RESEARCH-ARTICLE

The Unix Executable as a Smalltalk Method: And Its Implications for Unix-Smalltalk Unification

JOEL JAKUBOVIC, Charles University, Prague, Czech Republic

Published: 09 October 2025

Citation in BibTeX format

Onward! '25: 2025 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software
October 12 - 18, 2025
Singapore, Singapore

Conference Sponsors:
SIGPLAN

The Unix Executable as a Smalltalk Method

And Its Implications for Unix-Smalltalk Unification

Joel Jakubovic
Charles University
Prague, Czech Republic
jakubovic@d3s.mff.cuni.cz

Abstract

Unix and Smalltalk are very different in the details, but bear curious similarities in their broad outlines. Prior work has made these comparisons at a high level and sketched a path for retrofitting Smalltalk's advantages onto Unix (without compromising the advantages of the latter). Everybody seems to agree on identifying the Unix file with the Smalltalk object, but this still leaves much unspecified. I argue that we should identify the Unix executable with the Smalltalk method. A Smalltalk VM implementation via the filesystem falls out quite easily from this premise; however, the severe overhead associated with Unix processes casts doubt on its practical realisation. Nevertheless, we can see several ways around this problem. The connection shows promise for realising the benefits of Smalltalk within Unix without sequestering the former in a hermetically sealed image and VM.

CCS Concepts: • **Software and its engineering → Object oriented languages**; **Operating systems**.

Keywords: Unix, Smalltalk, unification, connection, isomorphism, homomorphism

ACM Reference Format:
Joel Jakubovic. 2025. The Unix Executable as a Smalltalk Method: And Its Implications for Unix-Smalltalk Unification. In *Proceedings of the 2025 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3759429.3762633

1 Introduction

In "Unix, Plan 9 and the Lurking Smalltalk", Stephen Kell contrasts the visions and realities of Unix and Smalltalk [Kell 2018b].[1] (Note that "Unix" is intended as a stand-in for working with software in terms of large-ish files and processes, which remains the norm whether on Linux, Mac or Windows.) Because of their all-encompassing nature as operating systems, he argues that they are more similar than we might expect, owing to convergent evolution. However, as competing organisms, Unix won in a way that Smalltalk did not: Smalltalk overwhelmingly exists engulfed *within* Unix as a VM *process* plus image *file* (and not, to my knowledge, on any significant amount of hardware). However, Kell has a message of hope for fans of Smalltalk's many comforts: a future Smalltalk lurks *within* Unix, implemented *in terms of* the concepts and infrastructure of the latter, which can be further evolved to fill in the remaining missing features instead of having to replace them wholesale.

I have completed the full journey from initially being skeptical of this thesis to now finding it to be basically correct. In this essay, I will treat Kell's comparison as a basis and draw heavily from it, with the intention of filling in some of the degrees of freedom that it leaves unspecified. I propose that the "high-insight" differences between Unix and Smalltalk, besides Kell's point on Unix lacking a meta-system, are merely naming and granularity. Naming is simple: Unix calls it a file, Smalltalk calls it an object. Granularity is as follows: Smalltalk is made out of many small units, while Unix is made out of fewer larger units which contain the same information as several Smalltalk units.

It follows that one possible way to make a Smalltalk out of Unix is to insist on making files and processes as small as possible.[2] In particular, the Unix program loader invites us to repurpose it as the runner for Smalltalk method activations, with executable files as the methods and processes as the activations. If only such a programming style could be made performant, without the predictable overhead of many tiny processes, it seems as though the niceties associated with Smalltalk programming could be realised quite easily in harmony with Unix — because as Kell observes, they are already there in a frustrated form. Even better, this approach harmonises with Unix's plurality of languages, in contrast

Onward! '25, Singapore, Singapore

ACM ISBN 979-8-4007-2151-9/25/10
https://doi.org/10.1145/3759429.3762633

[1] This essay will draw heavily from Kell's paper, so it is recommended reading for a full understanding. However, the key relevant points will be recapped in §2–4.

[2] It must be understood that such a "prescriptive" approach runs counter to Kell's own vision for this goal. I follow the same analysis of the two systems but then take it in a different direction; more on this in §2.2 and §12.

to the need to write all Smalltalk methods in the Smalltalk language.

The following §2 will motivate why unification of the two systems is both possible and desirable, and sketch what it might look like. §3 justifies the specific comparisons that I will focus on in the essay. I will review key background concepts from [Kell 2018b] in §4, and follow in §5 by interrogating the features of Smalltalk that I value the most — Persistence, Dynamic Software Updating, Uniformity, and GUI Openness — noting how they *almost* exist within Unix, but not to a sufficient extent. I will then argue for viewing executables as *methods* contra generic objects in §6, and for viewing generic objects as *directories* contra generic files in §7. These arguments lead to a proposal for "Smalltix" in §8, which I will concretely illustrate in code samples through §9 and compare with Unix proper. §10 will outline the value proposition from the perspective of §5's feature discussion, before §11 addresses the elephant in the room (performance). Finally, in §12 I will sketch next steps and compare my approach with Kell's own path towards Unix-Smalltalk unification, before concluding in §13.

2 Motivation for the Premise

Comparing Unix and Smalltalk only makes sense if they are of the same type, or auditioning for the same role. Despite appearances, this is in fact the case. We might first consider the programming language named Smalltalk, and note that there is no language named Unix. Clearly, Smalltalk-the-language and Unix are incomparable things. Yet the Smalltalk *language* is only part of a broader programming *environment*, which is (confusingly) also called Smalltalk. *This* Smalltalk constitutes not only a development environment for software applications, but also the run-time environment in which those applications are deployed. The Smalltalk we are interested to compare with Unix is not (just) a language, but a *programming system*:

> A *programming system* is an integrated and complete set of tools sufficient for creating, modifying, and executing programs. These will include notations for structuring programs and data, facilities for running and debugging programs, and interfaces for performing all of these tasks. Facilities for testing, analysis, packaging, or version control may also be present. Notations include programming languages and interfaces include text editors, but are not limited to these. [Jakubovic et al. 2023, §1]

Both Smalltalk and Unix fit this definition [Jakubovic et al. 2023]. They also audition for the role of *operating system,* asserting total control over the hardware, and providing a basic "common language" that all applications must speak (in the sense of a syscall or bytecode interface, or a set of structuring concepts like files or objects).

2.1 Why Unify?

Why might we want to unify Smalltalk into Unix (or the other way round)? As Kell has argued (2018b), and as I will further argue in §5, Smalltalk has many nice features beloved by its community and interested parties. Kell has also argued that there are beneficial aspects of Unix which Smalltalk lacks. Many of the advantages of either system do not seem to contradict each other (i.e. strength A of Smalltalk does not rely on Smalltalk's lack of strength B in Unix). So it is reasonable to hope that we could get the best of both worlds and merge their strengths.

The choice is then to take Unix as a base and evolve it towards Smalltalk (plus Unix), or to take Smalltalk as a base and make it more Unix-like. However, there is a vast asymmetry in favour of Unix as a base: namely, *hegemony.* Unix "won" as an OS in a way that Smalltalk did not. As a result, programmers who wish to enjoy Smalltalk's nice features do so via a Smalltalk *virtual machine* running within Unix. Meanwhile, programmers who prefer Unix simply go ahead and use it; to my knowledge, none are "trapped" within Smalltalk in the analogous way.

> 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 contemporary languages or environments matches the simplicity, uniformity or immediacy of a Smalltalk system. [Kell 2018b, §2.2]

In fact, as will be evident throughout this essay, the hegemonic "Unix" here is better understood as "organising software as a collection of large-ish files and processes with significant and heterogenous internal structure" — a practice that covers Windows and MacOS as well as Linux or Unix proper. The major operating systems are more similar to each other than any of them are to Smalltalk. However, Unix is representative enough to serve as a proxy for the broader

pattern. The ideas discussed in this essay will be adaptable to Windows and MacOS with small modifications.

In short, programming is by default performed on the "Unix" basis as opposed to Smalltalk, the latter approach living within the former. Consequently, it makes more *practical* sense to augment this existing baseline with Smalltalk's comforts rather than the converse. It does not seem like Smalltalk users want Smalltalk to be more like its host Unix, except superficially for reasons of compatibility (e.g. an interface to the host filesystem); running Smalltalk (or any language) in a VM isolates its programs from other programming languages, which can only be surmounted by heavy investment in interoperability technologies. Evolving Unix towards the union of the two systems promises the strengths of both *and* reduced friction with the de-facto standard outside Smalltalk.

2.2 What Concrete Vision are We Aiming for?

The question of what such a union might look like depends on which specific strengths we identify in each system, and whether they conflict with each other. Kell [2018b, §4–5] identifies Unix's "Smalltalk Wishlist" as:

- **Programmatic Availability:** A single, coherent model of programming (Smalltalk language and messaging) usable for user applications and "system" programming alike — contra Unix's split among the host instruction set, C, syscalls, shell scripts, and countless mini-languages.
- **Descriptive Availability:** A rich meta-system describing types of objects and what they can do (the class hierarchy) — contra the "selective and pre-determined" system of file metadata in Unix and its description of file formats / protocols only in non-machine-readable documentation.
- **Interposable Bindings:** Late-bound substitution of interface implementations in a uniform and transparent way — contra Unix's grab bag of limited file redirections, shared library linking, virtual memory management, and `sed`/`awk`-style stream rewriting.

Kell proposes that Unix has *already* evolved much of the way towards these features, and sketches how to hurry it along. This is to be achieved not by crudely slapping them on top of Unix — say, by importing Smalltalk's class hierarchy and demanding that developers use it — but by bridging between the varied partial solutions that already evolved under Unix [Kell 2015]. Although the vision in [Kell 2018b] remains vague to me, it sounds like a system that still *looks* like Unix, while *feeling* like Smalltalk (in terms of the harmony and coherence of the three items in the wishlist). This is because the proposed strengths of Unix over Smalltalk (pluralism, and having evolved a better approach to debugging/reflection [Kell 2018b, §7]) are to be preserved rather than traded off.

My vision in this essay follows Kell's analysis, but focuses on a *different* set of nice features:

- **Persistence:** Program data is safe from process termination or power loss *by default* — contra Unix process memory.
- **Dynamic Software Updating:** Small parts of an application's behaviour can be updated while it is running (no restarts).
- **Uniformity:** Because all objects in Smalltalk use the same binary representation in the VM, developers waste less time dealing with parsing and serialising various file formats.
- **GUI Openness:** GUI components are just persistent, addressable objects like everything else — contra those in Unix process memory.

Just like Kell's wishlist, these are already *partially* realised in Unix, as I will detail in §5. However, my vision then diverges from Kell's: all that is necessary to realise *these* properties is to program within Unix in a particular way (the easier part; §8) and to make it run acceptably fast (the harder part; §11). This is important because it means that Smalltalk fans may be able to program Unix *as if it were* Smalltalk, without being sequestered inside a VM, taking full advantage of compatibility with Unix and different languages within it.

From Kell's perspective, I imagine this "cheats" somewhat on the harder questions, effectively proposing a Unix that *looks like* Smalltalk *in order* to feel like it in the desired ways. In this proposal, Unix is merely the *status quo* of the world outside Smalltalk; I don't personally care for any advantages over Smalltalk and I may be willing to trade them if necessary. Instead, I simply want the above four Smalltalk features *by default* in my Unix programming, without having to use a VM or to roll my own ad-hoc solutions. This might initially seem like an arbitrary demand — but as I will show, Unix has already done most of the work![3] My contribution is thus to show how to seize this *opportunity* to achieve those four features within the world of files, processes, and languages other than Smalltalk.

3 The Many Ways to Compare Unix and Smalltalk

Unix structures a software environment in terms of text files, binary files, directory trees, processes, system calls, and so on. Smalltalk structures the same as a general graph of class objects, instance objects, and method objects containing bytecode. *A priori*, in drawing lines between the two vocabularies, we could join all sorts of pairs. A binary file is a bit like bytecode. System calls are a bit like VM primitives. Are directory trees like instance objects? What about processes?

[3] In other words, the nice things are almost there already; there would be little to write an essay about if everything had to be done from scratch.

As far as I can tell, Kell only draws one such line, between the File[4] and the Object:

> 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. [Kell 2018b, §6]

I am interested in arguing for another pair of concepts to be joined: the executable file and the Smalltalk method. Since the former is still a file and the latter is still an object, there is no contradiction with file-object equivalence.

There are many ways to argue that a Unix concept is "really" a Smalltalk this or that. It's all bits in the end! A text file and a Smalltalk method's source code could have exactly the same contents. A binary file could hold precisely its compiled bytecode. Smalltalk has a concept of a byte stream; someone could surely use this to argue for a lurking Unix within Smalltalk. For my idea here, I'm much more interested in the engineering than the metaphysics; I wish to demonstrate that drawing a connection between the Executable and the Method has *very useful consequences* for realising much of Smalltalk via Unix. I claim that, for this purpose, it is the best out of all the alternative equivalences we could draw for executables or methods.

In agreeing that Smalltalk features can be retrofitted onto Unix, we are saying that Unix already provides enough of the Smalltalk VM infrastructure, in disguise, to make Smalltalk niceties available with only a little work. The spirit is one of improvisation and exploitation of functionality that may have been intended for other purposes. The executable/method equivalence leverages existing Unix infrastructure about as well as I can imagine to achieve an emergent Smalltalk.

4 Prerequisites

Before I discuss my own contributions, I must first import some of Kell's concepts. I will rely on them heavily, but possibly under new labels in order to flavour them for my purposes. The key overarching theme is that Unix *organises software as a collection of large-ish files and processes with significant and heterogenous internal structure.*

4.1 Inter-Process vs. Within-Process Scope

> I will call the latter kind of programming [file- and process-level operations] "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.* [Kell 2018b, §5]

Unix programming is split into two disjoint levels, or scopes. The filesystem is a common namespace of state that all processes can access and use for communication. It has a uniform interface (barring the diversity of `ioctl`s for special devices) which individual programming languages can rely on. The same is true of other inter-process communication mechanisms like pipes and memory mapping, and of the managing of processes themselves. This is Kell's "file-or-device" level; I will refer to it as the "inter-process" scope. Sitting within this scope is what goes on *inside* individual processes and files. Unix stops short of prescribing any structure on the bytes within files or the bytes and instructions within processes, leading to a proliferation of formats that Kell terms *fragmentation.* This is his "application" level, which I will also call "within-process".

4.2 Fragmentation and Large Objects

Fragmentation refers to how every language ecosystem insists on its own version of very similar infrastructure: its own libraries, package manager, build system, debugger — and, more relevant for this essay, its own runtime mechanisms for name binding, memory management, and binary representation of structured data:

> 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. [Kell 2018b, §5]
>
> 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 and the advent of unified virtual memory systems (Gingell, Moran, and Shannon, 1987b)

[4] Technically, it's the file concept in Plan 9, Bell Labs' successor to Unix, but the logic carries over to Unix files in this essay.

> 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 inter-process communication, as with the example of small synthetic files in the `/proc` filesystem. [Kell 2018b, §5]

Kell emphasises that the inter-process abstractions (files and processes) are widely treated as "large objects". In other words, small data (like integers, strings, booleans, but also most programming data structures) tends to reside inside files and processes (as opposed to having many tiny files or many processes embodying a single line of code). Because files and processes have great freedom in choosing how to byte-encode these data structures, fragmentation is inevitable: there are many valid schemes, and there is no strong coordinating force ensuring that all developers — distributed across time and space — choose the same one.[5]

4.3 Endosymbiotic Smalltalk within Unix

> 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. [Kell 2018b, §2.3]

We must be aware of two different meanings of "Smalltalk": one as a peer of Unix, the other as its vassal. The former is the historical Smalltalk, its own operating system with full access to the hardware. Switching my metaphors — the latter is the modern implementation ever since the "files and processes" phenotype reached fixation (whether under the name "Unix", "Windows", or something else). Smalltalk was able to survive extinction by secreting its object memory into a Unix binary file (the "image") and mutating into a Unix binary executable (the "virtual machine"), which could then go on living as an endosymbiotic Unix process. The distinction matters occasionally when the "endo-Smalltalk" allows its Unix substrate to leak through, such as when Squeak asks to save before closing, which would not necessarily happen in "exo-Smalltalk".

[5] Of course, negotiation of common standards does happen *after* the fact, and sometimes it succeeds — just not reliably enough to make fragmentation irrelevant.

5 Smalltalk Niceties Already in Unix (but not the Process)

As recapped in §2.2, Kell identifies a wishlist of desirable features boasted by Smalltalk: programmatic availability, descriptive availability, and interposable bindings. He then explores how Unix grasps in the same directions while stopping short of a comparable realisation. I have nothing to contradict here, but a different set of Smalltalk "niceties" is more salient to me when I ponder what is lacking in the Unix programming experience.

Of course, upon closer inspection it is evident that Unix does *not* lack most of these comforts after all — just that they only exist in *inter-process* mechanisms that I'm not supposed to use as a substitute for application programming practices. Persistence, Dynamic Software Updating, and Uniformity all rhyme in Unix and Smalltalk at the inter-process scope, while GUI Openness is exclusively enjoyed by Smalltalk. If it were possible to program *applications* in Unix as if it were Smalltalk, then all of these niceties could be enjoyed in that domain as well.

5.1 Persistence

The experience of a Smalltalk user is that he creates objects and they stay around until he no longer needs them, or at least until he explicitly discards his references to them. For example, open GUI windows, and the objects backing them, will remain until the windows are explicitly closed. Local variables defined in the Workspace will serve as references for their objects until the Workspace is closed. There is no need for explicit "saving" and "loading". Compare the experience of Unix files in the filesystem: the user creates files and they stay around until he no longer needs them, or at least until he deletes them.

Smalltalk objects are also created *programmatically* by bytecode in running methods. Smalltalk is built to support many small, temporary objects frequently created in this way, in addition to the larger or longer-lived objects that are directly relevant to the user. Garbage collection is used to reclaim the storage used by inaccessible objects.

Unix files are also created and manipulated by code in Unix processes. However, the filesystem is not "supposed" to be used for small, frequent, short-lived files. To my knowledge, no extant filesystems are designed to support different "generations" of files with different sizes or lifetimes. There is no sophisticated garbage collection; instead, a simple reference counting scheme is used.

Smalltalk has an object graph, while the Unix filesystem is a strict tree (with symlinks providing weak references beyond the tree structure). Unlike Smalltalk objects, every file has at least the opportunity for a standard unique human-readable path. However, this may not be an apples-to-apples comparison; internally, we could compare Unix's *inodes* to Smalltalk's object handles, and see the filesystem as a canonical tree structure organising those inodes. It is conceivable that Smalltalk's object graph could have a similar tree structure imposed on it, just that this has not yet been found necessary.[6] Additionally, Unix files often contain embedded references to each other (e.g. file paths in a video editing project, or library includes in source code) which means that the *real* structure of references is a graph after all, even though the "primary" structure is a tree.

As we move into a Unix process, the default persistence of the filesystem inverts into a default *ephemerality* of process memory. All processes terminate sooner or later (whether because of successful completion, a fatal error, user action, or loss of machine power), which means that process memory is not merely a fast extension to the filesystem, but rather a fast *temporary scratchpad*.

Programming languages provide convenient syntax for various data structures in process memory. However, a programmer must remember at all times that when his program mutates these data structures, those changes are unlike changes to the filesystem; any new creations or changes that occur within the process must be explicitly copied back to the filesystem if they are supposed to stick around. Notice that the same is true of Smalltalk objects that have been locally created in a running method activation: the method "temporaries". When an activation returns, each of these temporary variables will get garbage collected unless its object reference was copied to some outer scope. This can be accomplished by returning it, or by copying the reference to an object that already existed before the activation.

Notice that in Smalltalk, it is only necessary to save the *reference*. Smalltalk activations, unlike Unix processes, do not commit locally created data to a private and transient address space; everything is just a garbage-collected object in a universal address space. Conversely, saving a reference to memory in a Unix process is not enough: the entire virtual memory space will be reclaimed by the system, requiring the memory contents *itself* to be copied to somewhere more durable. This invites us to consider Unix process memory as a specialised (accidental) implementation of Smalltalk temporaries with no outside references.

I intend the foregoing discussion to refer to "exo-Smalltalk" on its own hardware (§4.3), occupying the same role as Unix. Of course, the only practically available Smalltalk these days is the "endosymbiotic" version running as a Unix process, such as Squeak. This causes some divergence from what I've depicted. For example, the Squeak VM does not seem to have auto-persistence; when closing the system window, it asks if you want to save the image. However, this is clearly attributable to the fact that the VM is running as a process in Unix.

Additionally, the space available for Unix files (e.g. a 1 TB hard disk) is orders of magnitude greater than that of the Squeak object space, which is capped by available RAM. This makes garbage collection less necessary for the Unix disk image versus the Squeak object graph image file. However, there is nothing in the Smalltalk model that mandates this size restriction; it is just a matter of VM design. An implementation is conceivable[7] that sees 1 TB available for objects, treats RAM as a cache, and automatically persists the changes (reminiscent of Unix's optimisations for file I/O).

5.2 Dynamic Software Updating

In Smalltalk, I can open the class browser, find a method, change the code in the text box, and commit the changes via Ctrl-S (Figure 1). The method will be automatically recompiled, and all future activations of that method in that class use the new template. In Unix, I can open the file browser, find a source code file, change it, and save. I must *manually* recompile[8] and replace the binary (say, in `/bin`). All future processes for that program will then load from the new binary.

A Smalltalk method has only one canonical location, somewhat like the `/bin` (but also `/usr/bin`, `sbin` etc.) directories in Unix. In the GUI, a method is synonymous with its source code; for all the user knows, Smalltalk could be compiling it upon edits, JIT-compiling it, or slowly interpreting it (however, the class object can be used to obtain a reference to the `CompiledMethod` if you really want it). In contrast, the user of Unix sees the source and executable versions of a program as separate files in the tree, and multiple copies may exist distributed around the filesystem.[9]

All of this is to say that Unix already has at least as much "Dynamic Software Updating" (DSU, Hicks and Nettles [2005]) as Smalltalk — at the inter-process scope. The only non-dynamic code updates at this scope might be kernel updates, which resemble Smalltalk VM development from inside a running VM. The system needs to be restarted, yes —

[6] A subset of the object graph — the classes — do have an inheritance tree structure of arbitrary depth. Methods are also grouped into protocols, and classes into packages, but protocols and packages are not themselves tree-structured. The point is that the majority of objects — instances — float anonymously in the graph.

[7] For example, consider GemStone [Copeland and Maier 1984].

[8] Unless I use a Continuous Integration (CI) system or some other automatic watcher.

[9] Of course, method source code and bytecode are also separate objects in Smalltalk, but this fact is less salient to the user than in Unix. Methods are automatically re-compiled when their source code is updated in the class browser; similar behaviour can be set up in Unix by the user's own efforts, but it is not available "out of the box" and need not be uniform across the system. This is an example of Unix's fragmentation contrasting against Smalltalk's uniformity.

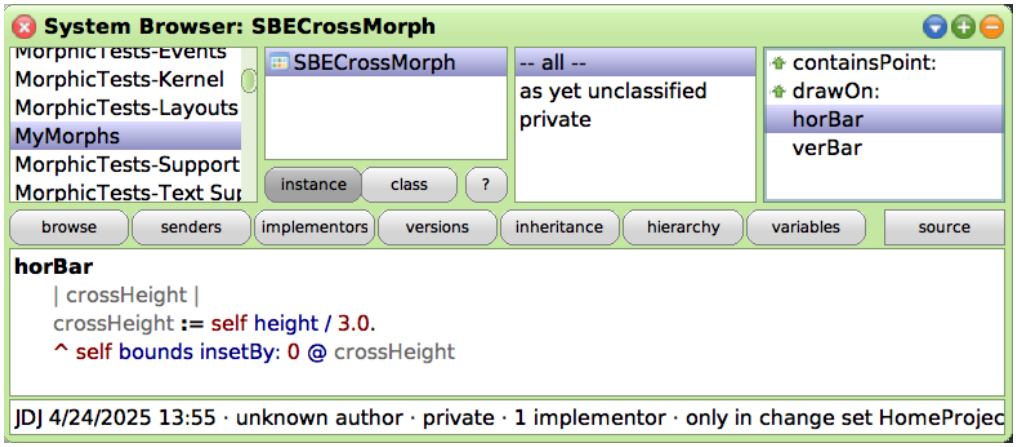

Figure 1. The Squeak Smalltalk class browser. Columns from the left: class categories, classes in the selected category, method protocols for the selected class, methods in the selected protocol. The bottom text area contains the Smalltalk source code for the selected method, which gets recompiled upon committing the edit with Ctrl-S.

but not reinstalled. At a sufficiently large scope, everything becomes dynamic.

Of course, the reason we have the term "Dynamic Software Updating" is because Unix does not have this capability *within* processes. Dynamically-linked shared objects do provide for it at a coarse grain. However, applications are not normally packaged as a collection of many dynamic libraries, one library per class or function. Consequently, updating code at a finer granularity is an established research area [Ahmed et al. 2020; Neamtiu et al. 2006].

However, Smalltalk method activations likely have similar limitations. I am not aware of any research on patching portions of Smalltalk bytecode within running activations, but crucially, I am also unaware of any demand for it.[10] Yet there is clear need for within-process DSU in Unix, because processes are much larger and more complex than a single small method.

5.3 Uniformity

In Smalltalk, an object is a set of named references to other objects (some of these will be optimised as tagged integers and the like, but they remain real objects at the user level). One special reference is to its class, which is used to resolve messages. All objects have named instance variables[11] which can refer to any object. Objects have a binary encoding prescribed by the runtime, but they are a simple structure and all use the same encoding scheme.

Compare the Unix directory, a special type of file which *directs* you to other files via simple names. Directories have a uniform access API via syscalls (`readdir`). Symlinks provide reference beyond the tree structure (they are weak references like object addresses, and will dangle if the destination gets moved).

Smalltalk is designed to support objects large and small. The filesystem, in contrast, was historically associated only with "large objects". As a result, directories and symlinks are rarely used to structure data all the way down to the finest granularity (i.e. with individual integers or strings as the leaf nodes — `procfs` is the exception that proves the rule). Instead, the norm is for directories and symlinks to organise data at the coarsest granularity, and then to leave further structuring to the individual leaf nodes — regular files.

In contrast to the `readdir` structure mandated for directories, regular files only have serial byte access as their interface. Since files are expected to contain further substructure, each one has to solve the same problem that the filesystem already solved: how to byte-encode internal references, nesting relations, primitives such as ints and strings, encoding of enum constants, and external references to other files. This hands-off approach by Unix invites extensive *fragmentation* (§4.2) as myriads of independent developer communities choose different encoding conventions in parallel. A multitude of binary and textual "file formats" proliferate and compete with each other to become de-facto standards. In Smalltalk, these would all just be more sub-graphs of the all-encompassing object graph, encoded in the same way at the binary level.

The same applies to executable formats too, such as ELF. Process overhead is non-trivial, and storing all program state in the filesystem is seen as too expensive. Therefore, executables and processes must be "large objects" which package together many individual program components (data and functions, or even "objects" and "methods" within the process). An analogous proliferation of different in-process

[10] However, there has been recent work in updating "active functions" in Unix processes [Strömbäck and Varró 2024], and such techniques could port over for updating running activations in Smalltalk.
[11] Or numbered indexes, as in `Array` and its subclasses.

binary formats (ABIs, runtimes, and VM data encodings) occurs here, though there is no corresponding proliferation of text formats.

In summary, Unix has an overarching uniformity in its data structuring mechanism, similar to Smalltalk's objects, except that it yields to anarchy once a certain scale is reached. Smalltalk's designed support for object graphs at a fine granularity limits the potential for such "fragmentation of conventions". After all, there are fewer ways to represent a single integer than a tree or graph of integers. Interestingly, source code in Smalltalk could in principle be fragmented across syntaxes that compile to Smalltalk bytecode. However, it appears that in practice, Smalltalk syntax is accepted as a suitable baseline; domain-specific "languages" are instead approximated via the flexibility of the message sending mechanism.

5.4 Counterexample: GUI Openness

It has long been lamented that GUIs do not compose as effortlessly as command-line pipes, if at all. It is natural to expect some additional challenge merely from the fact that an input/output connection (a point-to-point composition) is vastly simpler than what GUI composition would meaningfully require (composition across 2D surfaces, if you like; "phenotropic" composition [Lewis 2018]). However, the actual difficulty of GUI composition, and programmatic GUI manipulation, seems to be much higher than its essential complexity.

Command-line pipe composition is only possible because stdin and stdout are standard, public names associated to each process. GUI composition would require a similar "GUI openness" in the first place: GUI widgets would need *stable, public names* visible between applications.

In Smalltalk, GUI components are persistent, addressable objects. In the standard Morphic GUI framework, you can obtain a reference to a window object and query it to obtain sub-components. The standard Smalltalk tools (class browser, method finder, debugger) know how to communicate with each other and fill in GUI fields with the appropriate data.

The hypothetical analogue in Unix would have these GUI elements as persistent, addressable directories and files. However, what we actually see is that GUI components are data structures encoded in some binary format with each process' private memory. Not only are these not addressable from other processes (and probably encoded differently, as one of many fragmented GUI implementations) but they are also ephemeral. Even if we learned that a GUI component lived at `0x123456` in some process, this reference would cease to be valid after the process terminated (which is likely to occur eventually, whether by the user's deliberate action, a bug in the program, or the machine turning off). We could hope to address the component in the executable *file* instead, but this is not guaranteed to work. If we are lucky, it might exist as a static data structure. However, the most likely situation is that the executable only contains *instructions* for *creating* the GUI in a live process.

Unlike the other Smalltalk niceties I have listed, this example does not seem to be exhibited in Unix even at the inter-process scope. It seems to me that the privacy, ephemerality, and fragmentation of the Unix process are the root causes, leaking into and tainting the GUI components that call it home.

6 Is the Unix Process Just an Object?

Bracha [2022] analyses the Unix process as a disguised object, with ELF as its serialisation format. However, the ELF binary is not only a serialisation format (which could just as well be streamed down a network socket, rather than stored on disk) but is also the only *persistent* form of such an object. If the process is an object, with methods and data, then it is a highly unusual object whose data is inherently ephemeral, since any in-memory data structures will not survive the process' inevitable termination at some future point. Smalltalk does not have a native notion of "disk" versus "memory"; if these concepts are present, it is for compatibility with the host Unix environment. Smalltalk simply speaks of objects, which stick around as long as the user needs them, and it uses garbage collection to give the illusion of unbounded capacity with finite hardware resources.

It is true that ephemeral, short-lived objects exist in Smalltalk. They are objects created during method activations, bound to local temporaries, and whose references are not passed outside the activation. Conceptually, they are scoped to the lifetime of the activation: when the method has finished executing, the sole remaining references to the objects will be severed and they will be marked for garbage collection. The mandatorily ephemeral nature of the Unix process (as opposed to the file) means that it can only serve to model such a *locally scoped* object or a collection thereof.

We might be tempted to regard the ELF *binary* as the more general object, since (being a file) it is at least persistent. However, we'd face a different limitation: Unix processes generally do not or cannot modify their binaries, instead preferring to modify external "data" files. Perhaps we could get around this by analysing the binary as an *immutable* object. However, we would then have trouble explaining what is going on when Unix loads the binary into a process. Unix takes an immutable object, copies it into fast volatile memory, and now treats it as mutable — but only in this private, temporary namespace. Its internal methods may modify the in-process copies of its data, but these changes will (by default) have no permanent effect. Why?

I think that, under such scrutiny, the "process/binary as object" model raises more questions than it answers. However, it is understandable why we find it so tempting. For one, the process has private state (encapsulation!) and it has public interfaces (`main`, signals) which it can respond to

however it wants (messages!). It may advertise internal functions by name, which could be implemented differently in a different executable (polymorphism!). And if those weren't enough, we find Alan Kay himself criticising Unix processes as failed objects in his famous keynote, noting how the number 3 cannot be a process in Unix the way it is an object in Smalltalk [Kay 1997].

Nevertheless, a different model succeeds at explaining the puzzles. A binary resembles an instance of `CompiledMethod`, and its process resembles the corresponding `Context` activation object, more than either of them resemble a generic object. Private process state appears optimised for local temporary variables (at least while a lack of external references dooms them to garbage collection). The single standard public interface `main` is nothing but the entry point for running its code. (Consider also the established formalisation of objects as closures [Reddy 1988].) In hindsight, given these unintentional similarities, Alan Kay should have lamented that "*return* 3" cannot (morally) be a Unix process the same way it is a perfectly valid Smalltalk method.

This model lets us account for the observed facts more neatly:

- The binary (persistent, functionally immutable) gets loaded into a process (mutable but ephemeral) because the former is a method and the latter is one of its activations.
- The binary is functionally immutable in the same way a Smalltalk method is immutable: running instances seldom make small changes to the binary/method, but the user is free to recompile a *new* binary/method and replace it wholesale.
- Process memory is ephemeral because it is being used as a shortcut for garbage collection. The contract with the programmer is: "any data not saved to a file is scoped to the activation".

While this does work as an account of how Unix happens to implement parts of Smalltalk in disguise, real Unix *programs* only follow this model in a rough and obscured form. A Unix that *strictly* followed this correspondence would look very different; I will detail it later under the name "Smalltix" (§8). A real Unix binary fuses together *many* compiled methods, along with loading/saving code for its persistent mutable data. The persistent "objects" which these methods modify are similarly fused together in the binary or text files accessed by the process. I will say more on this in §9.2.

7 Is the Smalltalk Object Just a File?

At this point, I think it is worth re-examining the file-object equivalence. The guaranteed interface of a generic file is that of a byte stream: read and write, plus metadata like timestamps, permissions, and size. The guaranteed interface of a generic object in Smalltalk has a little more structure. It can be sent a message, but its interface also includes *named slots* (visible only to its methods). For a *file* containing a similar key-value structure, Unix prescribes nothing: a thousand flowers bloom in the form of syntaxes (e.g. JSON) and binary formats. Yet the Smalltalk language (via its compiler, bytecode, and ultimately the VM object format) presents one *unified* way to access object slots. There is even a way to enumerate an object's slots via reflection.

Hold on a minute — Unix *does* prescribe a similar interface, in the form of a special type of file known as a *directory*. Files in a directory can be enumerated and accessed only by the *uniform* interface of standard system calls.

> 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. [Kell 2018b, §5]

For the goal of hacking a Smalltalk via Unix concepts, this is surely a better equivalence to draw. If we translate objects to generic files, we are immediately confronted with fragmentation in how to encode their slots and references. If we translate objects to directories, the mapping writes itself: slots are generic files under the directory, and references are symlinks (or just files containing a path). The object-directory equivalence is a better choice for my goals, so I will adopt it from this point onward.

A possible objection: how are we to reconcile method-executable equivalence with object-directory equivalence? After all, a Smalltalk method is an object, but an executable is not a directory! I will have to admit this as an unprincipled exception, happily noting that *processes* already have corresponding directories via `procfs`. There is research that could remedy this inelegance by applying the `procfs` approach more generally; see §12.

8 Smalltix: a Unix of Small Files and Processes

To summarise the foregoing discussion: at the inter-process scope, Unix resembles Smalltalk with the concepts renamed and (recalling Kell [2018b]) lacking a meta-system. However, this equivalence breaks down at the within-process scope. Since most user applications consist of one process or a small number of them, the *application* programming experience diverges sharply from Smalltalk. This suggests that programming under Unix *could be* largely Smalltalk-like if we take the method-executable equivalence and force it through to the application scope. In other words: we insist on constructing applications from many tiny files, processes, and directories, and assume away the performance overhead (which we will revisit in §11). I will call such a system *Smalltix*.

I will now proceed to demonstrate more concretely what this could look like, given that I identify the Smalltalk method with an executable file and the activation with a process. I will refer to this as the **Smalltix Connection**,[12] or the **Connection** for short. I must emphasise that instead of strictly focusing on ELF *binaries*, I think it is appropriate to cast *any* executable file in the role of method. If it happens to be an ELF binary, then the loader will load it into a process and run the code. If it happens to be a text file with a shebang at the top, Unix will load the corresponding interpreter process (e.g. a shell, or Python) and feed the code through its stdin. The analogue in Smalltalk would be "executing" a string by passing it as an argument to an interpreter method. Focusing on executables means that compilation vs. interpretation, and the particular language used, are mere implementation details. This allows Smalltix to be language-agnostic down to the method level.

As mentioned in the previous section, I will also identify the Smalltalk object not with a general file, but with a directory. However, this is not as crucial as the Connection; the demonstration could probably be reworked based on a file format for individual objects.

9 A Smalltalk VM via the Filesystem

In effect, by equating Smalltalk methods with Unix executables, we are treating the Unix loader as that portion of the Smalltalk VM which instantiates method activations. Furthermore, while activations are (conceptually[13]) ordinary garbage-collected objects in Smalltalk, Unix process memory is inherently volatile; it is not as if every terminating process gets core-dumped to disk until the disk is full. Thus we are using the machine's RAM as a highly optimised shortcut for garbage collection of data that we expect to be ephemeral. It is conceivable that a hardware implementation of Smalltalk might use such an optimisation, moving activations to permanent storage as soon as the user obtains a reference to one (e.g. via an open debug session).

We can continue to locate pieces of a Smalltalk VM in the Unix "between-process" scope and fill in any remaining infrastructure ourselves. As always, there are many choice points of exactly how to implement a particular abstraction. I do not have the scope to discuss all of these or to thoroughly defend why I chose a particular one. I will simply present the following outline of one possible realisation of Smalltix, allowing future critique and experiment to flow from there.

9.1 SBECrossMorph Example Code

As a running example, I will adapt the class `SBECrossMorph` from the *Squeak By Example* book [Rein and Thiede 2023, Ch. 12]. `SBECrossMorph` is a graphical element ("Morph") in the shape of a coloured + shape. Consider an instance `aCross` of class `SBECrossMorph` and its slots (a.k.a. instance variables) `bounds owner submorphs color`. Each of these slots points to some other object.

The class of `aCross` could be considered to be a pseudo-slot; in Squeak, the method `class` is implemented as a VM primitive, but its usage resembles a slot getter method. Similarly, we model the `SBECrossMorph` class object as having pseudo-slots `superclass` and `methods`. The latter is an object containing slots `horBar verBar drawOn: containsPt:` which point to various methods. We can use the Smalltalk source for `horBar` as a jumping-off point:

```
SBECrossMorph >> horBar
  | crossHeight |
  crossHeight := (self height / 3.0) rounded.
  ↑ self bounds insetBy: 0 @ crossHeight.
```

The method implements the horizontal bar of the cross shape. It returns a rectangle with coordinates corresponding to the middle third of the enclosing bounds rectangle. For anyone unfamiliar with Smalltalk syntax, it is equivalent to the following pseudocode:

```
crossHeight =
        (self.height() / 3.0).rounded();
return self.bounds()
           .insetBy(
              0.makeCoords(crossHeight)
            );
```

According to the Connection, this method just needs to be translated into an executable file. We can write the code in *any* language — this is Unix, after all — and we can either put an interpreter shebang at the top, or subsequently produce a compiled binary like Smalltalk does with its bytecode.[14] Since we will be using the filesystem so much, Bash seems like a good choice.

```
self=$1
tmp1=$(send $self height)
tmp2=$(send $tmp1 / float/3.0)
crossHeight=$(send $tmp2 rounded)
tmp4=$(send int/0 \@ $crossHeight)
tmp5=$(send $self bounds)
send $tmp5 insetBy: $tmp4
```

[12] By very loose analogy to the Levi-Civita Connection: there will be other Unix-Smalltalk connections, but the Smalltix Connection has some particularly nice properties. I am hesitant to call it an isomorphism for reasons I will discuss in §9.2.

[13] Technically, this is an illusion maintained for the convenience of the Smalltalk user. Activations are normally optimised VM constructs, only becoming garbage-collected objects on-demand when the user reifies them (e.g. in the debugger, or via the keyword `thisContext`). Of course, the system is set up so that any activation to which the user has a *reference* behaves according to the conceptual model.

[14] Of course, Smalltalk does this upon committing a change to the method source code box in the browser, so it is transparent to the user.

It's a little ugly, as I've had to unfold the natural expression structure into explicit intermediate variables. But this is just an illustration; it could even be a compilation target of the original Smalltalk code. Writing it forced me to make decisions about how to encode Smalltalk VM concepts:

- Where a real VM has object addresses, we have file paths.
- The first argument to the script shall be a path to the `self` directory (i.e. the receiver object of `horBar`).
- Message sends are accomplished via an executable called `send`.
- A method returns a value via its stdout, hence the `$()`. The final send prints its return value to the script's stdout, thereby returning it from the script, so it does not need wrapping in the same way.
- A method takes its arguments via the standard Unix command-line `argv`.
- A VM's "tagged" integers/floats are encoded as paths under `int/` and `float/`. Those don't have to be real directories, or if they are they needn't contain anything. The paths can simply be recognised by `send` and optimised appropriately.

Next, I must ask: what does the `height` script look like? In Smalltalk, it forwards the `height` message to the self's `bounds` slot:

```
Morph >> height
  ↑ bounds height
```

This requires me to encode slot access, which is distinct from sending messages (note that I will omit the `self=$1` line from now on; assume it is always the first line):

```
bounds=$(getSlot $self bounds)
send $bounds height
```

OK, I've passed the buck to a `getSlot` primitive, implemented as an executable as usual. For simplicity, I will continue to implement the *primitives* in Bash (not just the methods). However, I hope it is clear that *any* language can be used for each of these files, because we are using Unix's inter-process media (`argv[]`, `stdout`, and the filesystem) for communication. At this level, the language really does appear to be just an implementation detail!

The `getSlot` primitive is easy given object-directory equivalence:

```
slot=$2
cat $self/$slot
```

Now to define `send`. I will factor out the actual class/superclass lookup chain into `bind`, following the terminology of Piumarta and Warth [2008]:

```
recv=$1
selector=$2
shift 2 # Now $@ will refer to the rest
        # of the args
args="$@"
method=$(bind $recv $selector)
$method $recv $args
```

We obtain a local reference to the method object (i.e. path to its executable file) via `bind`, and then execute it on the receiver with the arguments. What of `bind`?

```
recv=$1
selector=$2
class=$(cat $recv/class)
while [[
    ! -e $class/methods/$selector
  ]]; do
  class=$(cat $class/superclass)
done
cat $class/methods/$selector
```

I have chosen to encode the class of an object simply as a slot (i.e. a file under its directory). I also encode a class' methods under a `methods/` directory under the class object-directory. This is not inconsistent with object-directory equivalence, because I am working on the VM internals here; I shall consider it an error for method code to try and access these files. (If I had used a syntax like Smalltalk to write method bodies, I could automatically translate each slot reference into its appropriate filesystem encoding without worrying about such things. The risk occurs because I'm writing methods in a language with ordinary file access, the equivalent of arbitrary pointer access in a VM, which not even Smalltalk bytecode permits.)

`bind` first brings the receiver's class reference into its local process-scoped namespace (i.e. a Bash variable). It then searches up the superclass chain (`superclass` being implemented as an ordinary slot here) until finding a file under the right name that actually exists. This assumes a convention where nil references are encoded as the absence of a file; having a Nil object with a tagged reference is probably a better approach.

9.2 Smalltix vs. Unix

I have sketched a way to encode Smalltalk concepts fairly naturally via existing Unix concepts. The resulting system, which is Unix but programmed in a specific way, I call Smalltix. I originally wanted to call the Method-Executable equivalence a "Smalltalk-Unix Isomorphism", but this would not be accurate. I called it the Smalltix Connection because, insofar as it is an isomorphism, it only draws a bijection between Smalltalk and *Smalltix*, not Unix. To assess what implications the idea has for Unix, we have to ask how we get from Smalltix back to Unix. The root of the differences is granularity: in short, a Unix file is many Smalltix files fused together.

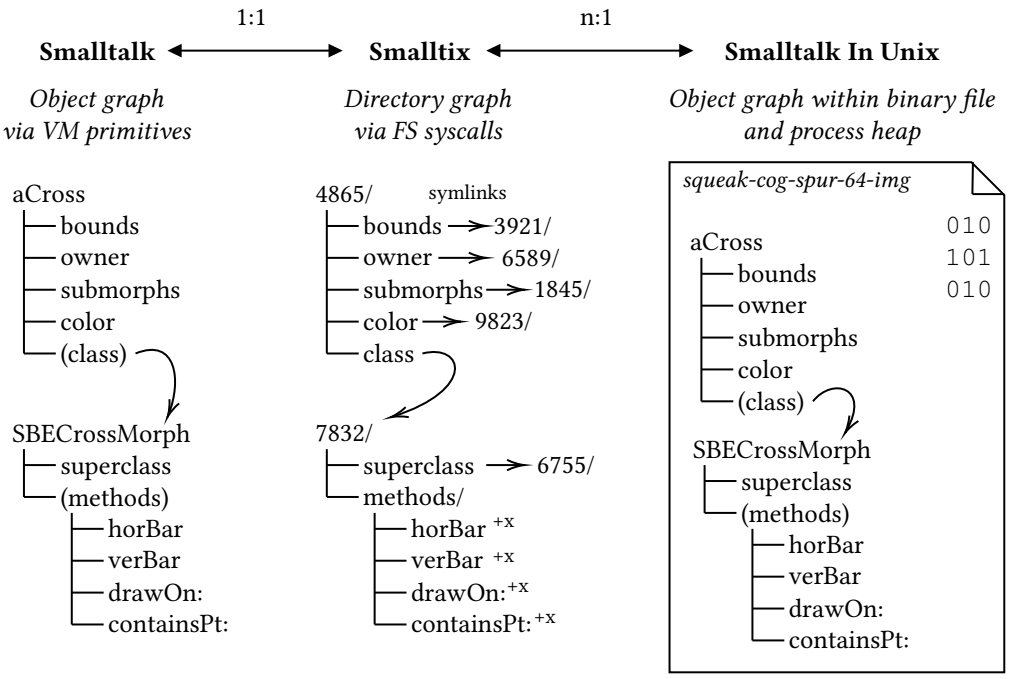

Figure 2. The Smalltalk-Smalltix Connection is one-to-one, based on object/directory and method/executable equivalence. Many units in Smalltalk/Smalltix end up fused into one large unit in Unix. For example, Smalltalk objects live entirely inside one primitive Unix object (the Squeak image file).

For example, in Smalltix there is one process per activation of a method, while a single process in Unix typically contains many smaller functions (which are synonymous with methods for this analysis; OOP is merely a way to select which specific function should run in response to a message).

A single-threaded process may only have one current activation at any time (the one at the top of its stack) but in general, it is being used as a sort of optimised composite activation, or activator, of a particular tree of function calls for which it was specialised. In Smalltix these calls would be stitched together via stdout and argv, as in the `horBar` example. We know that using many processes in this way would have very serious overhead (to be addressed in §11), so bundling the separate functions together in a single process has been the only viable way to do it for all of Unix's history. There is an important difference beyond performance, however: each individual Smalltix method may be implemented using any language, whether interpreted or compiled. On the other hand, if we fuse them together in the same Unix process, and opt to have them communicate via shared memory addresses, then they all have to be written in the same language (or at least target the same VM or language runtime).[15]

So we can take a tree of Smalltix methods growing from some root method call, and statically link them into a single binary (or copy each as a function into a single Bash script, etc). Do we have a Unix program yet? Not quite, because all access to state (other than the intermediate values between functions) is still going through the Smalltix filesystem. This is very unlike Unix.

Typically, a Unix program does not just define code; it also defines *data structures* that live in process memory. These are the usual target of code in the program; interaction with the filesystem only occurs during process initialisation (when the data is "loaded" from a saved state) or termination (when it is "saved" back to the filesystem).

Moreover, it is not as if every tree of heap blocks ends up saved to a large directory tree of many tiny files. Instead, a small number of larger files is used, each one of which defines its own internal structure via a *file format* (or, in the case of text files, a *syntax*). Essentially, many Smalltix data files (i.e. object references, tagged integers, and so on) and directories (objects) are fused into a single Unix data file. As a consequence of this fusion, the correspondence between file

[15] More or less. Under suitable conditions, some languages can compile to "native" object code linked together in the same binary (e.g. C, C++, Pascal). All languages can *in principle* be made to target native code, the Java VM, JS, and so on, but in practice there is only a narrow selection. The point is that in general, it is very hard to build a single executable from a free choice of several languages, while it is comparatively trivial for heterogenous executables to collaborate via the filesystem. Kell's liballocs [Kell 2015] seeks to achieve language agnosticism *within* the executable — that is, runtime agnosticism — by bridging the fragmentation of process memory contents; this is very different to the language agnosticism in Smalltix. The two approaches will be compared in §12.

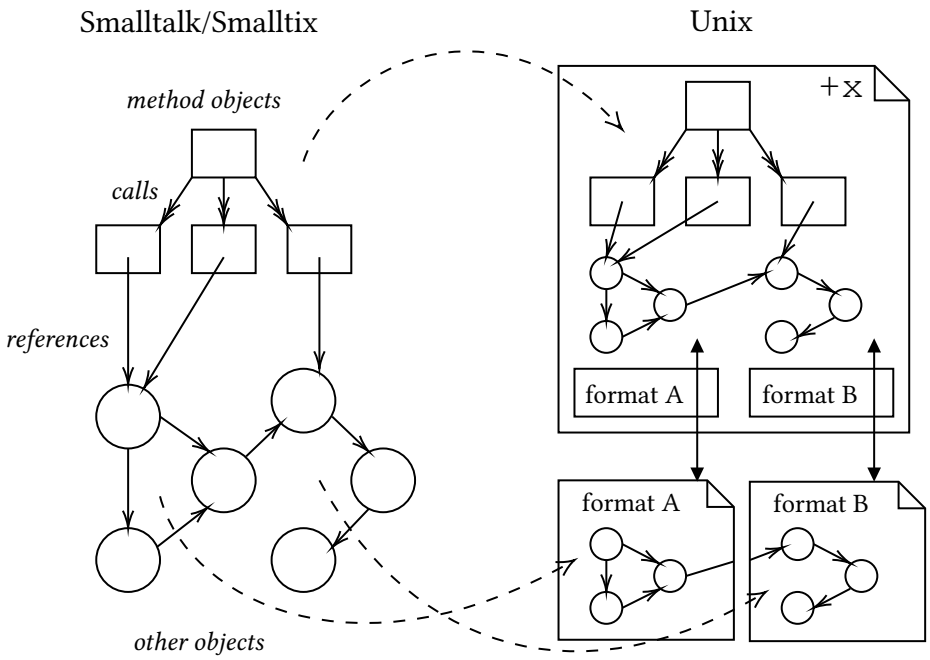

Figure 3. Unix applications from Smalltalk's perspective. Trees of method calls are fused into a specialised runner executable. The graph of objects referenced by the methods is partitioned and serialised into bytestreams in various formats. New methods are added to the executable to re-create the original object graph in RAM and keep the source bytestreams in sync. Note that there are ten Smalltalk objects on the left, and only three Unix files on the right. However, the diagram is not "to scale": there could realistically be 10,000 objects on the left, and still only three files on the right.

contents and heap contents must get encoded procedurally[16] as additional parsing/serialising code, which has no reason to exist in Smalltix.

That's it, as far as I can tell: if we perform the fusions and add the extra parsing/serialising code, we arrive at an ordinary Unix program. To summarise, in order to go from Smalltix to Unix we perform the following steps:

1. Fuse method call trees into a single larger executable under the same language or runtime. Each method (formerly an executable file) becomes a language-level "function", "procedure", or even "method" sub-unit inside the executable.
2. Fuse the object graphs accessed by the call tree into a small number of files, each of which may encode the object graph with its own binary or textual format.
3. Add in new loading and saving code which marshals those object graphs between the files and the process' address space.

The bridging between Smalltix and Unix is illustrated in Figures 2 and 3.

10 How to Enjoy the Smalltalk Niceties in Unix — Today!

The Smalltix Connection gives a novel perspective on why Unix programming fails to fully realise Persistence, Uniformity, Dynamic Software Updating, and GUI Openness. Let us peer naïvely through Smalltalk glasses and view Unix *as if it were trying to be Smalltix*. We would note the following peculiarities:

- Unix applications appear to be written as one enormous method[17] (run as a process) and a small number of large bytestream objects (data files). This is odd; large methods are hard to comprehend and cannot be updated piecemeal, and the bytestreams must contain significant object-like structure anyway.
- Instead of splitting the method into smaller methods and the large objects into smaller objects, a mini-Smalltalk (language runtime) is simulated inside the giant method (process) using its local variables (virtual memory). Differently-shaped and more heavily restricted *pseudo*-objects and *pseudo*-methods run along inside the true method. This is odd; why go to all the

[16] Though declarative descriptions are gaining prominence, for example Kaitai Struct (https://kaitai.io).

[17] Or perhaps, where dynamically linked libraries are used, a Lisp-like method *combination* assembled from several enormous methods.

trouble of building an entire separate object system when one already exists (the filesystem)?
- Some of these pseudo-objects (process data structures) are initialised by extracting byte ranges from one of the large objects (they are loaded from a data file). When the method returns (process termination), these pseudo-objects then compress their contents back into those byte ranges (save to data files). This is a highly consequential optimisation; did profiling recommend it? We create a local pseudo-object, which will get garbage collected upon return, initialise its state from a longer-lived object, do useful work on it, and then have to save this work from garbage collection by echoing the changes back to the longer-lived object. Instead of all that, why not just refer to *real* objects (directories) in the method and work on those, cutting out the middleman?
- Research is devoted to patching parts of method activations (processes) while they are running [Strömbäck and Varró 2024]. Is this really necessary? *Individual* methods (executables) can already be replaced while the whole system is running. If the current large methods were split up, and their internal pseudo-methods elevated into real methods, then this might not even be needed.
- GUIs run entirely within a single method activation (process). Their subcomponents exist as local pseudo-objects (data structures) whose names change each time the method is called. How odd; why tie the lifetime of a GUI to a method activation, instead of making it out of ordinary objects (directories) that the user can dispose of at will?

Of course, in actual fact Unix was never trying to be Smalltix. Nevertheless, each of these incredulous "alien" observations can be *reversed* to suggest a new way of programming applications in Unix, one which brings the best of both the Unix and Smalltalk worlds. Or at least, this is the ideal — there is still the pesky matter of *performance* standing in the way.

11 A Small Matter of Overhead

A large part of the reason Unix processes are not as tiny and numerous as Smalltalk methods is that they weigh a lot. The Smalltalk VM is specifically designed to efficiently run many small methods, but Unix was not optimised for a similar granularity of processes.

A Unix process is a virtualisation of the machine state: it consumes page tables for virtual memory mapping, CPU registers, an integer number of memory pages (even *one* of which may be much larger than the compiled code of a method), metadata (environment variables, command line, file handles), and kernel data structures. Spawning a new process involves a delay to set up these structures, and context switches must wait to save and restore some of them. There is also a delay for relocating the code of any shared modules required to be present in the virtual address space. The result is that processes are "large objects" (§4.2); I expect that bad things will happen if I try and create a million of them, and my coworkers will glare at me for programming in such an odd way. In contrast, Smalltalk methods may occupy less space than a page and carry only a small amount of metadata.

The story for files is more convenient. The great thing about files, for realising Smalltix, is that their implementation can be overridden via FUSE (Filesystem in User Space). This means that the objects-as-directories (and files-as-primitives like object paths and tagged integers) could be made as efficient as the Smalltalk implementation of objects via an appropriate FUSE mount; this resembles the Plan 9 approach also discussed by Kell [Kell 2018b, §6]. Effectively, we expose the compact and specialised object format of the Smalltalk VM to the filesystem and get the best of both worlds. There is the subtlety that the image file is not the same as the object graph in the running VM process, although they are similar. Effectively, key pieces of the VM (such as the memory manager with its garbage collector) would need to be moved into the FUSE implementation.

Things don't look too bad even if we forego FUSE. Unlike the Smalltalk memory manager, a Unix filesystem does not maintain separate spaces with different default reservations for new files, or different garbage collection algorithms. It gets by with reference counting: the filesystem is treated as the set of roots, and a file's contents are marked as part of the free space when it has been fully unlinked. At the very least, there would need to be some scheme for cleaning up the many temporary objects created by processes. It could be as simple as mandating that each process deletes all files that it created once it terminates.

More interesting would be the use of a virtual RAM filesystem to exploit the ephemerality of process memory for these objects. Given such a scheme, it is unclear if garbage collection would even be necessary for Smalltix, provided that it had gigabytes of disk memory to use. I expect that the objects in a Smalltalk system which are not coming and going in method activations are long-lived and human-meaningful like files are expected to be. The only wrinkle is that one ought to be able to obtain a reference to a process-local file from the outside, at which point it must no longer only live in RAM but must also become backed by a "real" file on disk.

Still, no amount of clever techniques like these would allow the native Unix filesystem implementation to cope with a rapid turnover of small files. It is safe to assume that FUSE virtualisation would be necessary for a practical realisation of Smalltix. The bigger problem is that we would need something similar for processes: allow a single process to intercept and simulate process management syscalls. Sadly,

such a thing does not yet exist. What are the workarounds in the meantime?

I think that all workarounds must be variations of somehow funnelling all Smalltix activations through a *single* process. Perhaps each activation could be a thread dynamically injected into the process (although Unix threads are implemented as processes in the kernel, so it is unclear how much overhead this would save). However, it is not clear how to invoke methods or VM primitives in a similar way. In the Bash example, each call to `send` starts a new Bash process, and likewise for the methods that `send` will eventually call. If we hope to funnel all of these through some generic runner, named for example `invoke-vm-func`, then this will still create a new process all the same. Perhaps these problems stem from Bash; if we decide to write all our method files in Python, then a function like `eval()` could let us execute further methods without leaving the interpreter process. However, we would then have lost our language-agnosticism.

Perhaps the shell is the proper place in which to root the workaround: a version of Bash which, when executing files marked as Smalltix programs, takes advantage of their restricted protocol relative to arbitrary Unix programs and executes them in an optimised way. This would effectively be a command-line Smalltalk REPL leveraging the filesystem.

12 Next Steps and Future Work

In order to test the viability of the Smalltix Connection, I will implement the entire `SBECrossMorph` example and come up with some Unix analogue of Squeak's drawing canvas to see it render.[18] The reason I consider `SBECrossMorph` a good use case to work through is that its method call tree touches major programming areas (graphics, interaction, arithmetic, collection operators), which will challenge the concept better than a toy example (say, Hello World or Factorial). Preliminary tinkering with a web page `<canvas>` element backed by a local server shows promise; Smalltix drawing methods could send commands to the server, which could receive input events from the web page and invoke Smalltix event handlers.

I intend to bravely press ahead with the naïve model, retreating to the workarounds in the previous section only if I am forced to do so. Obviously I am expecting to have a hard time performance-wise beyond toy examples (to say nothing of spawning many processes to render a new frame), but it is more important to discover any *other* problems with the idea that were not so predictable. Will my optimism in §10 be vindicated? If it goes well, then a more advanced example from *Squeak By Example* or even a simple Smalltalk GUI could perhaps be ported.

In this essay, I have fled from in-process data structures and instead sought refuge in the filesystem. However, I am also interested in the liballocs project [Kell 2015], which seeks to bring some order to the fragmentation of process memory contents. One of its goals is to soften languages into merely different *views* onto the same runtime data, making programming a language-agnostic enterprise (as opposed to the *status quo* of committing to one language's ecosystem and libraries, and no-one else's).

Smalltix (in theory) achieves language agnosticism in a different way. A process running language A has no need to share its memory with a process running language B, because they communicate all structured data via the universal language of the filesystem.[19] Furthermore, processes are (supposed to be!) so small that there is never a need for multiple languages to coexist in the same process. "Languages" are views onto private local data, and all languages have file APIs. However, liballocs' approach is more general and is designed to accommodate existing practice; Smalltix is just a new way of doing things, and I am not sure how it should interface with normal Unix applications.

It is possible that liballocs could pave an alternate path towards a less hermetically sealed Smalltalk in Unix, especially if combined with FUSE: the data structures of the running VM could be annotated, objects and methods could be identified, and a Smalltix filesystem interface could be provided to a live Squeak object memory. However, this does not address the lack of automatic persistence of process data, which I consider to be one of the main pain points of Unix contra Smalltalk.

I am enthusiastic about extending the ubiquitous filesystem interface to all "small data", Plan 9 style. In §7, I admitted an "unprincipled exception" to object-directory equivalence: Smalltalk methods are objects, but Unix executable files are not directories. The proposals of Files-as-Directories [Kell 2018a; Wimmer 2018] and Files-as-Filesystems [Greenberg 2021] show promise for patching this inelegance.

13 Conclusion

I have argued that some important comforts of the Smalltalk programming experience are within reach of non-Smalltalk programmers, in principle, though not yet in a realistically performant way. By treating the Unix executable file as a simple renaming of the Smalltalk method, and (optionally) by treating the directory as a renaming of the Smalltalk object, we get persistence, uniformity, dynamic software updating, and GUI openness for free — without having to sequester ourselves in a VM process or image. Even better, we can continue to program in whatever language we like, as long as we can stomach a very unusual philosophy as to how applications should be architected. Given these promised benefits, I hope we can agree that it is worth *figuring out*

[18] See the repository at https://github.com/jdjakub/smalltix/.

[19] Kell [2018b, §5] warns that Smalltalk's "solution" to fragmentation is to just advocate itself: "don't fragment; use Smalltalk for everything!". I concede that this is echoed by Smalltix: "don't fragment; use the filesystem for everything!"

how to optimise the performance of such a model to be viable in practice.

Acknowledgments

I would like to thank Stephen Kell himself for the original work that inspired this paper, for corrections to the final manuscript, and for guidance on future optimisation possibilities. This work has been supported by the Charles University grant PRIMUS/24/SCI/021 and by the Czech Ministry of Education, Youth and Sports grant ERC-CZ LL2325.

References

Babiker Hussien Ahmed, Sai Peck Lee, Moon Ting Su, and Abubakar Zakari. Sept. 2020. "Dynamic software updating: a systematic mapping study." *IET Software*, 14, 5, (Sept. 2020), 468–481. DOI: 10.1049/iet-sen.2019.0201.

Gilad Bracha. 2022. *The Prospect of an Execution: The Hidden Objects Among Us*. https://gbracha.blogspot.com/2022/06/the-prospect-of-execution-hidden.html.

George Copeland and David Maier. June 1984. "Making smalltalk a database system." *SIGMOD Rec.*, 14, 2, (June 1984), 316–325. DOI: 10.1145/971697.602300.

Michael Greenberg. 2021. "Files-as-Filesystems for POSIX Shell Data Processing." In: *Proceedings of the 11th Workshop on Programming Languages and Operating Systems* (PLOS '21). Association for Computing Machinery, Virtual Event, Germany, 17–23. ISBN: 9781450387071. DOI: 10.1145/3477113.3487265.

Michael Hicks and Scott Nettles. Nov. 2005. "Dynamic software updating." *ACM Trans. Program. Lang. Syst.*, 27, 6, (Nov. 2005), 1049–1096. DOI: 10.1145/1108970.1108971.

Joel Jakubovic, Jonathan Edwards, and Tomas Petricek. 2023. "Technical Dimensions of Programming Systems." *The Art, Science, and Engineering of Programming*, 7, 3. DOI: https://doi.org/10.22152/programming-journal.org/2023/7/13.

Alan Kay. 1997. *The Computer Revolution has not Happened Yet*. Around 41 minutes into the talk. https://www.youtube.com/watch?v=oKg1hTOQXoY&t=2460s.

Stephen Kell. 2018a. "Critique of 'files as directories: some thoughts on accessing structured data within files'." In: *Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming*, 175–179.

Stephen Kell. 2015. "Towards a dynamic object model within Unix processes." In: *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* (Onward! 2015). Association for Computing Machinery, Pittsburgh, PA, USA, 224–239. ISBN: 9781450336888. DOI: 10.1145/2814228.2814238.

Stephen Kell. 2018b. "Unix, Plan 9 and the Lurking Smalltalk." *Reflections on Programming Systems: Historical and Philosophical Aspects*, 189–213. DOI: https://doi.org/10.1007/978-3-319-97226-8_6.

Clayton Lewis. 2018. "Phenotropic Programming?" In: *PPIG*.

Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. June 2006. "Practical dynamic software updating for C." *SIGPLAN Not.*, 41, 6, (June 2006), 72–83. DOI: 10.1145/1133255.1133991.

Ian Piumarta and Alessandro Warth. 2008. "Open, Extensible Object Models." In: *Self-Sustaining Systems*. Ed. by Robert Hirschfeld and Kim Rose. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–30. ISBN: 978-3-540-89275-5. https://tinlizzie.org/VPRIPapers/tr2006003a_objmod.pdf.

Uday Reddy. 1988. "Objects as closures: abstract semantics of object-oriented languages." In: (LFP '88). Association for Computing Machinery, Snowbird, Utah, USA, 289–297. ISBN: 089791273X. DOI: 10.1145/62678.62721.

Patrick Rein and Christoph Thiede. 2023. *Squeak By Example*. Software Architecture Group, Hasso Plattner Institute, Germany.

Filip Strömbäck and Dániel Varró. 2024. "Active DSU: Dynamic Software Updates for Active Functions." In: *Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Onward! '24). Association for Computing Machinery, Pasadena, CA, USA, 26–37. ISBN: 9798400712159. DOI: 10.1145/3689492.3690046.

Raphael Wimmer. 2018. "Files as directories: some thoughts on accessing structured data within files." In: *Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming* (Programming '18). Association for Computing Machinery, Nice, France, 166–170. ISBN: 9781450355131. DOI: 10.1145/3191697.3214323.

Received 2025-04-24; accepted 2025-08-11