An architecture for multiple file system types in Sun UNIX. In: Proceedings of the USENIX Summer Conference, USENIX Association, vol 86, pp 238–247

Leslie I, McAuley D, Black R, Roscoe T, Barham P, Evers D, Fairbairns R, Hyden E (1996) The design and implementation of an operating system to support distributed multimedia applications. Selected Areas in Communications, IEEE Journal on 14:1280–1297

Makris K (2009) Whole-Program Dynamic Software Updating. PhD thesis, Arizona State University

Mochel P (2005) The sysfs filesystem. In: Proceedings of the Linux Symposium, Volume One, Linux Symposium

Neamtiu I, Hicks M, Stoyle G, Oriol M (2006) Practical dynamic software updating for C. In: PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on programming language design and implementation, ACM

Noble J, Biddle R (2002) Notes on postmodern programming. Tech. Rep. CS-TR-02-9, Victoria University of Wellington, Wellington, New Zealand

Pike R (1984) This Dorado is MINE, ALL MINE! Published in 2019 at https://commandcenter.blogspot.com/2019/01/notes-from-1984-trip-to-xerox-parc.html as retrieved on 2019/6/30.

Pike R, Presotto D, Thompson K, Trickey H, et al (1990) Plan 9 from Bell Labs. In: Proceedings of the summer 1990 UKUUG Conference

Rashid R, Baron R, Forin A, Golub D, Jones M, Orr D, Sanzi R (1989) Mach: a foundation for open systems [operating systems]. In: Workstation Operating Systems, 1989., Proceedings of the Second Workshop on, pp 109–113

Ritchie DM (1984) The UNIX system: The evolution of the UNIX time-sharing system. AT&T Bell Laboratories Technical Journal 63(8):1577–1593, DOI 10.1002/j.1538-7305.1984.tb00054.x, URL http://dx.doi.org/10.1002/j.1538-7305.1984.tb00054.x

Ritchie DM (1993) The development of the C language. In: The Second ACM SIGPLAN Conference on History of Programming Languages, ACM, New York, NY, USA, HOPL-II, pp 201–208, DOI 10.1145/154766.155580, URL http://doi.acm.org/10.1145/154766.155580

Ritchie DM, Thompson K (1974) The UNIX time-sharing system. Commun ACM 17:365–375, DOI http://doi.acm.org/10.1145/361011.361061, URL http://doi.acm.org/10.1145/361011.361061

Roscoe T (1995) CLANGER: an interpreted systems programming language. SIGOPS Oper Syst Rev 29(2):13–20, DOI http://doi.acm.org/10.1145/202213.202215

Schelvis M, Bledoeg E (1988) The implementation of a distributed Smalltalk. In: Gjessing S, Nygaard K (eds) ECOOP '88 European Conference on Object-Oriented Programming, Lecture Notes in Computer Science, vol 322, Springer Berlin Heidelberg, pp 212–232, DOI 10.1007/3-540-45910-3_13, URL http://dx.doi.org/10.1007/3-540-45910-3_13

Smaragdakis Y (2002) Layered development with (Unix) dynamic libraries. In: Gacek C (ed) Software Reuse: Methods, Techniques, and Tools, Lecture Notes in Computer Science, vol 2319, Springer Berlin Heidelberg, pp 33–45, DOI 10.1007/3-540-46020-9_3, URL http://dx.doi.org/10.1007/3-540-46020-9_3

Ungar D, Spitz A, Ausch A (2005) Constructing a metacircular virtual machine in an exploratory programming environment. In: Companion to the 20th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM, New York, NY, USA, OOPSLA '05, pp 11–20, DOI 10.1145/1094855.1094865, URL http://doi.acm.org/10.1145/1094855.1094865

Wall L, Loukides M (2000) Programming Perl. O'Reilly & Associates, Inc. Sebastopol, CA, USA

Würthinger T, Van De Vanter ML, Simon D (2010) Multi-level virtual machine debugging using the Java Platform Debugger Architecture. In: Pnueli A, Virbitskaite I, Voronkov A (eds) Perspectives of Systems Informatics, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 401–412

Zhao Q, Rabbah R, Amarasinghe S, Rudolph L, Wong WF (2008) How to do a million watchpoints: efficient debugging using dynamic instrumentation. In: Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction, Springer-Verlag, Berlin, Heidelberg, CC'08/ETAPS'08, pp 147–162, URL http://dl.acm.org/citation.cfm?id=1788374.1788388

Zimmermann H (1988) OSI reference model: The ISO model of architecture for open systems interconnection. In: Partridge C (ed) Innovations in Internetworking, Artech House, Inc., Norwood, MA, USA, pp 2–9, URL http://dl.acm.org/citation.cfm?id=59309.59310