Convivial Design Heuristics for Software Systems

Stephen Kell
University of Kent, UK
S.R.Kell@kent.ac.uk

ABSTRACT

Illich's notion of conviviality centres on the balance between individual freedom to act and collective freedom from domination. This balance, or tension, is present in the design of most user-facing computer systems, and especially in the design of programming systems. Software lore has arisen with at best a skewed perspective on such issues, having developed from an industrial viewpoint.

In this paper I survey some tentative design principles, extracted from examples of research work or (more often) systems used in practice, which (sometimes by accident) do show some regard for conviviality. Although preliminary, my hope is that these principles may yet develop into a collection of design hints at least equal, and largely countervailing, to the less conviviality-prone ideas circulating in today's software folklore. Relevant topics include language design, information hiding, language virtual machines, portability, classical logic, and layered system design. I also briefly consider the intertwined social and political constructs, such as copyleft, ownership and community responsibility, asking how to evolve or generalise these towards convivial ends.

CCS CONCEPTS

• **Software and its engineering**; • **Social and professional issues**;

KEYWORDS

Programming, design, modularity, culture, conviviality

ACM Reference Format:
Stephen Kell. 2020. Convivial Design Heuristics for Software Systems. In *Companion Proceedings of the 4th International Conference on the Art, Science, and Engineering of Programming (<Programming'20> Companion), March 23–26, 2020, Porto, Portugal*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3397537.3397543

1 TECHNICAL HEURISTICS

Jareth I ask for so little.
Just let me rule you, and you can have everything that you want.
Just fear me, love me, do as I say,
and I will be your slave.
Sarah You have no power over me.

from the film 'Labyrinth' [11]

Illich's notion of conviviality [10] centres on a balance: between individual freedom to act and collective freedom from domination. A tool that allows some individuals to achieve great power is unacceptable if it tends towards domination of the majority—even if this domination occurs unintentionally, indirectly or creepingly.

For Illich, writing in the 1970s, this meant several things. Institutionalised education is unacceptable because it creates a society dominated by the social capital of the 'most treated', rather than one of individuals empowered by learning. The extremes of modern medicine are unacceptable because they prop up the continued suffering of much of the population, amid the unhealthy environment of industrialised society. A society organised around high-speed transport is unacceptable because the resulting social imperative, or 'radical monopoly' of high-speed travel forcibly consumes the time, space and energy of anyone participating in that society, while accruing net benefit only to a wealthy few.

For us, concerned with computer systems in the present day, these ideas are no less relevant. Domination is rooted deeply in the culture of computer systems. 'Systems' dominate 'users', while 'languages' and 'tools' dominate 'programmers'. As we push against the frontier of absolute capability of our systems, and increase the capability of their most expert and advanced teams of users, we tend to shrink both the relative *and* absolute capabilities of the 'unit individual'. I assert this without proof, but believe it is a phenomenon increasingly observed and accepted. It is familiar, for example, to the generation who grew up programming early microcomputers and nowadays wrestle to achieve comparably simple feats using a 'modern' web framework. Although the modern technology is more sophisticated in many ways, this rarely translates to less work being needed to accomplish a simple human-meaningful task.

To reconsider *programming systems* with the goal of conviviality, not industrial notions of productiveness, what must change about our thinking? We must firstly stop the endless extrapolation behind industrial thinking, including its 'economies of scale'—for the same reason that in the long run we are all dead. Instead of seeing 'programming' in idealised form as the act of an individual, we must frame it scaled up to whole economies or ecosystems of people writing software to diverse ends. They all do so using existing programming systems, whose designs are informed by a small number of decades' thinking, under the strong influence of industrial ideas. The following countervailing 'design heuristics' are the (unfinished) product of thought experiments in rejecting industrial notions of 'larger problems' and instead considering the effects our technical choices as they are scaled up along human dimensions, through cultures and societies.

Value *reference* over *definition*. When we ask what a programming language can *express*, we mean what can be defined within it, not what things outside itself it can reference. So begins that language's play for domination. To achieve conviviality, this must be reversed. The more a system can draw from its context of

<Programming'20> Companion, March 23–26, 2020, Porto, Portugal

ACM ISBN 978-1-4503-7507-8/20/03…$15.00
https://doi.org/10.1145/3397537.3397543

use, the less it dominates its users. The 'polymorphic identifiers' of Weiher and Hirschfeld [24] are a recent rare example of work founded on expanding the space of reference; one can credit F#'s type providers [19] with their ability to 'bring in' diverse sources of data from outside. The same thinking is found in the systems and networking tradition of of unifying namespaces, such as the filesystem in Unix, or the space of network interfaces in IP. In each case, the easier it is for a 'mere programmer' or 'mere user' to extend the space of names, the better; sadly, even in most of the systems mentioned, this remains an involved process.

Value *linking* over *containment*. This is the same idea restated. What we contain, we control. By contrast, linking a (partial) program to an external definition requires some kind of negotiation across a boundary. This might be by consensus, when the link is 'direct', meaning referer and referent are well-matched syntactically and semantically. Or it might be by mediation, where some adaptation layer intercedes on the interaction. This distinction ought to be as essential as that between synchronous and asynchronous communication, yet it remains a non-feature of most programming systems. I wrote previously about how at the level of module interfaces, mediation is a separable problem which is often simpler than general-purpose programming [13]. It is no coincidence that graph-theoretic notions of *dominance* generalize containment ('all paths go through me'), nor that the trend in computing use cases (but often not technologies' designs!) is for hierarchy to give way to heterarchy—as we gain hardware capable of transcending fixed spaces and fixed communication structures.[1]

Beware fragile affordances. Mention of 'linking' calls to mind the web. The web appears to embrace linking, but in practice its linkable design is easily trampled on. Computationally sophisticated web pages are usually *not* functionally linkable in such a way that would allow, for example, one to usefully reference their contents 'from the outside' or cause their internals to be re-bound to alternative referents elsewhere. Imagine, for example, a hyperlink that functioned as a 'continuation' that would allow a friend to resume at the same page; this falls within the idealised promise of the web, but is seldom found in reality. Although linking is possible, the capability is fragile, and the 'design fabric' of the web does little or nothing to preserve it. Indeed one could argue that the separateness of design elements such as cookies, referers and 'post data' effectively *discourages* linkability. Meanwhile, the module boundaries and dynamic (re)binding that would make for useful 'linking' are also often absent. Consider how many web applications *contain* their own embedded 'rich text' editing widget. If linking were truly at the heart of the web's design, a user (not just a developer) could supply their own preferred editor easily, but such a feat is almost always impossible. A convivial system should not contain multitudes; it should permit linking to them.

Support referential structures, not [just] definitional structures. What else might it mean to be 'expressive' with respect to the outside? Infrastructure should make it easy to 'construct views', that is specifying new ways to refer to already extant definitions. Relational databases have succeeded in this like few others. In networks, despite the goal of evolutionary application deployment that lies behind the Internet's end-to-end design, IPv4's possessiveness of the network namespace—'the only addressable entities are IPv4 interfaces'—defied even contemporary thinking on how to do naming, and continues to stymie evolution of IP in deployment (specifically, to version 6 [6]).

Forget world domination. Implementers of new languages sometimes speak of a 'libraries' problem. What they mean is that their language cannot adequately interface with existing code, and to compound the problem, it has also not yet achieved world domination—meaning thousands of developers have not yet been compelled to spend thousands of hours, on either reimplementing that existing code, or writing and maintaining shims or wrappers. This is not a problem that can or should be 'solved'. Rather, the problem is in the assumed necessity of bootstrapping a new ecosystem—the hope of world domination. Such hopes must be dashed for the good of all. If we follow the other principles mentioned here, perhaps such domination will not be necessary.

Bring the outside in. In older languages such as Pascal, C, Fortran and so on, external entities can be declared, often with a suitable type annotation and/or calling convention, and thereafter used much like a 'native' object. However, in more 'modern' garbage-collected languages, this kind of *referencing the outside* has been deemed intractable. The art of language implementation has turned inwards—on an entanglement of compilers and managed heaps, a fully circumscribed universe. Only if we untangle these and turn a language's attention *outside*, liberating 'memory' or 'objects' or 'environments' or 'bindings' as a *space* within which diverse entities may be *referenced*, will we cease seeking to dominate. This embraces, but goes beyond, the question of what we can name or reference; it is about the primacy of communication, over bounded (fenced-in) computation.

Do not create worlds. At or near the origin of the universe, creating new worlds is a necessary activity. But when existing matter is plentiful, it is either a futile gesture or a bid for domination. A key idea behind subject-oriented programming [8]. was that creating a new *taxonomy* need not force creation of a new world. There is an open question of how to achieve this kind of subjectivity in the large, and among existing systems. Much as database-style 'views' seem a useful abstraction, more generally some degree of mapping or 'gatewaying' of *spaces* also appears essential, since a single unifying space will never emerge by consensus. (Ironically, gatewaying is what the Internet Protocol sought to eliminate; but, as we noted, its approach was only an option for first-movers, and was itself based on domination within the network layer.)

All objects should appear in all spaces. To continue the idea of gatewayed spaces: if new worlds cannot be created, how can we create a new reality? The answer is to create a new space—but one that maps the old world within it. This notion of 'space' is not an established concept in programming systems design. Although difficult to define, spaces are fairly easy to recognise. Any 'namespace', 'environment' or even denotative 'language' is likely to be a space in a sufficiently close sense. In the land of Unix, VFS [7] has been used to broaden the filesystem namespace to other ends, but users are not afforded an easy (shell-level) means to define new

[1] In the late 2000s I read a text which first introduced to me the notion of 'heterarchy' in computer systems observing its tendency to replace hierarchy. I was sure it was a draft by Robin Milner of his book about bigraphs [15], but this is a false memory: neither the published book nor the draft dated January 2008 uses the word. If any reader knows where I might have read this observation, I would be grateful for a reference!

filesystem views; instead this remains an expert task ('write a `fuse` [22] file server'). The 'files as directories' concept of Wimmer [25] is yet another articulation of this pattern's value. Slightly more laterally, the central design of the Dynamicland laboratory in Oakland, as articulated in a talk by Victor [23] aptly entitled 'Seeing Spaces', is also an instance of this pattern: programmatic objects are also physical objects that appear (or 'are seen') in the user's spatial realm, while conversely, ideally a large fraction of physical objects in that realm are programmatically available.

Say no to classical logic. Ostermann, Giarrusso, Kästner, and Rendel [17] wrote about the link between classical logic and conventional notions of modularity. Classical logics have the property that new facts cannot invalidate old inferences. When applied to modularity, such approaches bring scalability, because locally established facts (such as the correctness of a client with respect to a module specification) are never jeopardised by changes elsewhere (such as varying the implementation within the module). However, *facts do change*. Programmers change interfaces constantly; on what happens then, information hiding has nothing to say (except that some past programmer made a mistaken prediction of the future). Similarly, although classical inferences are robust to new axioms, if a prior axiom must change, all bets are off. The promises of stability and scalability therefore rest rather precariously on the presence of inviolable 'axioms' or specifications that, in the case of software, are defined by fallible humans. A far more realistic model is one where new facts may indeed disrupt old inferences. Cook [5] observed that abstract data types gain their amenability to reasoning from the way in which a given type's representation, although existentially hidden, remains a singular fact (hence allowing class-style encapsulation). By contrast, objects are hard to reason about because they radically delay the fixing of their implementation details, this limiting them, at least in their purest form, to per-object encapsulation. But this same intractability is also the source of flexibility and diversity, for example in permitting many *interoperable* implementations of the same abstraction (a theme discussed by Aldrich [1]). The concepts of 'closed-world' and 'open-world' reasoning are relevant here, although slippery: classical modularity's monotonicity permits nominally 'open-world' reasoning, and indeed 'for all time'—but only thanks to fixed axioms which constrain the domain of the program (to a single implementation of a given abstraction, say). Non-classical logics, by admitting transient inferences such as those produced by closed-world reasoning, actually keep that domain open (to many new and coexisting implementations of a given abstraction, say).

Implement porously, not portably. The best portable specifications provide unifying 'views' onto a multitude of external definitions. But in contrast, a portable *implementation* is often problematic, created as a 'needs must' approach to mitigating external diversity. In order to avoid linking with such diverse outside entities, it tends to contain fresh re-implementation addressing only a fixed 'lowest common denominator' view of the outside. In the late 1990s and early 2000s, Java's 'Swing versus AWT versus SWT' design discourse flowed back and forth over this issue [16]. More recently, a bug report on the infamous systemd project shows a more trivial but no less real instance, over whether system usernames were allowed to contain digits.[2] The verdict: 'in order to make `systemd` unit files portable between systems, we'll hence enforce something that resembles more the universally accepted set, rather than accept the most liberal set possible'. The chosen trade is to 'level down' outside the circle (reduce the space of valid usernames), in order to level up within it (increase the core system's portability, unmodified, to other platforms). This is locally expedient, but is globally the non-convivial option: the fallout is on the hapless user whose system already contains these now-illegal usernames. A portable implementation tends to dominate its users. To level up rather than down, the system needs to take a less portable or at least more diverse view of the outside.

Abstraction definition as a costly operation. One can quibble that the preceding example is some unclear mix of the cultural and the technical. On one level it reveals simply that the codebase is *insufficiently polymorphic*. In this example, a polymorphic solution would somehow be parameterised over the domain of usernames supported by the host system. Coding it in such a manner may or may not be convenient, but would certainly be possible, at the price of adding a further layer of indirection or abstraction (that which is hidden behind the polymorphic interface). The choice not to offer this, one the other hand, might be blamed on the culture of infrastructure software, which likes to define *itself* as the authority on the world around it, by specifying a 'unifying-or-else' abstraction that, at least in this example, is not actually very unifying. The definition of new abstractions is often perceived as a 'free' operation, or at least an author's prerogative, but it is actually a high-cost operation precisely because it is a play at dominating others. Although technical affordances, such as polymorphic coding or other indirections, can reduce the need for such dominating moves, they almost always do so with additional human or technical cost, meaning that (as with the systemd example) programmers prefer not to use such means unless or until they are deemed 'needed'. This judgement depends on the dominance of one's position.

Reasoning scales best when it's small. Limiting the use of abstraction and preferring non-monotonic reasoning would seem to put some dampeners on our ability to reason about large systems. Yet I certainly would not argue in favour of unreliable systems. How do we reclaim this goal? The 'obvious' answer is to keep our systems small. Formal verification research has suffered from a phenomenon that could be called the 'Zeno-Wirth paradox': by the time verification can scale to a given extent, commodity systems have become that much less lean [26] that it no longr suffices. Unlike Zeno's case, the former's speed need not outstrip the latter's. Meanwhile, as automated reasoning engines become more powerful, the lengthening 'chain of trust' in their soundness becomes a significant problem [14]. Faith in a long bootstrapping chain is an industrial idea: industrial development has proceeded via the bootstrapping of powerful machinery on which ever-larger populations depend ever more critically. We need not to discard this, but to rein it in. Unlike Wirth, we are not limited to pleading or to developing our own 'lean' systems. Commodity systems of even ten years ago are 'small' compared with current ones, and the pile-on of complexity in mainstream systems appears to be yielding diminishing returns. Being optimistic for a second, this may be reason to believe that

[2] https://github.com/systemd/systemd/issues/6237

manageably-sized systems can be made lean enough to be reasoned about, yet also compatible enough to compose usefully with the mainstream.

Information guiding, not hiding. If defining new interfaces is high-cost, then where does that leave perhaps the most time-honoured design heuristic, information hiding in the sense of Parnas [18]? Such a technique requires predicting what is likely to change, so that we know what to hide. These predictions may be correct in the short term, but they have a half-life and can always be wrong. In a world where software is not developed in 'closed project' mode, but may live on in surprising ways, hard-boundaried hiding is a recipe for disposability. Clark and Basman [4] have already made this case in more detail. I also wrote about the relative virtues of language features for 'guidance' as distinct from enforced abstraction in the context of type-checking [12].

2 SOCIAL HEURISTICS

Master Yoda The dark side of the Force are they.
Easily they flow, quick to join you in a fight.
If once you start down the dark path,
forever will it dominate your destiny.
Luke Is the Dark Side stronger?
Master Yoda No, no, no. Quicker, easier, more seductive.

from the film 'The Empire Strikes Back' [3]

Yoda was speaking of anger, fear and aggression. But perhaps, given opportunity, he would also have mentioned certain anti-convivial traits mentioned above, such as hierarchical containment, world creation, and over-reliance on classical logic. These are also 'quick to join' in the fight of building software; yet they dominate our destiny in adverse ways. The light side may ultimately be stronger, but how can we prevent the fall of the galaxy?

'Conviviality' as both social and technical. The free software movement, particularly its GNUist, copyleft-advocating section, takes an approach to licensing that could almost be quoting Illich. Illich wrote that 'a convivial society would be the result of social arrangements that guarantee for each member the most ample and free access to the tools of the community and limit this freedom only in favor of another member's equal freedom'. Stallman has described the GNU General Publice Licence as 'fighting fire with fire' [21], by using copyright law to protect freedoms rather than curtail them. If free software has failed, perhaps that is partly from the its relative lack of technical differentiation; Pike lamented twenty years ago [20] how the designs of free software components have rarely innovated much over their commercial competitors and predecessors. A further weakness of the movement is that many free software projects have proven amenable to corporate capture, and not coincidentally, this has most often occurred in those projects where a huge industrial-strength team is required simply to tread water. Therefore, perhaps we should start to recognise software projects not only on their quality or completeness, but on their tractability to individual contributors and customisers. This tractability is likely to be *inversely* correlated with the rate of code churn and size of community—two metrics commonly associated, sometimes perversely, with the health of a project. Obviously, such an inverse association only appears at some point after these metrics have crossed beyond some lower threshold of basic viability. This 'two thresholds' model is familiar. Illich wrote of medicine passing two thresholds: a first where medical technology delivered verifiable benefit to society, and a second where further developments increasingly delivered overall harm, but where this harm had come to be defined as benefit, owing to the use of metrics that discipline of medicine had itself created.

Exploiting irrationality. Illichian thinking is challenged by the tragedy of the commons. Once a powerful tool exists in a society (such as cars), those within reach will use it (to drive), and this easily pushes society down a slippery slope (to increasing networks of roads, sprawl, and quasi-compulsory driving). Although perhaps kicked off by some degree of political or institutional enablement (such as building roads), beyond a certain point this tragedy arises from selfish rational agents descending a gradient (that of 'anti-utility', in economists' terms). How can we steer our culture towards the 'light side', so that it may be healthy and well-functioning even in the presence of such dynamics? Humans also have wonderful capacity for irrationality, and convivial culture must surely harness this. Humanity's stable 'common' institutions and restraints can be traced perhaps to evolved ethical or even religious tendencies, which instil a sense of moral culpability overruling what are locally 'selfish, rational' choices in favour of collective interests. Kant observed the moral value of 'institutions', as basic as truth-telling, preserved by acts of restraint that may frequently be irrational to a selfish individual, but which human society has shown some capacity to preserve. The battle for convivial software in this sense appears similar to other modern struggles, such as the battle to avert climate disaster. Relying on local, individual rationality alone is a losing game: humans lack the collective consciousness that collective rationality would imply, and much human activity happens as the default result of 'normal behaviour'. To shift this means to shift what is normal. Local incentives will play their part, but social doctrines, whether relatively transactional notions such as intergenerational contract, or quasi-spiritual notions of our evolved bond with the planet, also seem essential if there is to be hope of steering humans away from collective destruction.

Exploit the desire to own and to simplify. The normalization of overpowerful tools challenges humans' capacity for collective restraint. Like pre-industrial humanity, computer science has evolved in a mindset of scarcity—making it vulnerable to cravings for the apparent fruits of industrial over-efficiency. In an era of plenty, these cravings become unhealthy and even self-destructive. Yet humans also routinely strive to eliminate the 'negative' from their habits, environment or diet, and often are limited as much by limits of understanding than by limits of effort or discipline. Basman [2] wrote about the 'ownability of software', or its absence; the psychological appeal of ownership is something which, if effectively harnessed, would provide impetus for the smaller, more individually tractable and more convivial.

Exploit norms of social responsibility. Hinsen [9] has written of the 'moral commitments' that impinge on even a community-owned project such as Python, and how the traditional tenets of open-source have failed to recognise them. These are responsibilities that come from occupying a 'monopoly' or otherwise 'dominant' position, and where 'if you don't like it, you're free to fork the code' is an open abdication of those responsibilities. Yet communities are great at spawning codes, tenets, and taboos, however

indiscriminately. If user-disrespecting or anti-convivial designs were made taboo, healthier tendencies for the project might easily result. Linus Torvalds's strict code of 'not breaking user-space' is usefully perpetuated by such a taboo, even if one rather unpleasantly established through the profane excoriation of those who well-intentionedly submit non-compliant patches.

To conclude, it bears repeating that many of the supposedly technical principles of programming are as much a cultural matter as a technical one. I hope the foregoing thoughts can become part of a broader exploration of how to make computer systems work for humanity rather than vice-versa.

REFERENCES

[1] Jonathan Aldrich. 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* (Onward! 2013). ACM, Indianapolis, Indiana, USA, 101–116. ISBN: 978-1-4503-2472-4. DOI: 10.1145/2509578.2514738. http://doi.acm.org/10.1145/2509578.2514738.

[2] Antranig Basman. 2016. Building software is not (yet) a craft. In *Proceedings of the 27th Annual Conference of the Psychology of Programming Interest Group (PPIG)*.

[3] Leigh Brackett and Lawrence Kasdan. 1980. The Empire Strikes Back. Screenplay to a film distributed by 20th Century Fox. (1980).

[4] Colin Clark and Antranig Basman. 2017. Tracing a paradigm for externalization: avatars and the gpii nexus. In *Companion to the First International Conference on the Art, Science and Engineering of Programming* (Programming 201917). Association for Computing Machinery, Brussels, Belgium. ISBN: 9781450348362. DOI: 10.1145/3079368.3079410. https://doi.org/10.1145/3079368.3079410.

[5] William R. Cook. 2009. On understanding data abstraction, revisited. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (OOPSLA '09). ACM, Orlando, Florida, USA, 557–572. ISBN: 978-1-60558-766-0. DOI: 10.1145/1640089.1640133. http://doi.acm.org/10.1145/1640089.1640133.

[6] S. Deering and R. Hinden. 1995. Internet Protocol, Version 6 (IPv6) Specification. IETF Request for Comments. (1995).

[7] Robert A. Gingell, Joseph P. Moran, and William A. Shannon. 1987. Virtual memory architecture in SunOS. In *Proceedings of the USENIX Summer Conference*. USENIX Association, 81–94.

[8] W Harrison and H Ossher. 1993. Subject-oriented programming: a critique of pure objects. *ACM SIGPLAN Notices*, 28, 411–428.

[9] Konrad Hinsen. 2020. The rise of community-owned monopolies. Blog article. Retrieved on 2020/5/7. (2020). https://blog.khinsen.net/posts/2020/02/26/the-rise-of-community-owned-monopolies/.

[10] Ivan Illich. 1973. *Tools for Conviviality*. Harper & Row.

[11] Terry Jones. 1986. Labyrinth. Screenplay to a film distributed by TriStar Pictures. (1986).

[12] Stephen Kell. 2014. In search of types. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Onward! 2014). ACM, Portland, Oregon, USA, 227–241. ISBN: 978-1-4503-3210-1. DOI: 10.1145/2661136.2661154. http://doi.acm.org/10.1145/2661136.2661154.

[13] Stephen Kell. 2009. The mythical matched modules: overcoming the tyranny of inflexible software construction. In *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA 2009). ACM, Orlando, Florida, USA, 881–888. DOI: 10.1145/1639950.1640051. http://doi.acm.org/10.1145/1639950.1640051.

[14] Ramana Kumar. 2016. Self-compilation and self-verification. Tech. rep. UCAM-CL-TR-879. University of Cambridge, Computer Laboratory, (Feb. 2016). https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-879.pdf.

[15] AJRG Milner. 2009. *The Space and Motion of Communicating Agents*. Cambridge University Press.

[16] Ella Morton. 2005. James Gosling Q&A. *Builder AU*. Web article. Archived at https://web.archive.org/web/20051210233810/http://www.builderau.com.au/program/work/0,39024650,39176462,00.htm.

[17] Klaus Ostermann, Paolo G. Giarrusso, Christian Kästner, and Tillmann Rendel. 2011. Revisiting information hiding: reflections on classical and nonclassical modularity. In *Proceedings of the 25th European Conference on Object-oriented Programming* (ECOOP'11). Springer-Verlag, Lancaster, UK, 155–178. ISBN: 978-3-642-22654-0. http://dl.acm.org/citation.cfm?id=2032497.2032509.

[18] David L. Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15, 1053–1058.

[19] Tomas Petricek, Gustavo Guerra, and Don Syme. 2016. Types from data: making structured data first-class citizens in f#. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '16). Association for Computing Machinery, Santa Barbara, CA, USA. ISBN: 9781450342612. DOI: 10.1145/2908080.2908115. https://doi.org/10.1145/2908080.2908115.

[20] Rob Pike. 2000. Systems software research is irrelevant. Presentation. Slide contents available at http://doc.cat-v.org/bell_labs/utah2000/utah2000.html as retrieved on 2020/5/7. (2000).

[21] Richard M. Stallman. 1998. The X Window System trap. Web note, available at https://www.gnu.org/philosophy/x.html as retrieved on 2020/5/19. (1998).

[22] [n. d.] `fuse`—filesystem in userspace (fuse) device. Linux Programmer's Manual page. Available at http://man7.org/linux/man-pages/man4/fuse.4.html as retrieved on 2020/5/7. ().

[23] Bret Victor. 2014. Seeing spaces. Presentation. Video and comic available at http://worrydream.com/SeeingSpaces/ as retrieved on 2020/5/7. (2014).

[24] Marcel Weiher and Robert Hirschfeld. 2013. Polymorphic identifiers: uniform resource access in Objective-Smalltalk. In *Proceedings of the 9th Symposium on Dynamic Languages* (DLS '13). Association for Computing Machinery, Indianapolis, Indiana, USA, 61–72. ISBN: 9781450324335. DOI: 10.1145/2508168.2508169. https://doi.org/10.1145/2508168.2508169.

[25] Raphaël Wimmer. 2018. Files as directories: some thoughts on accessing structured data within files. In *Companion to the Second International Conference on the Art, Science and Engineering of Programming* (Programming '18). ACM, Nice, France.

[26] Niklaus Wirth. 1995. A plea for lean software. *Computer*, 28, 2, (Feb. 1995), 64–68. ISSN: 0018-9162. DOI: 10.1109/2.348001. https://doi.org/10.1109/2.348001.