Relentless Repairability or Reckless Reuse

Whether or Not to Rebuild a Concern with Your Familiar Tools and Materials

Marcel Taeumel
marcel.taeumel@hpi.uni-potsdam.de
Hasso Plattner Institute
Potsdam, Germany
University of Potsdam
Potsdam, Germany

Robert Hirschfeld
robert.hirschfeld@uni-potsdam.de
Hasso Plattner Institute
Potsdam, Germany
University of Potsdam
Potsdam, Germany

Abstract

We must retain liveness and exploratory practices within the programming systems that make us feel most productive. However, the temptation to just reuse *black boxes* through limited interfaces is pervasive. We *expect* time savings and better performance at the cost of *poor repairability*. Fortunately, we also know about the benefits of having an open implementation constructed from familiar materials, integrated with familiar tools. Consequently, it is primarily a matter of "just building it" ... again? Piece of cake. What could possibly go wrong?

CCS Concepts: • **Software and its engineering → Software testing and debugging**; *Object oriented languages*; *Integrated and visual development environments*; Runtime environments.

Keywords: self-sustaining systems, exploratory programming, liveness, reuse, open source, direct manipulation, symbolic debugging, code simulation, flow, Smalltalk, Squeak

ACM Reference Format:
Marcel Taeumel and Robert Hirschfeld. 2022. Relentless Repairability or Reckless Reuse: Whether or Not to Rebuild a Concern with Your Familiar Tools and Materials. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '22), December 8–10, 2022, Auckland, New Zealand*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3563835.3568733

1 The Dilemma

You are a programmer. Your program requires a certain feature. Now, you are torn between implementing something from scratch or reusing an existing piece from *somewhere*. The ecosystem around the programming language of your choice might offer multiple alternatives, ready to be imported. If not, there is typically C below everything, and you can pick a library from that space, integrate it with a tiny adapter. Your operating system might already ship those libraries, which saves you some deployment troubles. So, why bother implementing something on your own? It takes time; you will probably make mistakes along the way; you are not (yet) an expert in that feature's domain anyway.

What is it that your program requires? A new parser? There are many document formats, mostly with human-readable specifications. You might want to be able to parse JSON or XML. Files or web requests might carry images (e.g., png, gif, svg), archives (e.g., tar, zip, rar), rich text (e.g., pdf, docx, html), sound (e.g., mid, mp3, wav), or even unfamiliar source code (e.g., JavaScript, Python, C/C++).

Or do you need to figure out a new *cross-platform* specification? Maybe your program should offer access to input devices, displays, file systems, and networks on all major platforms such as Microsoft Windows, Apple macOS, and Ubuntu Linux. Surely, somebody has already done that.

Or do you want to address international users with your program? Parsing font files might be manageable, but actual text shaping and compositing is quite challenging with its multitude of language- or country-specific rules. A glyph is a glyph but not necessarily a valid letter or syllable.

Or do you need to establish a secure and reliable communication space in a distributed network? Starting with secure hash algorithms and UUID generation, you might need several protocols and services across the ISO/OSI session, presentation, and application layers (e.g., TLS, HTTP(S), SSH, SMTP, WebDAV, XMPP).

Or does your program benefit from local, yet specific, hardware? Accelerated graphics output might improve the user experience in your game or data visualization. High-quality sound processing might be part of the must-haves to satisfy your target audience. Computation-intensive tasks might benefit from using all the cores you have in your processor.

Or do you bother with storage and persistence? There are all kinds of databases for version control, data logging, or application-state checkpointing. Multi-user access is typically provided through a centralized client-server architecture. There are index, relational, object, or graph databases. You might want to implement your own client or also (extend) the server side.

Onward! '22, December 8–10, 2022, Auckland, New Zealand

ACM ISBN 978-1-4503-9909-8/22/12.
https://doi.org/10.1145/3563835.3568733

Or do you need any other sophisticated algorithm and control/data flow? On the one hand, there are engines and frameworks that can support any "business" logic for your application. On the other hand, many best practices and architectural patterns have also been documented, waiting for you to be followed in your own implementation.

[|]

We believe that it is desirable to adhere to the *values and practices* that define a certain community and its ecosystem. This desire entails a *carefulness* that programmers should be aware of and enact when working with software artifacts. Reusing existing artifacts from other programmers, teams, or communities might break established practices or values. Incompatibility might arise at a technical and social level. For example, crossing the system boundary compromises tool integration due to hidden structure or unexpected semantics. Any extra *black box* with its *limited interface*[1] might impede exploration[2] and debugging, making it unnecessarily complicated to fix issues that might arise in an unforeseeable future. Discussions with that other community might be challenging if no common ground can be established. Incommensurability is real [7]. Thus, we might want to reimplement software artifacts not because we think that we are smarter than somebody else. Instead, we might want to avoid reuse in favor of repairability. We believe that *familiar tools and materials* can help: known programming languages, known paradigms, known vocabulary, known patterns, known exploratory practices. Any step out of this trusted realm might lead to conflict. *Can we avoid that conflict?*

We do not know how, we wish we did. In this essay, we try to summarize the dilemma programmers face every time they add new features to their programs. Similar to a pattern form [12], we know about the *problem context* and the *problem forces*. However, we are not able to resolve these forces into an actionable *solution*. Thus, programmers keep on gambling with the risk of choosing the wrong path time and again. The forces that represent a programmer's *fears of reimplementation* for repairability are as follows:

- ♦ Your efforts to rebuild a thing will not be acknowledged if it is already available as an external module.
- ♦ You are not an expert and cannot (yet) fully understand the given specification.
- ♦ You will make mistakes and not cover all the corner cases in your reimplementation. Will 80 percent be enough?

The forces that represent a programmer's *fears of reuse* from foreign communities are as follows:

- ♦ You must decide soon: talented people will not join your community if crucial components are not yet available.
- ♦ That "tiny" adapter to the external module turns out to be a lot of work.
- ♦ That external module has many dependencies that will also be part of your program's dependencies.

The forces that represent a programmer's *hopes for repairability* when reimplementing are as follows:

- ♥ You will follow familiar practices when rebuilding something. Unforeseen issues can be explored and debugged with familiar tools.
- ♥ You can omit unnecessary features from a specification when being in control of the implementation. It will be more concise and maintainable.
- ♥ There are already many useful things in your system, which can be used when implementing a new feature. You do not start "from scratch."

The forces that represent a programmer's *hopes for short-term gain* when reusing are as follows:

- ♥ That external module is almost ready to be used in your program. It is basically "for free."
- ♥ That external module is polished and optimized for performance. You probably cannot do better.
- ♥ Your program does not have *that many* dependencies yet. One more cannot hurt.

Our main goal is to preserve the *values and practices* of the programming community and system of our choice. We value openness [34], liveness [27], directness [11], malleability [18], and feedback [26] in our system. We value communication [2], respect [2], curiosity [8], and forgiveness[3] in our community. We practice *exploratory programming*,[4] where specifications unfold gradually through experimentation, where failure is not punished but encouraged for the sake of learning, where programs and systems keep running for ages. We typically work with Smalltalk systems [10] such as Squeak [14, 18, 34].[5] In this essay, we use the term *repairability* to cover all our values and practices, which arguably transcend into other communities and systems. We favor repairability, and thus occasional reimplementation efforts, over unreflected reuse. *We invite readers to reflect on what "repairability" means in their preferred working environment.*

[|]

[1] Providers can mitigate this issue with an *open implementation* [15], where multiple kinds of interfaces can serve different kinds of users.

[2] While programmers use tools to explore source code manually, there are automated analysis tools that require full access to code for meaningful results.

[3] The Extreme Programming method mentions *courage* [2] as a core value, which we complement with curiosity and forgiveness to not only encourage experimentation, but also deal with failure in an expedient way.

[4] We consider *exploration* a skill that is shaped by the explorer's mindset and available tools. That is, specific programming tools may or may not support certain exploratory practices [36]. The benefits of *exploratory programming* were first experienced by practitioners working with the original Smalltalk systems [29, 38].

[5] The Squeak/Smalltalk programming system, https://squeak.org/

This essay is a collection of anecdotal perspectives. From hereon, we untwist our main thread about repairability and reuse into six "fibers," each being a text related to this theme. We begin each text with a *personal statement* that reflects our mood and sets the scene. You can read the stories in any order. A brief summary for each one goes as follows:

Section 2 A short dialog between two programmers who share a desire for liveness and exploration but make quite different trade-offs.

Section 3 A brief introduction into the authors' favorite tools and materials, filled with personal experiences and unsolved challenges.

Section 4 A quick reflection on whether the object-oriented paradigm is actually a good fit for any kind of problem.

Section 5 A faint distress signal that calls out for more skilled programmers who want to live our values and follow our practices.

Section 6 A small report on the first author's experiences about making Squeak releases and the issue of tagging along a visible community.

Section 7 A fair assessment of how to improve Squeak's liveness to yield predictable feedback loops for arbitrary application domains.

We revisit this essay's dilemma in **Section 8** and raise even more questions for an unknown future.

2 Two Cultures, One Liveness

> *Let us not dwell in the past. There are new technologies and supportive companies. The Web is the future. Everybody is there. Impact first!*

Two programmers, *Harmony* and *Sage*, are experts and long-time enthusiasts in the field of exploratory programming. Both care for a different, yet similar, self-sustaining system, full of liveness and immediate feedback. It is lunch break, and they have a discussion about today's accomplishments. Harmony is *happy* and somewhat proud of what she achieved. Sage is rather *skeptical* after listening to her remarks. She cannot quite understand and hence share the satisfaction:

𝕊 : Hi, there. Anything new in Squeak this morning?
ℍ : Hello! Yes! I finished the refactoring and partial rewrite of the font-file parser.
𝕊 : Font rendering? Huh. I thought that there was already this plugin...
ℍ : Yeah, we have quite a few implementations at this point. I worked on the one in the base system. It did not age well over the years.
𝕊 : Why didn't you just delete that code and use that existing plugin?
ℍ : We don't want to ship more dependencies with the virtual machine. And now that we have a simple implementation in Smalltalk, users can actually explore and learn about the font format. And extend it.
𝕊 : Hmm... and you can render all Unicode codepoints and international texts?
ℍ : Not quite. I still have to work on the font stack and fallbacks. Also, text composition is still limited to Western fonts. Well, there is at least by-character line-break support for Japanese text. So...
𝕊 : ...still a long way to go, huh? In Lively, we have all these things working out of the box. Google takes care of it. A lot of people are working on that.
ℍ : But you cannot look into the implementation, right?
𝕊 : We don't need to. It just works. And it's free.
ℍ : What if a user does not have the Chrome browser installed?
𝕊 : Well, they can use other Web browsers. International text rendering should work there as well. And our system should perform good enough in other browsers...
ℍ : You only optimize for Chrome? A platform that you cannot control?
𝕊 : The community around web development is huge. I wouldn't bother.
ℍ : And Lively itself? What if you use a not yet standardized feature that Google's Chrome interprets differently than, say, Apple's Safari?
𝕊 : Yeah, we had that in the past. The community around Lively is not super-big. And you can install Chrome on macOS as well. Nothing to worry about.

Harmony takes a sip of soda and changes the subject:

ℍ : Ha. Okay. Well, besides fonts, I also discovered a rather interesting overhead in the code that handles mouse clicks in push buttons...
𝕊 : You have to write *that* kind of low-level code? Is there even time to program actually useful stuff?
ℍ : The base system *is* useful. People use it to develop all kinds of applications. Take our students, for example. They really appreciate a responsive system.
𝕊 : Ah, I forgot that Squeak's Morphic is at the breaking point of responsiveness... do you even get 60 frames per second?
ℍ : Sure. If you follow some simple practices, you can get sufficient rendering performance out of the system.
𝕊 : It's still based on BitBLT from the 70's, right? Hmm... what about those high-resolution displays? 4K? Are there still no accelerated graphics? Still CPU-bound?
ℍ : Yes, that's okay. Most of the applications' graphics in Squeak have mostly stable contents between frames.
𝕊 : Maybe because one cannot easily implement something more dynamic like a full-frame, animated 3D application?
ℍ : Oh, you can. You can write your own VM plugins to get more performance, bypassing objects and messaging. People have done that.
𝕊 : And what about the tangibility and directness that Morphic provides? Could you still click on those 3D

elements and explore them if your custom VM plugin does accelerated magic?

ℍ : Hmm... one would have to consider that, too, I suppose...

𝕊 : In Chrome, graphics are fast and I can still inspect all the nodes of the document object model. It actually feels like Squeak's Morphic.

Harmony tries to think of a feature in Squeak that is comparably powerful but not available in Sage's Lively system:

ℍ : But you cannot interrupt and debug a long-running script. The Smalltalk debugger is awesome! And users can hit CMD+Dot, any time the system gets stuck.

𝕊 : Well, no, we cannot do that. I miss that sometimes.

ℍ : Have you tried writing a browser plugin to extend debugging support somehow?

𝕊 : Not sure whether Google even wants you to have this kind of control over that very fast JavaScript VM...

ℍ : It seems like Squeak and Lively make quite different trade-offs.

𝕊 : Yes, it seems so.

`[|]`

Numerous concepts we appreciate in today's interactive environments have their roots in the 60's and 70's with Lisp machines and Smalltalk programming systems [30]. Being more than just languages, these *systems* experimented with – and successfully demonstrated – a refreshing *programming experience*: simplicity, purity,[6] reflection, meta-circularity, code simulation, overlapping windows, menus, icons, and many more. Also, non-programming end-users benefit from some ideas to this day. In the 80's and 90's, the concept of *direct manipulation* [11] emerged through projects such as the Alternate Reality Kit [31] and GUI frameworks such as Morphic [18, 19, 34]. Over the years, many of these concepts got replicated and adapted in all kinds of systems and applications. Unfortunately, originally pure ideas faded alongside modern experiments, trends, and even fears ... safety first? Compromise is ubiquitous, maybe even infectious. That is, for example, one can find dynamic languages and interactive windows in modern environments, but the purity and openness of Smalltalk systems remains unmatched.

Harmony works with Squeak/Smalltalk, which runs on the OpenSmalltalk VM.[7] She enjoys the pure object-oriented paradigm where source code describes object structure and messaging between objects to yield the desired behavior. Thanks to the creators of Squeak, even the virtual machine can be extended with a comparably efficient programming experience [14, 20]. All the original concepts are there. Harmony can build new things, maybe even push the originals to the next level. Baby steps toward innovation? Maybe. Unfortunately, the (rather small) Squeak community relies on mostly unpaid, voluntary efforts. While there is compromise at times, it is not so much for the sake of growing or staying competitive. Yet, little competition means only little alternatives to choose from and reuse in your project, at least compared to modern web-development platforms. Plus, different Smalltalk dialects and versions can make reuse even more challenging due to compatibility issues. A custom implementation (or fork) might be inevitable.

Sage works with Lively [13, 17], a Smalltalk-inspired programming system for the Web that uses modern web technologies such as HTML, CSS, and JavaScript. Just visit a web page[8] and get started. Thus, Sage enjoys the benefits of a company-backed, fast JavaScript VM and the fact that many computers have such web browsers these days. This means that Lively can rely on a virtually unlimited community of web developers, directly or indirectly contributing to this programming system through shared JavaScript modules. Unfortunately, Lively programmers live at the mercy of big companies, who increasingly see more value in *security* and *liability* than in *openness*. As an effect, the leverage for systems like Lively might fade away eventually. A generic `eval()` might be prohibited, only typed scripting languages allowed, maybe ending up in a Harvard architecture, where programs and data are separate after all. Sage's desire for reflection and malleability might not be possible to achieve anymore. The history knows the (remotely related) cases of ActiveX and Flash: huge potential, moderate acceptance, not future-proof.

All in all, *Harmony* values repairability and thus avoids reusing external artifacts if possible, having to (re-)implement features herself. She is virtually in control of the entire runtime that drives the Smalltalk system. *Sage* enjoys reusing foreign JavaScript modules, provided by the huge community of web developers. She is, unfortunately, dependent on the design decisions made by the companies in charge of the runtime (i.e., web browser), which might interfere with her values and practices. In both cases, it is challenging to balance the pros and cons of repairability versus reuse.

3 Our Tools and Materials

> *Squeak is not that system where you just put in content as objects to then never find them again. Its inward openness and outward integrated-ness is unprecedented.*

We borrowed the term "tools and materials" from the eponymous metaphor [4, 28], which sketches a workshop-like setting to make all kinds of programming tasks tangible. That is, software artifacts can be both tools and materials, depending on the perspective and task. We like this abstraction away from objects while still capturing a notion of self-sustaining malleability. This essay's scope is not limited to object-oriented programming; it rather focuses on liveness

[6] Every "thing" in the system is an object, even numbers or classes.

[7] The OpenSmalltalk VM, https://opensmalltalk.org/

[8] The Lively web programming system, https://lively-kernel.org/

and exploration as core values.[9] Yet, we are aware that our expertise in Squeak/Smalltalk frames this discussion and guides it toward objects and messaging.

Squeak, together with the OpenSmalltalk VM, is a typical Smalltalk implementation, where the object memory[10] can be saved to disk at any time, the current control flow be resumed later. While you can use these checkpoints (or images) to undo mistakes, they make the entire system feel like running forever. This compares to operating systems, their file system, and the suspend-to-disk feature. Programs within the Squeak system are made of objects, which can of course come and go as you design via messaging, which thus mimics the exit or restart of "conventional" applications.

We believe that having everything built from the same material is beneficial for program comprehension and tool integration. In a basic Squeak image, there are objects for input, graphics, or sound [34]. The Morphic framework provides a *UI process*, which sets up and keeps certain objects in action so that users can experience an interactive system. Building new features means designing new (kinds of) objects and messages, directly integrated with the existing ones. This *same material* – the objects – can then be explored in the shared object space (or memory) to be understood and refined. While there are means to reach outside this space, such as into files on disk, tool support is rather limited because the system cannot be in control of everything that happens there. Compromise is inevitable.

We leverage repairability by following exploratory practices [36]. In particular, we explore by having "conversations" with the system through one or more tools, each providing a useful (informational) context [35]. Squeak's toolset shines with at least three beautiful aspects: (1) code evaluation, (2) code simulation, (3) graphics inspection. First, programmers can evaluate Smalltalk expressions in any tool's text buffer; the tool's context will provide (variable) bindings as local context. Second, programmers can suspend any process to simulate its execution (or message sends) stepwise; the Smalltalk debugger works that way. Third, programmers can meta-click on graphical elements to reveal a (Morphic) halo to then open inspectors on their object structure to finally reach the code responsible for their actions. In sum, we can observe objects with or without inherent graphical representations, spot frictions in their state or behavior, and *repair* those efficiently. We are not afraid of debuggers; we appreciate them. We are happy to see the first 80 percent working with little effort; we take care of the rest along the way. Thus, it is often a viable strategy to reimplement a concern to assure repairability.

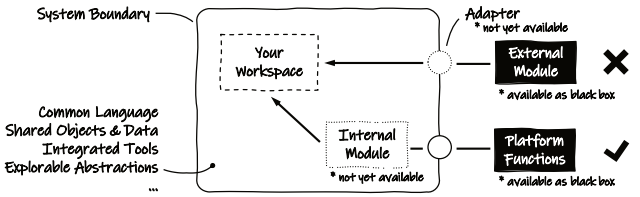

Figure 1. Programmers can choose between writing a new *internal* or reusing an existing *external* module. Both options entail different trade-offs. Reaching outside the system boundary might violate core values and impede exploratory practices. A reimplementation might not be feasible for small communities.

Squeak programmers are not trapped inside the system as illustrated in Figure 1. Maybe not even being aware of it, they reach out constantly when working with ordinary things such as numbers and files. The VM offers three mechanisms for reusing components from the host platform [23]: (1) primitive calls, (2) plugin calls, (3) FFI calls. First, there are (numbered or named) *primitives*, which primarily bypass the object world to do fast operations such as arithmetics, array manipulation, and closure handling. Besides performance, there are also primitives that write object memory to disk, access the clipboard, or set the host-window extent. This kind of reuse requires programmers to modify the VM and deploy a custom version. Second, there are modular *VM plugins*, which can be deployed separately and loaded on-the-fly while the VM is running. For example, file and socket access is realized through plugins. Via network access, programmers can send web requests and reuse components from the Internet. Third, there is *FFI*, which is itself a plugin, but generalizes calls out of the Squeak system to arbitrary C libraries. Given that a C library is already part of the host platform, programmers can now focus on writing and shipping pure Smalltalk code. However, there are fewer "safety nets," which limits the usual Smalltalk debugging experience. For example, memory access violations can crash the entire VM and thus lead to data loss.

4 One Paradigm Fits All?

> *Objects themselves should not be the reason for or against reuse. Identity, state, and behavior are capable of capturing other paradigms as well.*

Smalltalk is a multi-paradigm language, best suited for problems that can be expressed through *objects* and messaging. Still, the Collections interface [5] promotes a *functional* programming style; class-side methods can represent *procedures*. Yet, a good object-oriented design considers *identity*, *state*, and *behavior* [3, pp. 75–87]. An object-based library might be especially challenging to reuse if it has a different notion

[9] We invite readers who are also programmers to take a closer look at their favorite programming language, environment, or system. What are the means that make you feel most productive? Chances are that those tools origin from Smalltalk systems. Also, probably, some valuable exploratory practices [35, 36] got "lost in translation" and should be recovered.

[10] Actually, Squeak *is* the object memory, yet there are several Squeak-specific implementation details built into the virtual machine such as where to find fundamental classes/objects in the image file.

of identity. For example, "ids" or "names" can be manifold when integrating external databases, apparently providing "objects" without behavior, which is added only by *your* application. For another example, while Squeak's object memory is persistent, the VM itself entails volatile state from the host's perspective. That is, any file handles stored as objects will be invalid, the next time the VM starts. At the end of the day, replication or combination of multiple paradigms can increase the complexity of your implementation, regardless of whether you plan for repairability or reuse.

The *expression problem* [16] contributes to the challenges of reuse, even within the same system. Provided that your program consists of data types and operations, you can choose whether you want to allow for new data types or new operations without touching the existing implementation. You often cannot have both. For example, in a functional design, you can easily add operations, but you have to update existing code for new data types. It is the other way around in an object-based design, where you can often easily refine objects through (class-based) inheritance, but you have to touch all classes for new operations. The *Visitor* pattern [9, pp. 331–349] flips this trade-off again, yet does not resolve it. The expression problem often entails a notion of *compilation boundary*, where a compiled module cannot be changed at all and thus extensibility must be provided in a different way. While Squeak comes with all the source code, you might not want to change the base system to avoid compatibility issues with projects that live nearby.

We see *data-driven* programming as a valuable paradigm (or style or pattern) for interactive, object-oriented systems. Programmers might want to apply an operation to a collection of objects to process a plenitude of related data points. On the one hand, some form of *automation* can help mitigate issues of direct manipulation [11], like *verb-noun* interaction [24, pp. 59–62] on multiple "nouns." On the other hand, *external resources* drive many applications these days: streams of data from (hardware) sensors, iterators (or cursors) over the results of database queries, anything time-related that users want to explore and understand. *Dataflow* is prevalent in systems that operate on data points about "time and space." *Datalogy* [22, pp. 175–325] (or data science) is a field that has been complementing *computation* (or algorithm science) for a very long time. We associate the latter with an object's *behavior*, the former with its *state*. Both require sufficient support in programming systems. If you want to implement a new feature that benefits from a data-driven perspective, efforts might increase unless such a framework is already available. The UNIX pipes-and-filters pattern [25, pp. 266–280] is a famous example of script-based, data-driven, modular programming. Does your favorite system have such a framework, maybe even integrated with other programming styles?

5 Wanted: Expert Knowledge

There is good Smalltalk code. There is bad Smalltalk code. Programmers better become domain experts and master explorers.

Programmers might want to reuse a component because they are no domain experts, they might even lack basic domain knowledge. The *correctness* of an implementation is a quality that determines its acceptance. Does it minutely follow the (publicly available) specification? While knowledge can be acquired during iterative development, it takes time to become an expert, efforts that might better be spent in designing and polishing the solution. What should be an object? How to design the messages? A rule of thumb suggests: "Don't roll your own security." While specifications are open and detailed, inaccurate implementations can have severe consequences. Squeak follows this advice, for example, by interfacing the host's security components through a VM plugin. Eventually, there can be implicit knowledge [21] in a specification, hidden "between the lines." A fair level of *robustness* accounts for unexpected input to gracefully continue operation. To look beyond and infer from the known facts, it requires expert knowledge.

Performance can be a critical factor and promotes either reuse or reimplementation. Think of a parser for regular expressions, support for code completion, or on-the-fly syntax highlighting. The *material* on which programs want to apply these tools is already *in* the system: strings, texts, AST nodes. Using external modules for these tasks might seem awkward, especially if interface adapters entail a noticeable overhead due to data copying. However, it takes time to tune parsers to satisfy certain performance requirements. For example, the regex engine should process gigabytes of data in under a second, the completion module should make suggestions based on thousands of nodes, or highlighting should be applied while typing within a few milliseconds. Yearlong experience and expert knowledge might be required to reach these goals. Reuse might better be tried and evaluated first before starting the adventure of reimplementation. But how to consider *unquantifiable* values such as feedback in a discussion about performance?

Exploratory programming encourages *experimentation* and embraces *failure* for the sake of *learning*. However, at some point, programs get released and actually used. People then start to depend on their functionality. Any uncaught mistake or freshly introduced error can have consequences. Was the change already deployed? How many users are affected? Is data loss involved? While security might be among the features of highest priority, any issue around correctness, performance, and overall stability can have a negative impact on the reputation of a community. Consequently, programmers might be tempted to reuse a component from another community not only to save time but also for plausible deniability. Still, users are not always that open-minded or

understandable. They might still blame the program they are using and thus the wrong community. If programmers have control over all the involved sources, they can at least quickly react and mitigate damage. External components might take longer to fix.

Expert knowledge might not be available if there is no clear specification in the first place. When you cannot know which experts to look for, you might as well become one yourself. Maybe there is a specification but it is incomplete and prone to change. Programming then entails not only iterating over the solution space but also the problem space. What *is* the domain? What terms and definitions belong to the domain vocabulary? Are objects a good fit? While these concerns benefit from onward communication with domain experts, they do require programmers to acquire expert knowledge as well. For example, we can observe this challenge with Squeak. Do we just want to preserve original ideas or do we want to innovate somehow? Can we discover new opportunities? Reinterpret familiar problems in modern contexts? Are there even better *role models*[11] worth imitating? What are the risks of growing a (modern) community at all cost? We are aware that not all Smalltalk programmers share the same skill set. There is good Smalltalk code, and there is bad Smalltalk code. It is not difficult to write code for the compiler; it is difficult to keep it readable and compatible with certain values. Exploratory practices [35, 36] can be quite challenging to teach to both fresh and longtime Smalltalkers.

6 Make a Release, Release the Flow of Exploratory Programming

> *If we could only keep on doing what we like to do. Explore, understand, create, refine. Embrace failure. No, we do not want to package a bug-free experience. There is no such thing.*

It feels like hitting the brakes. When the community works toward a new release of the Squeak system, the flow [6] of experimentation comes to a halt. We know that application developers need some kind of "anchor" to have some guarantees about interface stability. Programming against the "bleeding edge" version of a library can be quite stressful. However, our values and practices feel somewhat incompatible with the idea of "making a release." If someone missed crucial things from "that other 20 percent," who will have the capacity to patch the released version? Many users seem to start stress-testing only *after* the release, when they have something *tangible* to work with. In smaller communities, such a split of resources might not be sustainable. Still, the Squeak community manages to push out a release every one or two years. While there is the desire to do it more frequently, one must not neglect the overhead that comes along: writing release notes, motivating programmers to update their projects, finally starting to fix those nasty bugs that no one else wanted to deal with because they are not *that* serious... Yet, with every new "anchor," a community's reputation is put on trial.

It is the peak of bug-fixing efforts. Our assumption is that implementing the first 80 percent of a feature makes sense because exploratory practices help programmers cover the rest when needed. Squeak is an open system and everybody can participate with their skills and expertise. However, a new version of a system makes certain tacit promises, promises about that other 20 percent. Users might expect corner cases to work, the entire artifact to be polished. Unfortunately, there is no "dial" to crank up the testing or fixing efforts on short notice. Freezing the main branch to prohibit new features helps on a technical but not on a social level. If your mindset is tuned to *exploration*, you might as well just bear the release process and wait for the permission to add new things again. Experiments can always be done in your private workshop. You may actually never stop *your* flow of exploration.

It has structure but remains risky. The Squeak release process is simple: feature freeze for 4-8 weeks (aka. beta versions), code freeze for another 2-4 weeks (aka. release-candidate versions), and then back to trunk development (aka. alpha versions). Estimated times might double or triple depending on the issues discovered. There are over 5000 automated tests in Squeak 6.0, but this does not imply a "comprehensive" test suite. The community is expected to do extensive *manual* testing with their favorite projects. For three times, this essay's first author was a so-called *release manager*, which is a person who not only oversees the process but also invests a lot of time in bug-fixing and polishing. In Squeak 5.1, user-interface themes and "dark modes" got added, which entailed tedious discussions about colors. In Squeak 5.3, the remnants of Etoys [1] were cleaned out and revived, which only addresses an even smaller sub-community within the Squeak community.[12] In Squeak 6.0, support for high-resolution displays was improved, which entailed crucial updates in the VM.[13] Besides such "eye candy," many things got improved in the standard library, programming tools, and virtual machine. The version number itself is also a compromise: "6.0" could as well have been "5.4," but increasing the major version sends a stronger signal to encourage Squeakers to finally port *their* projects from some ancient version or private image.

[11] We think that many of today's, often file-based, programming environments do not serve as good examples for innovation, not even their cloud-based variants.

[12] Some time ago, Etoys was the number-one reason, people (including programmers) might have heard about Squeak/Smalltalk in the first place.

[13] The release process of the OpenSmalltalk VM is separate. We bundle the best possible VM version with each Squeak release. However, it is worthwhile to also try recent VMs with older images.

It might leave people behind. How big *is* the Squeak community actually? There are several active groups around the globe, yet, personal usage makes it hard to pin down an exact number of Squeakers. Smalltalk's image concept spawns many different scenarios. Users are in full control of a multimedia world filled with tools, games, simulations, and other kinds of applications. Thus, there is no need to maintain your program against official releases to be part of the Squeak family. Liveness, directness, malleability: users can live these values without having the most recent "features." By living in their custom image, they might even create a private (or fan) community without noticing it. Looking back on the last 10 years of Squeak releases, it was mostly[14] clean-up and polishing. Still, we think that this progress has helped the *entire* community to appreciate openness and liveness even more. Having modern features that "catch up" with popular software systems, the original notion of Smalltalk exploratory programming is now available to a new generation of programmers.

7 The Limits of Liveness

> *When programmers understand a system's limitations, they can get immediate feedback through domain-specific constraints.*

The *liveness* we value in Squeak entails several trade-offs during exploration. It is far from ideal: level-4 liveness [37] means that when programmers make a change, the system will exhibit an observable effect only *after some time*. The duration of this *emergence phase* depends on the particular application [26]; it may take forever. That is, programmers can change object behavior through code modification and object state through (manual) code evaluation. Whether or not that object participates in an activity that actually needs that state or behavior, they might not know in advance. Still, there are idioms that help grasp the scope of a change such as `initialize` methods, `update` methods, and Morphic's `step` methods. Programmers can then check whether their change is something invoked from those familiar places. This *gulf of evaluation* [11] can be a serious hurdle in interactive systems, interfering with direct manipulation and thus the feedback loop during exploration.

Experienced Smalltalkers know where to look for and find short feedback loops, independent from the kind of program. The code browser and idioms mentioned above are complemented with three powerful tools: (1) Workspace, (2) Inspector, and (3) Debugger. First, workspace-like interfaces are part of almost all other tools. The *Workspace* itself has the simplest form: a single text field where programmers can type Smalltalk expressions, evaluate arbitrary text selections, and manage state through (variable) bindings. They are in full control of the length of the feedback loop. Second, the *Inspector* reveals an object's state while also embedding a mini workspace to allow for state changes via Smalltalk expressions. The inspector's UI is refreshed frequently, exposing each instance variable's string representation. Third, the *Debugger* represents a suspended process, ready for code modification, state inspection, and workspace-like evaluation. Thus, debuggers practically combine all other Smalltalk tools. Programmers can change an *active* method from the process stack; control flow can be resumed from that point. This can lead to *immediate feedback* and a sense of *liveness*, even though programmers are working with an *immutable past* and an overall uncertain *emergence phase*.

Domain-specific application (or tool-building) frameworks can tame Squeak's unreliable emergence phase. Such frameworks *prescribe* a specific "choreography of objects," which then allows for direct tracing of code or state changes to observable effects. That is, a simple *Observer* [9, pp. 293–303] can leverage Squeak's meta-programming facilities to notify frameworks about such changes. We followed this approach and designed a *tool-building environment*, called Vivide [33]:

> *We propose a new perspective on graphical tools and provide a concept to build and modify such tools with a focus on high quality, low effort, and continuous adaptability. That is, (1) we propose an object-oriented, data-driven, declarative scripting language that reduces the amount of and governs the effects of glue code for view-model specifications, and (2) we propose a scalable UI-design language that promotes short feedback loops in an interactive, graphical environment such as Morphic known from Self or Squeak/Smalltalk systems.*

Our goal was to hide as much glue code as possible when connecting data sources to graphical views. Several observers in the environment are constantly monitoring *a certain collection of objects* to then precisely reinitialize the affected view models or graphics. Overall, this mechanism is a fine-grained version of automating a program's exit-and-restart or an object's destroy-and-recreate. We argue that having Vivide available will reduce the efforts required for reimplementing your favorite visualization or database adapter.

8 The Dilemma Revisited

So ... should we *reuse* that existing module, written in a different language by a different community following different values? We would *really* like to have an implementation of that feature in *our* system, written in Smalltalk by our community following our values... Can this desire form a vision that encourages an entire community to seek for innovation in the field of liveness and exploration? Or might it be one of

[14] The OpenSmalltalk VM made a lot of progress in terms of (JIT) performance and processor compatibility. Our argument focuses on the tools and practices that programmers experience through Squeak's graphical interface.

the reasons that such noble values get scattered and diminished to eventually be forgotten in history? We think that it comes down to two things: time and people. First, time is essential to fully understand certain practices and live the values. There will be too much compromise when things get rushed. Second, a single person might have an expedient idea, but often cannot succeed without the help of a group – a team or community – that shares a vision, lives the same values, and knows the best practices.

In this essay, we collected anecdotal perspectives that do not resolve the dilemma but emphasize its intricacy. We touched on conceptual, technical, and social issues. We are certain that liveness and exploratory practices are the way to go forward to tackle unknown challenges without leaving people and their values behind. However, when actually exploring the solution space and having to make decisions, programmers might be tempted to follow the "wrong" path, time and again. Time is always short; people may not be skilled enough. How to stay motivated to keep trying?

We surely missed many perspectives in this discussion. Being longtime Smalltalkers, we selected stories about "Repairability or reuse?" based on our daily practices. While there are other core values thinkable, we do favor the approaches of the two systems Squeak and Lively. Even there, compromise seems inevitable. Having *impact* might be the one value that overshadows others. Is it worthwhile what I am programming? Do people appreciate my efforts?

Finally, there are many open-source communities that have to cope with this dilemma. The Debian GNU/Linux distribution, for example, manages to ensure backwards compatibility through its package tracking system [32], a structured approach for reuse. There are more such Linux-centered communities way bigger than Lively or Squeak. And it seems to work out for them ... or does it? Who keeps track of the integrity of a community's core values? The answer to that question may be explored in another essay.

Acknowledgements

We are grateful to our colleagues Jens Lincke, Patrick Rein, and Tom Beckmann for pivotal discussions and valuable feedback on earlier versions of this essay. Their expertise in Smalltalk, JavaScript, exploratory programming, and web development revealed crucial perspectives. We thankfully acknowledge the financial support of the HPI Research School on Service-oriented Systems Engineering (www.hpi.de/en/research/research-schools) and the Hasso Plattner Design Thinking Research Program (www.hpi.de/en/dtrp).

References

[1] BJ Allen-Conn and Kimberly Rose. 2003. *Powerful Ideas in the Classroom.* Viewpoints Research Institute, Inc.

[2] Kent Beck and Cynthia Andres. 2004. *Extreme Programming Explained: Embrace Change (Second Edition).* Addison-Wesley.

[3] Grady Booch, Robert A. Maksimchuk, Michael W. Engle, Bobbi J. Young, Jim Conallen, and Kelli A. Houston. 2007. *Object-oriented Analysis and Design with Applications (Third Edition).* Addison-Wesley.

[4] Reinhard Budde and Heinz Züllighoven. 1992. Software Tools in a Programming Workshop. In *Software Development and Reality Construction.* Springer, 252–268. https://doi.org/10.1007/978-3-642-76817-0_20

[5] William R. Cook. 1992. Interfaces and Specifications for the Smalltalk-80 Collection Classes. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications* (Vancouver, BC, Canada). ACM, 1–15. https://doi.org/10.1145/141936.141938

[6] Mihaly Csikszentmihalyi. 2008. *Flow: The Psychology of Optimal Experience.* Harper Perennial Modern Classics.

[7] Richard P. Gabriel. 2012. The Structure of a Programming Language Revolution. In *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software.* 195–214. https://doi.org/10.1145/2384592.2384611

[8] Richard P. Gabriel. 2014. I Throw Itching Powder at Tulips. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Portland, OR, USA). ACM, 301–319. https://doi.org/10.1145/2661136.2661155

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Abstraction of Reusable Object-oriented Software.* Addison-Wesley.

[10] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley.

[11] Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. 1985. Direct Manipulation Interfaces. *Human-Computer Interaction* 1, 4 (12 1985), 311–338. https://doi.org/10.1207/s15327051hci0104_2

[12] Takashi Iba and Taichi Isaku. 2016. A Pattern Language for Creating Pattern Languages: 364 Patterns for Pattern Mining, Writing, and Symbolizing. In *Proceedings of the 23rd Conference on Pattern Languages of Programs.* 1–63. https://doi.org/10.5555/3158161.3158175

[13] Daniel H. H. Ingalls, Tim Felgentreff, Robert Hirschfeld, Robert Krahn, Jens Lincke, Marko Röder, Antero Taivalsaari, and Tommi Mikkonen. 2016. A World of Active Objects for Work and Play: The First Ten Years of Lively. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software.* 238–249. https://doi.org/10.1145/2986012.2986029

[14] Daniel H. H. Ingalls, Ted Kaehler, John H. Maloney, Scott Wallace, and Alan C. Kay. 1997. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (Atlanta, GA, USA). ACM, 318–326. https://doi.org/10.1145/263700.263754

[15] Gregor Kiczales. 1996. Beyond the Black Box: Open Implementation. *IEEE software* 13, 1 (1996), 8–11. https://doi.org/10.1109/52.476280

[16] Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. 1998. Synthesizing Object-oriented and Functional Design to Promote Re-use. In *European Conference on Object-Oriented Programming.* Springer, 91–113. https://doi.org/10.1007/BFb0054088

[17] Jens Lincke, Patrick Rein, Stefan Ramson, Robert Hirschfeld, Marcel Taeumel, and Tim Felgentreff. 2017. Designing a Live Development Experience for Web-Components. In *Proceedings of the Programming Experience 2017.2 (PX/17.2) Workshop* (Vancouver, BC, Canada). ACM, 28–35. https://doi.org/10.1145/3167109

[18] John H. Maloney. 2002. *An Introduction to Morphic: The Squeak User Interface Framework.* Prentice Hall, Chapter 2, 39–67.

[19] John H. Maloney and Randall B. Smith. 1995. Directness and Liveness in the Morphic User Interface Construction Environment. In *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology* (Pittsburgh, PA, USA). ACM, 21–28. https://doi.org/10.1145/215585.215636

[20] Eliot Miranda, Clément Béra, Elisa Gonzalez Boix, and Daniel H. H. Ingalls. 2018. Two Decades of Smalltalk VM Development: Live VM Development Through Simulation Tools. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. ACM, 57–66. https://doi.org/10.1145/3281287.3281295

[21] Peter Naur. 1985. Programming as Theory Building. *Microprocessing and Microprogramming* 15, 5 (5 1985), 253–261. https://doi.org/10.1016/0165-6074(85)90032-8

[22] Peter Naur. 1992. *Computing: A Human Activity*. ACM Press.

[23] Tobias Pape, Tim Felgentreff, Fabio Niephaus, and Robert Hirschfeld. 2019. Let Them Fail: Towards VM Built-in Behavior That Falls Back to the Program. In *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming* (Genova, Italy). ACM, 1–7. https://doi.org/10.1145/3328433.3338056

[24] Jef Raskin. 2000. *The Humane Interface: New Directions for Designing Interactive Systems*. Addison-Wesley.

[25] Eric S. Raymond. 2004. *The Art of UNIX Programming*. Addison-Wesley.

[26] Patrick Rein, Stefan Lehmann, Toni Mattis, and Robert Hirschfeld. 2016. How Live are Live Programming Systems?: Benchmarking the Response Times of Live Programming Environments. In *Proceedings of the Programming Experience 2016 (PX/16) Workshop* (Rome, Italy). ACM, 1–8. https://doi.org/10.1145/2984380.2984381

[27] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2018. Exploratory and Live, Programming and Coding. *The Art, Science, and Engineering of Programming* 3, 1 (2018), 1:1–1:33. https://doi.org/10.22152/programming-journal.org/2019/3/1

[28] Dirk Riehle and Heinz Züllighoven. 1995. A Pattern Language for Tool Construction and Integration based on the Tools and Materials Metaphor. In *Pattern Languages of Program Design*, James O. Coplien and Douglas Schmidt (Eds.). Addison-Wesley, 9–42.

[29] David W. Sandberg. 1988. Smalltalk and Exploratory Programming. *ACM SIGPLAN Notices* 23, 10 (10 1988), 85–92. https://doi.org/10.1145/51607.51614

[30] Beau Sheil. 1998. *Datamation®: Power Tools for Programmers*. Morgan Kaufmann, Inc., Chapter 33, 573–580. https://doi.org/10.1016/B978-0-934613-12-5.50048-3

[31] Randall B. Smith. 1987. Experiences with the Alternate Reality Kit: An Example of the Tension between Literalism and Magic. *IEEE Computer Graphics and Applications* 7, 9 (9 1987), 42–50. https://doi.org/10.1109/MCG.1987.277078

[32] Sebastian Spaeth, Matthias Stuermer, Stefan Haefliger, and Georg von Krogh. 2007. Sampling in Open Source Software Development: The Case for Using the Debian GNU/Linux Distribution. In *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*. IEEE, 166a–166a. https://doi.org/10.1109/HICSS.2007.471

[33] Marcel Taeumel. 2020. *Data-driven Tool Construction in Exploratory Programming Environments*. Ph. D. Dissertation. University of Potsdam, Digital Engineering Faculty, Hasso Plattner Institute. https://doi.org/10.25932/publishup-44428

[34] Marcel Taeumel and Robert Hirschfeld. 2016. Evolving User Interfaces From Within Self-supporting Programming Environments: Exploring the Project Concept of Squeak/Smalltalk to Bootstrap UIs. In *Proceedings of the Programming Experience 2016 (PX/16) Workshop* (Rome, Italy). ACM, 43–59. https://doi.org/10.1145/2984380.2984386

[35] Marcel Taeumel, Jens Lincke, Patrick Rein, and Robert Hirschfeld. 2022. A Pattern Language of an Exploratory Programming Workspace. In *Design Thinking Research: Achieving Real Innovation*. Springer, 111–145. https://doi.org/10.1007/978-3-031-09297-8_7

[36] Marcel Taeumel, Patrick Rein, and Robert Hirschfeld. 2021. Toward Patterns of Exploratory Programming Practice. In *Design Thinking Research: Translation, Prototyping, and Measurement*. Springer, 127–150. https://doi.org/10.1007/978-3-030-76324-4_7

[37] Steven L. Tanimoto. 2013. A Perspective on the Evolution of Live Programming. In *2013 1st International Workshop on Live Programming (LIVE)* (San Francisco, CA, USA). IEEE, 31–34. https://doi.org/10.1109/LIVE.2013.6617346

[38] Jason Trenouth. 1991. A Survey of Exploratory Software Development. *Comput. J.* 34, 2 (1 1991), 153–163. https://doi.org/10.1093/comjnl/34.2.153

Received 2022-09-05; accepted 2022-10-03