Technical Dimensions of Programming Systems

JOEL JAKUBOVIC, University of Kent, UK
JONATHAN EDWARDS
TOMAS PETRICEK, University of Kent, UK

Many programming systems go beyond programming languages. Programming is usually done in the context of a stateful environment, beyond just writing code, by interacting with a system through a graphical user interface. Much research effort focuses on building programming systems that are easier to use, accessible to non-experts, moldable and/or powerful, but such efforts are often disconnected. They are informal, guided by the personal vision of the authors and thus are only evaluable and comparable on the basis of individual experience using them. In other words, they fail to form a coherent body of research. It isn't clear how to build on what has been done. Much has been said and done that allows comparison of *programming languages*, yet no similar theories exist for *programming systems*. We believe that programming systems deserve a theory too. We examine some influential past programming systems and review their stated design principles, technical capabilities, and styles of user interaction. We propose a framework of *technical dimensions*, capturing the underlying system characteristics and providing a means for conceptualising and comparing programming systems. We thus break the holistic view of programming systems down into characteristics that can be compared or advanced independently. These make it possible to talk about programming systems in a way that can be shared and agreed or constructively disagreed upon, rather than relying just on individual experience of using them. Our aim is to bring programming systems into the focus of programming research by providing the foundations that allow such research to proceed in a more systematic way. In other words, we wish the designers of future programming systems to finally stand on the shoulders of giants.

ACM Reference Format:
Joel Jakubovic, Jonathan Edwards, and Tomas Petricek. 2021. Technical Dimensions of Programming Systems. In *Proceedings of PLoP*. ACM, New York, NY, USA, 29 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Many forms of software have been developed to enable programming. The classic form consists of a *programming language*, a text editor to enter source code, and a compiler to turn it into an executable program. Instances of this form are differentiated by the syntax and semantics of the language, along with the implementation techniques in the compiler or runtime environment. Since the advent of graphical user interfaces (GUIs), programming languages can be found embedded within graphical environments that increasingly define how programmers work with the language (by directly supporting debugging or refactoring, for instance.) However, the rise of GUIs also permits diverse visual forms of programming, including visual languages and GUI-based end-user programming tools. This paper relies on, and encourages, a shift of attention from *programming languages* to the more general notion of "software that enables programming"—in other words, *programming systems*.

A *programming system* may include tools, protocols, notations and languages. It is a software artefact that makes it possible to construct programs, debug them, and turn them into operational, maintained and evolvable artifacts running on appropriate hardware. This notion covers classic programming languages together with their

PLoP, 2021, online
© 2021 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

editors, debuggers, compilers and other tools. Yet it is intentionally broad enough to accommodate image-based programming environments like Smalltalk, operating systems like Unix, and hypermedia systems like Hypercard, as well as various other examples we will mention.

1.1 What is the problem?

There is a growing interest in broader forms of *programming systems*, both in the programming research community and in industry. On the one hand, researchers are increasingly studying topics such as *programming experience* and *live programming* that require considering not just the *language*, but further aspects of a given system. On the other hand, commercial companies are building new programming environments like Replit or low-code programming tools like Dark and Glide. Yet, such topics remain at the sidelines of mainstream programming research. While *programming languages* are a well-established concept, analysed and compared in a common vocabulary, no similar foundation exists for the wider range of *programming systems*.

The academic research on programming suffers from the lack of common vocabulary for talking about programming systems. While we can thoroughly assess programming languages, as soon as we add interaction or graphics into the picture, all we can say is that the resulting system is vaguely "cool" or "interesting". Moreover, when designing new systems, inspiration is often drawn from same few somewhat disconnected sources of ideas. These might be influential past systems like Smalltalk, programmable end-user applications like spreadsheets, or motivational illustrations by thinkers like Victor [Victor 2012].

Instead of forming a solid body of work, the ideas that emerge are difficult to relate to each other. Similarly, the research methods used to study programming systems lack the more rigorous structure of programming language research methods. They tend to rely on singleton examples, which demonstrate the author's ideas, but are inadequate methods for comparing new ideas with the work of others. This makes it hard to build on top and thereby advance the state of the art.

Studying *programming systems* is not merely about taking a programming language and looking at the tools that surround it. It presents a *paradigm shift* to a perspective that is, at least partly, *incommensurable* with that of languages. When studying programming languages, everything that matters is in the program code; when studying programming systems, everything that matters is in the *interaction* between the programmer and the system. As documented by Gabriel [Gabriel 2012], looking at a *system* from a *language* perspective makes it impossible to think about concepts that arise from interaction with a system, but are not reflected in the language. Thus, we must proceed with some caution. As we will see, when we talk about Lisp as a programming system, we mean something very different from a parenthesis-heavy programming language!

1.2 Contributions

We propose a new common language as an initial, tentative step towards more progressive research on programming systems. Our set of "Technical Dimensions for Programming Systems" seeks to break down the holistic view of systems along various specific "axes": verbal conceptual prompts inspired by the *Cognitive Dimensions of Notation* [Green and Petre 1996]. While not strictly quantitative, we have designed them to be narrow enough to be comparable, so that we may say one system has more or less of a property than another. Generally, we see the various possibilities as tradeoffs and are reluctant to assign them "good" or "bad" status. If the framework is to be useful, then it must encourage some sort of rough consensus on how to apply it; we expect it will be more helpful to agree on descriptions of systems first, and settle normative judgements later.

The set of dimensions can be understood as a map of the design space of programming systems (Figure 1). Past and present systems will serve as landmarks, and with enough of them, unexplored or overlooked possibilities will reveal themselves. So far, the field has not been able to establish a virtuous cycle of feedback where practitioners are able to situate their work in the context of others' for subsequent work to improve on. Our aim is to provide foundations for the study of programming systems that would allow such development.

1. We discuss in detail three clusters of technical dimensions: *interaction structure*, *notational structure*, and *conceptual structure*. The rest of our initial set resides in the Appendix.
2. We define these dimensions by reference to landmark programming systems of the past, and discuss any relationships between them.
3. We demonstrate the salience of these dimensions by applying them to example systems from both the past and present. We situate some experimental systems as explorations at the frontier of certain dimensions.

2 RELATED WORK

While we do have new ideas to propose, part of our contribution is simply integrating a wide range of existing concepts under a common umbrella. This work is spread out across different domains, but each part connects to programming systems or focuses on a specific characteristic they may have.

2.1 Which "systems" are we talking about?

The programming systems that shape our framework come from a few recognisable clusters:

- "Platforms" supporting arbitrary software ecosystems: UNIX, Lisp, Smalltalk, the Web
- "Applications" targeted to a specific domain: spreadsheets
- Mixed aspects of platform and application: HyperCard, Boxer, Flash, and programming language workflows

Richard Gabriel noted a "paradigm shift" [Gabriel 2012] from the study of systems to the study of languages in computer science, which informs our distinction here. One consequence of the change is that a *language* is often formally specified apart from any specific implementations, while *systems* resist formal specification and are often *defined by* an implementation. We do, however, intend to recognizeprogramming languages as a small part of the space of possible systems (Figure 1). Hence we refer to the *interactive programming system* aspects of languages, such as text editing and command-line workflow.

Our "system" concept is mostly technical in scope, with occasional excursions as in "Learnability & Sociability" (Section A.8). This contrasts with the more socio-political focus found in [Tchernavskij 2019]. It overlaps with Kell's conceptualization of Unix, Smalltalk, and Operating Systems generally [Kell 2013], and we ensure UNIX is a part of ours. We do not, however, extend it to systems distributed over networks, although our framework may still be applicable there.

2.2 Industry and research interest in programming systems

There is renewed interest in programming systems in both industry and research. In industry we see:

- Computational notebooks such as Jupyter[1] that make data analysis more amenable to scientists by combining code snippets and their numerical or graphical output in a convenient document format.
- "Low code" end-user programming systems that present a simplified GUI for developing applications. One example is Coda,[2] which combines tables, formulas, and scripts to enable non-technical people to build "applications as a document".
- Specialized programming systems that augment a specific domain. For example Dark,[3] which creates cloud API services with a "holistic" programming experience including a language and direct manipulation editor with near-instantaneous building and deployment.

[1]https://jupyter.org/
[2]https://coda.io/welcome
[3]https://darklang.com/

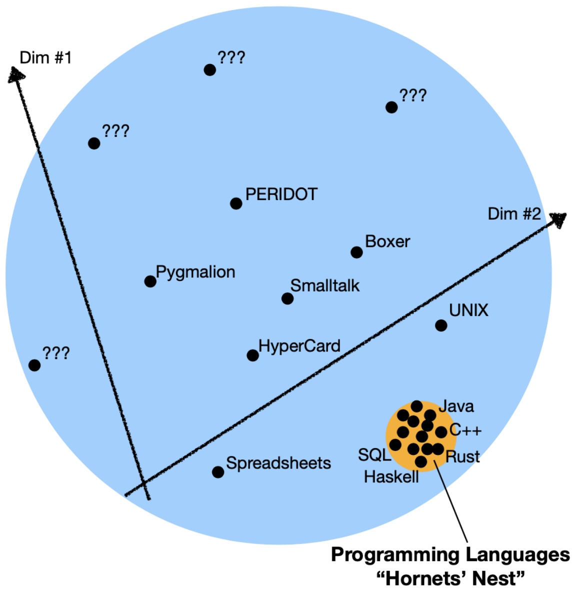

Fig. 1. A speculative sketch of one 2-dimensional slice of the space of possible systems.

- Even for general purpose programming with conventional tools, systems like Replit[4] have demonstrated the benefits of integrating all needed languages, tools, and user interfaces into a seamless experience available from a browser with no setup.

In research, there are an increasing number of explorations of the possibilities of full programming systems:

- Subtext [Edwards 2005], which combines code with its live execution in a single editable representation.
- Sketch-n-sketch [Hempel et al. 2019], which can synthesize code by direct manipulation of its outputs.
- Hazel [Omar et al. 2017], a live functional programming environment featuring typed holes which enable execution of incomplete or type-erroneous programs.

Several research venues investigate programming systems:

- VL/HCC (IEEE Symposium on Visual Languages and Human-Centric Computing)
- The LIVE programming workshop at SPLASH
- The PX (Programming eXperience) workshop at ⟨Programming⟩

2.3 Related methodology

There are several existing projects identifying characteristics of programming systems. Some of these revolve around a single one, such as levels of liveness [Tanimoto 2013], or plurality and communicativity [Kell 2017]. Others propose, as we do here, an entire collection:

- *Memory Models of Programming Languages* [Sitaker 2016] identifies the "everything is an X" metaphors underlying many programming languages.

[4]https://replit.com/

- The *Design Principles of Smalltalk* [Ingalls 1981] documents the philosophical goals and dictums used in the design of Smalltalk.
- The original *Design Patterns* [Gamma et al. 1995] names and catalogues specific tactics within the *codebases* of software systems.
- The *Cognitive Dimensions of Notation* [Green and Petre 1996] introduces a common vocabulary for software's *notational surface* and shows how they trade off and affect the performance of certain types of tasks.

Of these sources, the latter two bear the most obvious influence on our proposal. Our framework of "technical dimensions" continues the approach of the Cognitive Dimensions to the "rest" of a system beyond its notation. Our individual dimensions naturally fall under larger *clusters* that we present in a regular format, similar to the presentation of the classic Design Patterns. As for characteristics identified by others, part of our contribution is to integrate them under a common umbrella: Liveness, pluralism, and uniformity metaphors ("everything is an X") are incorporated as dimensions already identified by the related work.

We follow the attitude of *Evaluating Programming Systems* [Edwards et al. 2019] in distinguishing our work from HCI methods and empirical evaluation. We are generally concerned with characteristics that are not obviously amenable to statistical analysis (e.g. mining software repositories) or experimental methods like controlled user studies, so numerical quantities are generally not featured.

Similar development seems to be taking place in HCI research focused on user interfaces. The UIST guidelines[5] instruct authors to evaluate system contributions holistically, and the community has developed heuristics for such evaluation, such as *Evaluating User Interface Systems Research* [Olsen 2007]. Our set of dimensions offers similar heuristics for identifying interesting aspects of programming systems, though they focus more on underlying technical properties than the surface interface.

Finally, it is worth mentioning an influence from the history and philosophy of science. In fields like physics, superseded theories made predictions that were eventually falsified by the world they tried to model. Much of *computing*, however, involves creating custom "worlds" with their own rules, making the judgement cast by "supercession" seem rather less final. Hence we follow Chang's "Complementary Science" approach [Chang 2004], drawing our attention to historical systems that have been superseded, forgotten, or largely ignored—a "Complementary Computer Science".

2.4 What we are trying to achieve

In short, while there is a theory for programming languages, programming *systems* deserve a theory too. It should apply from the small scale of language implementations to the vast scale of operating systems. It should be possible to analyse the common and unique features of different systems, to reveal new possibilities, and to build on past work in an effective manner. In Kuhnian terms, it should enable a body of "normal science": filling in the map of the space of possible systems (Figure 1), thereby forming a knowledge repository for future designers.

3 HISTORY OF PROGRAMMING SYSTEMS

Here, we will present a brief and incomplete history of programming systems. This serves two purposes. First, looking at a number of examples from the past helps build an intuitive understanding of what we mean by a programming system. Second, it allows us to introduce example systems that we will use in the next section to illustrate the individual technical dimensions.

3.1 Interacting with computers

The key aspect of computers that enabled the rise of programming systems was the ability for a programmer to interact one-on-one with a computer. This was not possible in the 1950s when most computers were large and

[5]https://uist.acm.org/uist2021/author-guide.html

operated in a batch-processing mode. Two historical developments enabled such interactivity from the 1960s. First, time-sharing systems enabled interactive shared use of a computer via a teletype. Second, smaller computers such as the PDP-1 and PDP-10 provided similar direct interaction, while 1970s workstations such as the Alto and Lisp Machines added graphical displays and mouse input.

3.1.1 Lisp. The Lisp programming language (in the form of LISP 1.5 [McCarthy 1962]) was designed before the rise of interactive computers. Nevertheless, the existence of an interpreter plus the absence of declarations made it natural to use Lisp interactively, with the first interactive implementations appearing in the early 1960s. Two branches of the Lisp family,[6] MacLisp and the later Interlisp, fully embraced the so called "conversational" way of working. Interaction occured through the teletype at first, later giving way to the screen and keyboard. Even on the teletype, the system incorporated a number of ideas that remain popular with programming systems today.

Both MacLisp and Interlisp adopted the idea of *persistent address space.* The address space (what Smalltalk calls the *image*) contained both program code and program state, meaning that both could be accessed and modified interactively as well as programmatically using the *same means.* This idea of persistent address space appeared on time-sharing systems and culminated with the development of Lisp Machines, which embraced the idea that the machine runs continually and saves the state to disk when needed. Today, while this is still not the default state for systems running "natively" on some hardware, it is widely seen in cloud-based services like Google Docs and online IDEs.

One idea not widely seen today, yet pioneered in MacLisp and Interlisp, was the use of *structure editors.* These let programmers work with Lisp data structures not as sequences of characters, but as nested lists. In Interlisp, for example, the programmer would use commands such as `*P` to print the current expression, or `*(2 (X Y))` to replace its second element with the argument `(X Y)`. The PILOT system [Teitelman 1966], later integrated into Interlisp, offered even more sophisticated conversational features. For typographical errors and other slips, it would offer an automatic fix for the user to interactively accept, modifying the program in memory and resuming the execution.

3.1.2 Smalltalk. Smalltalk came on the scene in the 1970s, with the ambition of providing "dynamic media which can be used by humans of all ages". The authors saw computers as *meta-media* that could become a range of other media for applications like animation, circuit design, simulation and others not yet invented. Smalltalk was designed for single-user workstations with graphical display, and this display was pioneered not just for applications, but also for programming. In Smalltalk 72, one wrote code in the bottom half of the screen. When editing a definition, the window became a structure editor logically similar to that in Lisp, but controlled using a mouse and menu instead of a teletype.

Smalltalk shared a number of other characteristics with Lisp, although its key concept was that of an *object* rather than a *list.* Today, Smalltalk is perhaps best known for adopting the aforementioned persistent address space model of programming, where all objects remain in memory. Any changes made to the system state by programming or execution are preserved when the computer is turned off. The graphical interface then provides easy access to all objects and their methods through the *object browser.* The fact that much of the Smalltalk environment was implemented in itself, a property shared with many Lisp later systems, made it possible to significantly modify the system from within.

3.1.3 UNIX. Both Lisp and Smalltalk worked, to some extent, as operating systems. The user started their machine directly in the Lisp or Smalltalk environment and was able to do everything they needed from *within*

[6]The Lisp family consists of several branches, including MacLisp, InterLisp, ZetaLisp, Common Lisp, Scheme, Racket, and Clojure. see [Steele and Gabriel 1993]

the system.[7] This explains why it is worth considering (especially programmer-oriented) operating systems as programming systems as well. A prime example of this is UNIX, which was a simpler 1970s operating system for time-sharing computers.

As we discuss later under "Learnability & Sociability" (Section A.8), many aspects of programming systems are shaped by their intended target audience. UNIX was built for computer hackers themselves and, as such, its interface is close to the machine. While everything is an object in Smalltalk, the ontology of the UNIX system consists of files, memory, executable programs, and running processes. Interestingly, there is an explicit stage distinction here, that is not present in Smalltalk or Lisp. The ontology, however, enables an open pluralistic environment.

3.2 Early application platforms

The previously discussed programming systems did not focus on any particular kind of application. They were universal. As computers became more widely used, it became clear that there are typical kinds of applications that need to be built. For those, more specialized programming systems began to appear. Although they are more focused, they also support programming based on rich interactions with specialized visual and textual notations.

3.2.1 Spreadsheets. Spreadsheets, along with word processors, were the application that turned personal computers from playthings for hackers into a business tool. The first system, VisiCalc, became available in 1979 and helped analysts perform budget calculations. Spreadsheets developed over time, acquiring features that made them into powerful programming systems in a way VisiCalc was not. The final step was the 1993 inclusion of *macros* in Excel, later extended with *Visual Basic for Applications*. As programming systems, spreadsheets are notable for their programming substrate (a grid) and evaluation model (automatic re-evaluation).

3.2.2 HyperCard. While spreadsheets were designed to solve problems in a specific application area, the next system we consider was designed around a particular application format. 1987 saw HyperCard [Michel 1989], with programs as "stacks of cards" containing multimedia components and controls such as buttons. The logic associated with the controls could be programmed with pre-defined operations (e.g. "navigate to another card") or otherwise via the HyperTalk scripting language.

As a programming system, HyperCard is interesting for a couple of reasons. It effectively combines visual and textual notation. Programs appear the same way during editing as they do during execution. Most notably, HyperCard supports gradual progression from the "user" role to "developer": a user may first use stacks, then go on to edit the visual aspects or choose pre-defined logic until, eventually, they learn to program in HyperTalk.

3.3 Developer ecosystems

Programming systems such as Smalltalk and HyperCard are relatively *self-contained*. It is clear what is *part of* the system, and what is on the *outside*. For many systems that began to appear in the late 1980s, this is not the case. To think about them, we have to consider a number of components, some of which may be conventional programming languages.

3.3.1 Early and late Web. The Web appeared in 1989 as a way of sharing and organizing information, implementing the ideas of *hypertext*. The web gradually evolved from an *information sharing* system to a *developer platform* when client-side scripting using JavaScript became possible. The Web ecosystem started to consist of server-side and client-side programming tools. Today, the Web combines the notations of HTML, CSS, a wide range of server-side programming systems as well as JavaScript, and many languages that compile to JavaScript.

[7]When the Lisp and Smalltalk systems were implemented on specialized computers—InterLisp and Smalltalk on the Alto and Xerox D, ZetaLisp and Common Lisp on Lisp machines—the user would start their computers directly in the programming system environment. When implemented on commodity hardware, the user would resume a saved image of the programming system.

In the 1990s, the "early Web" became a widely used programming system. JavaScript code was distributed in a form that made it easy to copy and re-use existing scripts, which led to enthusiastic adoption by non-experts. This is comparable to the birth of microcomputers like Commodore 64 with BASIC a decade earlier.

The Web ecosystem continued to evolve. In the 2000s, multiple programming languages started to treat JavaScript as a compilation target, while JavaScript started to be used as a programming language on the server-side. This defines the "late Web" ecosystem, which is quite different from its early incarnation. JavaScript code was no longer simple enough to whimsically copy and paste, yet advanced developer tools provided functionality resembling early interactive programming systems like Lisp and Smalltalk. The *Document Object Model (DOM)* structure created by a web page is transparent, accessible to the user and modifiable through the built-in browser debugging tools, and third-party code to modify the structure can be injected via extensions. In this, the DOM resembles the *persistent image* model. The DOM also inspired further research on image-based programming: Webstrates [Klokmose et al. 2015] synchronizes DOM edits made in the browser to all other clients connected to a single server.

3.3.2 REPLs and notebooks. Another kind of developer ecosystem that evolved from simple scripting tools are the modern tools for data science, such as Jupyter. Their roots date back to conversational implementations of Lisp, where users could type commands to be evaluated and see the results printed. This interaction became known as the REPL (Read-Eval-Print Loop). In the late 1980s, Mathematica 1.0 combined the REPL interaction with a notebook document format that shows the commands alongside visual outputs.

Today, REPLs exist for many programming languages. Unlike in Lisp, they are often separate from the running program. REPLs often maintain an execution state independent of a running program and there are many strategies for prototyping code in a REPL before making it a part of an ordinary compiled application.

Notebooks for data science are a particularly interesting example. Their primary output is the notebook itself, rather than a separate application to be compiled and run. The code lives in a document format, interleaved with other notations. Code is written in small parts that are executed (almost) immediately, offering the user more rapid feedback than in conventional programming. A notebook can be seen as a trace of how the result has been obtained, although one often problematic feature of notebooks is that they allow the user to run code blocks out-of-order. Retracing the individual steps in a notebook may thus be more subtle than following a trace produced from a conventional REPL (for example, using the `dribble` function in Common Lisp).

3.3.3 Haskell and other languages. The aforementioned 1990s paradigm shift from thinking about *systems* to thinking about *languages* means that researchers tend to emphasize the language side of programming. However, all programming languages are a part of a richer ecosystem that consist of editors and other tools. In our analysis, we choose Haskell as our example as a clearly language-focused programming system.

Like most programming languages, Haskell code can be written in a wide range of text editors, some of which support assistance tools such as syntax highlighting and auto-completion. These offer immediate feedback while editing code, such as when highlighting type errors. This way, "lapse" and "slip" type errors are mitigated—we discuss this further under "Handling of Errors" (A.4).

Haskell is mathematically rooted and relies on mathematical intuition for understanding many of its concepts. This background is also reflected in the notations it uses. In addition to the concrete language syntax (used when writing code), the Haskell ecosystem also uses an informal mathematical notation, which is used when writing about Haskell (e.g. in academic papers or on the whiteboard). This provides an additional tool for manipulating Haskell programs and experimenting with them on paper *"in vitro"*, in ways that other systems may attempt to achieve through experimentation within the system *"in vivo"*.

4 TECHNICAL DIMENSIONS

In this section, we present our most worked-out technical dimensions under three *clusters*. This is a consequence of how we developed the framework. In the beginning, we considered these clusters the “dimensions” themselves—yet, as we added detail, they soon developed an internal “glossary” structure defining more primitive terms. Since it was easier to see how these primitives could be compared and assigned values for concrete systems, and unclear how the same would be possible for the clusters, we tentatively settled on the format that follows. The clusters may be regarded as “topics of interest” or “areas of inquiry” when studying a given system, grouping together related dimensions against which to measure it.

Each cluster is named and opens with a short *summary*, followed by a detailed *description*, and closes with a list of *examples* and any *relations* to other clusters. Within the main description, individual *dimensions* are identified. Sometimes, a particular value along a dimension (or a combination of values along several dimensions) can be recognized as a familiar pattern—perhaps with a name already established in the literature. These are marked as *instances*. Finally, interspersed discussion that is neither a *dimension* nor an *instance* is introduced as a *remark*.

For brevity, we present three dimension clusters from our current list, and leave the rest to the Appendix. These are *interaction structure*, *notational structure*, and *conceptual structure*.[8]

4.1 Interaction structure

4.1.1 Summary. How do users execute their ideas, evaluate the result, and generate new ideas in response?

4.1.2 Description. An essential aspect of programming systems is how the user interacts with them when creating programs. Take the standard form of statically typed, compiled languages with straightforward library linking: here, programmers write their code in a text editor, invoke the compiler, and then read through error messages they get.

Other forms are yet possible. For example, the compilation or execution of a program can contain further nested programmer interactions, or they may not even be perceptible at all. To analyze all forms of programming, we use the concepts of *gulf of execution* and *gulf of evaluation* from *The Design of Everyday Things* [Norman 2002].

Dimension: feedback loops. In using a system, one first has some idea and attempts to make it exist in the software; the gap between the user’s goal and the means to execute the goal is known as the *gulf of execution*. Then, one compares the result actually achieved to the original goal in mind; this crosses the *gulf of evaluation*. These two activities comprise the *feedback loop* through which a user gradually realises their desires in the imagination, or refines those desires to find out “what they actually want”.

A system must contain at least one such feedback loop, but may contain several at different levels or specialized to certain domains. For each of them, we can separate the gulf of execution and evaluation as independent legs of the journey, with possibly different manners and speeds of crossing them.

Instance: immediate feedback. The specific case where the *evaluation* gulf is minimized to be imperceptible is known as *immediate feedback*. Once the user has caused some change to the system, its effects (including errors) are immediately visible. This is a key ingredient of *liveness*, though it is not sufficient on its own. (See *Relations*)

The ease of achieving immediate feedback is obviously constrained by the computational load of the user’s effects on the system, and the system’s performance on such tasks. However, such “loading time” is not the only way feedback can be delayed: a common situation is where the user has to manually ask for (or “poll”) the relevant state of the system after their actions, even if the system finished the task quickly. Here, the feedback could be described as *immediate upon demand* yet not *automatically demanded*. However, for convenience, we include the latter criterion—automatic demand of result—in our definition of immediate feedback.

[8]The fact that these all have “structure” in the name is a curious coincidence—more variety can be found in our less fleshed-out clusters in the Appendix.

Instance: direct manipulation. The origin of *direct manipulation* is in the real world, such as programming a robot by physically dragging its arms to record the process that it will later replay. Since most interaction with software takes place through keyboards, screens, and mice, it is more common for this to be simulated.

In this case, direct manipulation is a special case of an immediate feedback loop. The user sees and interacts with an artefact in a way that is as similar as possible to real life; this typically includes dragging with a cursor or finger in order to physically move a visual item, and is limited by the particular haptic technology in use.

Naturally, because moving real things with one's hands does not involve any waiting for the object to "catch up"[9], direct manipulation is necessarily an immediate-feedback cycle. However, if one were to move a figure on screen by typing new co-ordinates in a text box, then this could still give immediate feedback (if the update appears instant and automatic) but would not be an example of direct manipulation.

Dimension: modes of interaction. Programming systems can give programmers different modes in which to operate, each mode potentially supporting different feedback loops. A typical example here is debugging. In many programming systems, the user can debug a running program and, in this mode, they can modify the program state and get (more) immediate feedback on what individual operations do. When a program is not running, and outside of debug mode, this kind of feedback loop is not available.

A programming system may also be designed with just a single mode. The Jupyter notebook environment does not have a distinct debugging mode; the user runs blocks of code and receives the result. The single mode can be used to quickly try things out, and to generate the final result. However, even Jupyter notebooks distinguish between editing a document and running code.

The idea of interaction modes goes beyond just programming systems, appearing in software engineering methodologies. In particular, having a separate *implementation* and *maintenance* phase would be an example of two modes.

4.1.3 Examples.

Spreadsheets have:

1. A tight, immediate feedback loop for direct manipulation of values and formatting. This is the same as in any other WYSIWYG application.
2. A loop for formula editing and formula invocation. Here, there is larger execution gulf for designing and typing formulas. The evaluation gulf is often reduced by editor features, e.g. immediate feedback previews for cell ranges. However, one sees what the *effect* of a formula is only on "committing" the formula, i.e. pressing enter.

In a *REPL* or *shell*, there is a main cycle of typing commands and seeing their output, and a secondary cycle of typing and checking the command line itself.

- The output of commands can be immediate, but usually reflects only part of the total effects or even none at all. The user must manually issue further commands afterwards, to check the relevant state bit by bit.
- The secondary cycle, like all typing, provides immediate feedback in the form of character "echo", but things like syntax errors are only reported *after* the entire line is submitted.
- One counterexample to this is Firefox's JavaScript console, where the line is "run" in a limited manner on every keystroke. Thus, simple commands without side-effects (e.g. a pure function call) can give instant previewed results, though partially typed expressions and syntax errors do not trigger previews.

A statically checked *programming language* (e.g. Java, Haskell) involves several feedback loops (Figure 2):

[9]In some situations, such as steering a boat with a rudder, there is a delay between input and effect. But on closer inspection, this delay is between the rudder and the boat; we do not see the hand pass through the wheel like a hologram, followed by the wheel turning a second later. In real life, objects touched directly give immediate feedback; objects controlled further down the line might not!

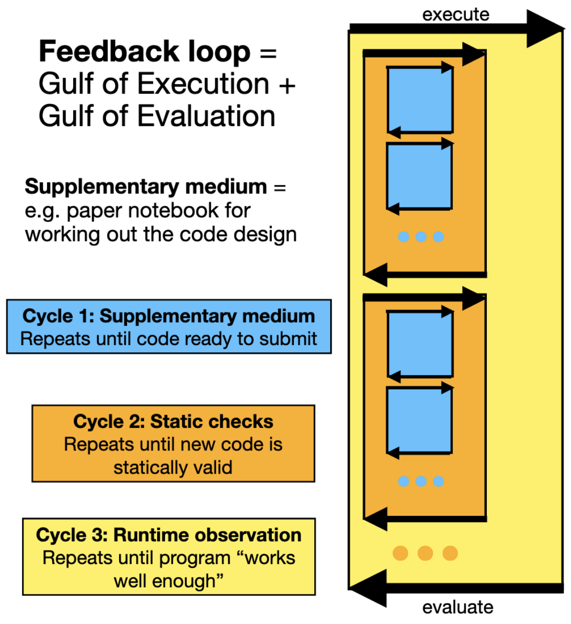
Feedback loop =
Gulf of Execution +
Gulf of Evaluation
Supplementary medium =
e.g. paper notebook for
working out the code design
Cycle 1: Supplementary medium
Repeats until code ready to submit
Cycle 2: Static checks
Repeats until new code is
statically valid
Cycle 3: Runtime observation
Repeats until program "works
well enough"
execute
evaluate

Fig. 2. The nested feedback loops of a statically-checked programming language.

1. The notation is plain monospaced text, so users will typically invoke another medium (pencil and paper, whiteboard, or another piece of software) to work out details (e.g. design, organization, mathematics) before translating this into code. This *supplementary medium* has its own feedback loop.
2. The code is written (through multiple iterations of cycle 1) and is then put through the static checker. Errors here send the user back to writing code, but in the case of success, they are "allowed" to run the program, leading into cycle 3.
   - The execution gulf comprises multiple cycles of the supplementary medium, plus whatever overhead is needed to invoke the compiler (such as build systems).
   - The evaluation gulf is essentially the waiting period before static errors or a successful termination are observed. Hence this is bounded by some function of the length of the code (the same cannot be said for the following cycle 3.)
3. With a runnable program, the user now evaluates the *runtime* behavior. Runtime errors can send the user back to writing code to be checked, or to tweak dynamically loaded data (e.g. data files) in a similar cycle.
   - The execution gulf here may include multiple iterations of cycle 2, each with its own nested cycle 1.
   - The *evaluation* gulf here is theoretically unbounded; one may have to wait a very long time or create very specific conditions to rule out certain bugs (e.g. race conditions) or simply to consider the program as fit for purpose.
   - The latter issue, of evaluating according to human judgement, is made tractable by targetting "good enough for this release" instead of some final, comprehensive measure.
   - By imposing static checks, some bugs can be pushed earlier to the evaluation stage of cycle 2, reducing the likely size of the cycle 3 *evaluation* gulf.
   - On the other hand, this can make it harder to write statically valid code, which may increase the number of level-2 cycles, thus increasing the total *execution* gulf at level 3.

- Depending on how these balance out, the total top-level feedback loop may grow longer or shorter.

4.1.4 *Relations.*

- A longer evaluation gulf delays the detection of errors. A longer execution gulf can increase the *likelihood* of errors (e.g. writing a lot of code or taking a long time to write it). By turning runtime bugs into statically detected bugs, the combined evaluation gulfs can be reduced. All of these are implications for *handling of errors* (Section A.4.)
- The *execution* gulf is concerned with software using and programming in general. The time taken to realizean idea in software is affected by the user's familiarity and the system's *learnability* (Section A.8), as well as the *expressive power* of the system's ontology.
- *Liveness* is impossible without immediate feedback. However, the motto "The thing on the screen is supposed to be the actual thing" [Ungar and Smith 2013] suggests that representations should be "canonical" in some sense, i.e. equipped with faithful behavior rather than intangible shadows cast by the hidden *real* object. This seems to overlap with Direct Manipulation and *bidirectionality*, but it is unclear if it is meant to be synonymous with them.
- Different *notational structures* have different feedback loops.

4.2 Notational structure

4.2.1 Summary. What are the different notations, both textual and visual, through which the system is programmed? How do they relate to each other when programming the system?

4.2.2 Description. Programming is always done through some form of notation. We consider notations in the most general sense and include any structured gesture using textual or visual notation, user interface or other means. Textual notations include primarily programming languages, but may also include, for example, configuration files. Visual notations include graphical programming languages. Other kinds of structured gestures include, for example, user interfaces for constructing visual elements used in the system. *Cognitive Dimensions* [Green and Petre 1996] provides a comprehensive framework for analysing individual notations, but our focus here is on how multiple notations are combined, i.e., the notational structure.

In practice, most programming systems use multiple notations. Even when we consider traditional programming languages, those *primary notations* are often supported by *secondary notations* such as annotations encoded in comments, and build tool configuration files. More interestingly, some systems have multiple different *primary* notations. Those can be either provided as different ways of programming the same part of a system, or they can be provided as complementing ways of programming different aspects of a system.

Dimension: overlapping notations. A programming system may provide multiple notations for programming the same aspect of the system. An example is Sketch-n-sketch [Hempel et al. 2019], a tool for creating and editing SVG and HTML documents. Sketch-n-sketch displays the source code of the document side-by-side with the resulting output. The author can edit either of the two. An edit in the source code will be immediately visible in the output, and editing the output will cause corresponding changes to the source code.

The crucial issue in this kind of arrangement is synchronization between the different notations. If the notations have different characteristics, this is not a straightforward mapping—for example, source code may allow a more elaborate abstraction mechanism (such as a loop) which will not be apparent in the output (it will result in visible repetitions). What should such system do when the user edits a single object that resulted from such repetition? For programming systems that use *overlapping notations*, we need to describe how the notations are synchronized.

Dimension: complementing notations. A programming system may also provide multiple complementing notations for programming different aspects of its world. An example is Excel spreadsheets, where there are two different notations that complement each other. The first is the formula language, where users describe calculations with data on the sheet. The second is the macro language (Visual Basic for Applications), which lets users automate the system in ways that go beyond what is possible through the formula language. There is also a third, even more basic, level which is entering data into the sheet. This cannot be used to specify new computations, but can be used to adapt or run a computation, for example when entering different alternatives in "What-If Scenario Analysis".

The key issue for systems with complementing notations is how the different notations are connected and how the user may progress from one to the next level if they need. In Excel, macros can be used to control the lower level (you can write macros that evaluate formulas). This is also a question for learnability. In Excel, you need to learn a completely different language if you want to move to the macro level. The approach optimizes for easy learnability at one level, but introduces a hurdle that users have to cross to get to the second level.

Dimension: primary and secondary notations. Some systems have multiple primary notations. They can live alongside each other, but they can also be nested. In HyperCard [Michel 1989], there is a visual "notation" for designing the user interface: the card itself, which can be edited directly. Then there is HyperScript, which is secondary and can be associated with individual controls.

Programming systems typically have further notations or structured gestures associated with them. The primary notation in UNIX is the C programming language. Yet this is enclosed in a programming *system* providing (multi-step) mechanisms for running C code via the terminal, assisted by secondary notations such as shell scripts. Even an ordinary programming language such as Java is only a part of (at least one) programming system that includes structured gestures—running code in Eclipse—and secondary notations for the management of dependencies and build configurations.

Some programming systems attempt to integrate tools that normally rely on secondary notations into the system itself, reducing the number of secondary notations that the programmer needs to master. For example, in the Smalltalk descendant Pharo, versioning and package management is done from within Pharo, removing the need for secondary notation such as `git` commands and dependency configuration files.[10]

Dimension: expression geography. A crucial feature of a notation is the relationship between the structure of the notation and the structure of the behavior it encodes. Most importantly, do *similar expressions*, in a particular notation, represent *similar behavior*?

In textual notations, this may easily not be the case. Consider the two C conditionals:

- `if (x==1) { ... }` evaluates the Boolean expression `x==1` to determine whether `x` equals `1`, running the code block if the condition holds.
- `if (x=1) { ... }` assigns `1` to the variable `x`. In C, assignment is an expression returning the assigned value, so the result `1` is interpreted as `true` and the block of code is *always* executed.

A notation can be designed to map better to the logic behind it, for example, by requiring the user to write `1==x`. This solves the above problem as `1` is a literal rather than a variable, so it cannot be assigned to (`1=x` is a compile error).

Visual notations may provide more or less direct mapping. On the one hand, similar-looking code in a block language may mean very different things. On the other hand, similar looking design of two HyperCard cards will result in similar looking cards—the mapping between the notation and the logic is much more direct in this case.

Dimension: uniformity of notations. One common concern with notations is the extent to which they are uniform. A uniform notation can express a wide range of things using just a small number of concepts. The

[10]The tool for versioning and package management in Pharo can still be seen as an *internal* domain-specific language and thus as a secondary notation, but its basic structure is *shared* with other notations in the Pharo system.

primary example here is S-expressions from Lisp. An S-expression is either an atom or a pair of S-expressions written `(s1 . s2)`. By convention, an S-expression `(s1 . (s2 . (s3 . nil)))` represents a list, written as `(s1 s2 s3)`. In Lisp, the uniformity of notations is closely linked to uniformity of representation. In the idealized model of LISP 1.5, the data structures represented by an S-expression is what exists in memory. In real-world Lisp systems, the representation in memory is more complex. A programming system can also take a very different approach and fully separate the notation from the in-memory representation.

Lisp source code is represented in memory as S-expressions, which can be manipulated by Lisp primitives. In addition, Lisp systems have robust macro processing as part of their semantics: expanding a macro revises the list structure of the code that uses the macro. Combining these makes it possible to define extensions to the Lisp system in Lisp with syntax indistinguishable from Lisp. Moreover, it is possible to write a Lisp program that constructs another Lisp program and not only run it interpretively (using the Lisp function `eval`) but compile it at runtime (using the Lisp function `compile`) and then execute it. Many domain-specific languages as well as prototypes of new programming languages (such as Scheme) were implemented this way. Lisp, the language, is, in this sense, a programmable programming language. [Felleisen et al. 2018; Foderaro 1991]

A programming system can also take a very different approach and fully separate the notation from the in-memory representation.

4.2.3 *Primary examples.*

- *Spreadsheets and HyperCard.* Spreadsheet systems such as Excel use complementing notations consisting of (i) the visual grid, (ii) formula language and (iii) macro language such as VBA. The notations are largely independent and have different degrees of expressivity. A user gradually learns more advanced notations, but experience with previous notation does not help with mastering the next one. The notational structure of HyperCard is similar and consists of (i) visual design of cards, (ii) visual programming (via menu) with a limited number of operations and (iii) HyperTalk for scripting.
- *Boxer and Jupyter.* Boxer [diSessa and Abelson 1986] uses *complementing notations* in that it combines a visual notation (the layout of the document and the boxes it consists of) with textual notation (the code in the boxes). In case of Boxer, the textual notation is always nested within the visual. The case of Jupyter notebooks is similar—the document structure is graphical (edited by adding and removing cells); code and visual outputs are nested as cells in the document. This arrangement is common in many other systems such as Flash or Visual Basic, which both combine visual notation with textual code, although one is not nested in the other.
- *Haskell.* The primary notation is the programming language, but there are also a number of secondary notations. Those include package managers (e.g. the `cabal.project` file) or configuration files for Haskell build tools. More interestingly, there is also an informal mathematical notation associated with Haskell that is used when programmers discuss programs on a whiteboard or in academic publications. Indeed, the idea of having a mathematical notation dates back to the Report on Algol 58 [Perlis and Samelson 1958], which explicitly defined a "publication language" for "stating and communicating problems" that used Greek letters and subscripts.

4.2.4 *Further examples.*

- *Sketch-n-sketch.* Sketch-n-sketch [Hempel et al. 2019] is an example of a programming system that implements the *overlapping notations* structure. The user edits HTML/SVG files in an interface with a split-screen structure that shows source code on the left and displayed visual output on the right. They can edit both of these and changes are propagated to the other view. In terms of *expression geography*, the code can use abstraction mechanisms (such as functions) which are not visible in the visual editor. The visual view thus hides some aspects of the geography, but is more "pedantic" in other aspects. More generally, Sketch-n-sketch can be seen as an example of a *projectional editor*.

- *UML Round-tripping.* Another example of a programming system that utilizes the *overlapping notations* structure are UML design tools that display the program both as source code and as a UML diagram. Edits in one result in automatic update of the other. An example is the Together/J system. In order to match the two notations, such systems often need to store additional information (e.g. location of classes in UML diagram after the user rearranges them) in the textual notation. This can be done by using a special kind of code comment.
- *Paper Tools.* The importance of notations in the practice of science, more generally, has been studied by Klein [Klein 2003] as "paper tools"—things like formulas, which can be manipulated by humans in lieu of experimentation. Programming notations are similar, but they are a way of communicating with a machine—the experimentation does not happen just on paper. One exception here is the mathematical notation in Haskell, which is used as a "paper tool" for experimentation on a whiteboard. In the context of programming notations, the Cognitive Dimensions framework [Green and Petre 1996] offers a detailed way of analysing individual notations.

4.2.5 Relations.

- *Factoring of complexity* (Section A.6): Using multiple complementing notations implicitly factors complexity by expressing different aspects of a program using different notations.
- *Interaction structure* (Section 4.1): The feedback loops that exist in a programming system are typically associated with individual notations. Different notations may also have different feedback loops.
- *Learnability* (Section A.8): Notational structure can affect learnability. In particular, complementing notations may require (possibly different) users master multiple notations. Overlapping notations may improve learnability by allowing the user to edit program in one way (e.g. visually) and see the effect in the other notation (e.g. in code).

4.3 Conceptual structure

4.3.1 Summary. What tradeoffs are made between conceptual integrity and diversity? How compatible is the system with established technologies? How open is it to the changing practices of the software industry?

4.3.2 Description.

> I will contend that Conceptual Integrity is the most important consideration in system design. It is better to have a system omit certain anomalous features and improvements, but to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas. — Fred Brooks [Brooks 1995]

> Conceptual integrity arises not (simply) from one mind or from a small number of agreeing resonant minds, but from sometimes hidden co-authors and the thing designed itself. — Richard Gabriel [Gabriel 2008]

The evolution of programming systems has led away from conceptual integrity, towards an enormously complex ecosystem of specialized technologies and standards. Any attempt to unify parts of this ecosystem into a coherent whole will create *incompatibility* with the remaining parts, which becomes a major barrier to adoption. Designers seeking adoption are pushed to focus on localized incremental improvements that stay within the boundaries of existing practice. This tension manifests in two dimensions: how open we are to the pressures imposed by society, and how highly we value conceptual elegance.

Dimension: conceptual integrity. Smalltalk and Lisp elegantly provide a uniform way to compose elements: *objects* and *lists*[11], respectively. In UNIX, everything is a file allowing reading and writing of a binary data stream[12], or a directory naming other files and directories. The great advantage to everything being the same sort of thing is that code written to handle those things becomes universally applicable. One avoids the babel of many languages for saying the same thing.

The *Memory Models of Programming Languages* essay [Sitaker 2016] suggests further examples:

- In COBOL, state consists of nested records in a tax form.
- In Fortran, state consists of parallel arrays.
- In SQL, state is a set of relations with key constraints.

Dimension: conceptual openness. The choice between compatibility and integrity correlates with the personality traits of pragmatism and idealism. It is pragmatic to accept the status quo of technology and make the best of it. Conversely, idealists are willing to fight convention and risk rejection in order to attain higher goals. We can wonder which came first: the design decision or the personality trait? Do Lisp and Haskell teach people to think more abstractly and coherently, or do they filter for those with a preexisting condition? Likewise, introverted developers might prefer the cloisters of Smalltalk or Lisp to the adventurous Wild West of the Web.

Dimension: conceptual granularity. Many programming languages and systems impose structure at a "fine granularity": that of individual variables and other data and code structures. This replaces and constrains the flat, "anything goes" memory landscape of the machine level, and the similar "run it and see what happens" indifference of machine instruction streams. Conversely, systems like UNIX or the Web impose fewer restrictions on how programmers represent things—in UNIX's case, delegating all fine-grained structure to the client program and insisting only on a basic infrastructure of "large objects" [Kell 2013]. The price to pay for this flexibility is that the system can provide relatively little insight into "what is going on" inside the programs if they use a peculiar representation.

Instances: worse is better. Richard Gabriel first described the problem in his influential 1991 essay "Worse is Better" [Gabriel 1991] analyzing the defeat of Lisp by Unix/C. Because Unix and C were so easy to port to new hardware, they were "the ultimate computer viruses" despite providing only "about 50%–80% of what you want from an operating system and programming language". Their conceptual openness meant that they adapted well to the evolving conditions of the external world.

Remark: the end of the history? Today we live in a highly developed world of software technology. It is estimated that 41,000 person years have been invested into Linux. We describe software development technologies in terms of *stacks* of specialized tools, each of which might capitalize over 100 person years of development. Programming system designers face an existential question: how to make a noticeable impact on the overwhelming edifice of existing technology? There are strong incentives to focus on localized incremental improvements that don't cross the established boundaries.

The history of computing is one of cycles of evolution and revolution. Successive cycles were dominated in turn by mainframes, minicomputers, workstations, personal computers, and the web. Each transition built a whole new technology ecosystem replacing or on top of the previous. The last revolution, the web, was 25 years ago. Many people have never experienced a disruptive platform transition. Has history stopped, or are we just stuck in a long cycle, with increasingly pent-up pressures for change? If it is the latter, then incompatible ideas now spurned may yet flourish.

[11]ISP 1.5 also had arrays but no literal syntax for them; later Lisps supported many other types of collections, data structures, and object systems, but kept code as lists.
[12]With the escape hatch of `ioctl` calls.

Instances: living with pluralism. Python and Perl provide opposite examples when it comes to conceptual integrity. On the one hand, Python follows the principle that “There should be one—and preferably only one—obvious way to do it” in order to promote community consensus on a single coherent style. On the other hand, Perl states that “There is more than one way to do it.” and considers itself “the first postmodern programming language” [Wall 1999]. “Perl doesn’t have any agenda at all, other than to be maximally useful to the maximal number of people. To be the duct tape of the Internet, and of everything else.”

The Perl way is to accept the status quo of evolved chaos and build upon it using duct tape and ingenuity. Taken to the extreme, a programming system becomes no longer properly speaking a system, but rather a toolkit for improvising assemblages of found software. This perspective may reveal a path between the Scylla of compatibility and the Charybdis of coherence. It has appealing connections to the philosophies of Postmodernism and Pluralism. Unfortunately, efforts in this direction have so far had limited success.

4.3.3 Examples.

- *Unix and C*: Unix rapidly spread across many hardware architectures because C served as a portable assembly language. Porting Unix requires only bootstrapping a C compiler and writing some device drivers. The tradeoff is that C is very weakly specified, with many platform dependencies and undefined behaviors.
- *The Unix Programming Environment*: Unix establishes many standard formats and APIs for modularly constructing systems, and a set of utilities and conventions for program development. Examples are the Posix APIs, the ELF executable file format, the text piping constructors of the shell, and the infrastructure of programming tools like `vi` and `make`. The tradeoff is that these fixed boundaries stifle innovation. It is difficult to build a cross-process datastore other than a file system. Programming languages are limited to transforming textual source files into binary executables.
- *The Web*: HTTP endpoints have proven to be an even more adaptable and viral abstraction than Unix files. They operate at a similar level of abstraction as files, but support richer content and encompass internet-wide interactions between autonomous systems. In a sense HTTP GET and PUT have become the “subroutine calls” of an internet-scale programming system. The most surprising thing about the web is that its usefullness came as a surprise to everyone involved in designing it or competing with it.
- *Smalltalk and Lisp machines*: These classical systems offered a conceptually coherent and self-sufficient programming world using a single programming language throughout. This coherence offered great leverage to experts, many of whom now swear that nothing since has come close. The tradeoff was that these systems were isolated from the rapidly evolving mainstream of heterodox programming. Early implementations of Smalltalk ran on custom hardware, and were ported to standard systems at a very low level, taking over control of disks and displays at the hardware level. Several companies and universities built Lisp machines. Later implementations of Smalltalk and Lisp improved interoperability with the outside world, but still as a somewhat second-class citizen.
- *C++*: C++ added to C the OO (Object Oriented) concepts pioneered by Smalltalk, while remaining 100% compatible with C, down to the level of ABI and performance. This strategy was enormously successful for adoption, but came with the tradeoff of enormous complexity compared to languages designed from scratch for OO, like Smalltalk, Ruby, and Java.
- *Perl and Python*: Perl claims to be a *postmodern* programming language in which “There is more than one way to do it”, enabling idiosyncrasy. In Python the principle that “There should be one—and preferably only one—obvious way to do it” promotes community consensus on a single coherent style.
- *Imperative vs Functional Programming*: Imperative programming embraces the hardware capability to write anywhere in memory through any pointer, whereas functional programming creates a higher-level and safer world of immutable data at the cost of more awkward interaction with the external stateful world.

4.3.4 Relations.

Factoring of complexity (Section A.6): Normalizing redundancy across interrelated descriptions improves coherence.

5 CONCLUSIONS

There is a renewed interest in developing new programming systems. Such systems go beyond the simple model of code written in a programming language using a more or less sophisticated text editor. They combine textual and visual notations, create programs through rich graphical interactions, and challenge accepted assumptions about program editing, execution and debugging. Despite the growing number of novel programming systems, it remains difficult to evaluate the novelty of programming systems and see how they improve over work done in the past. To address the issue, we proposed a framework of "technical dimensions" that captures essential characteristics of programming systems in a qualitative but rigorous way.

The framework of technical dimensions puts the vast variety of programming systems, past and present, on a common footing of commensurability. This is crucial to enable the strengths of each to be identified and, if possible, combined by designers of the next generation of programming systems. As more and more systems are assessed in the framework, a picture of the space of possibilities will gradually emerge. Some regions will be conspicuously empty, indicating unrealized possibilities that could be worth trying. In this way, a domain of "normal science" is created for the design of programming systems.

6 ACKNOWLEDGEMENTS

We would like to thank Richard Gabriel for his detailed comments and shepherding of the drafts for this paper. We also thank the committee of the *Pattern Languages of Programming (PLoP)* conference, who enabled such a collaboration for our submission. We have additionally benefited from discussions with members of the *Temporary Computing Collective* during its regular reading group and elsewhere.

A APPENDIX

The rest of our current set of technical dimensions are less complete than the three we selected above. We list them here to give an impression of the wider space we have hinted at.

A.1 Information loss

A.1.1 Summary. At which points in the system is information irreversibly destroyed or scrambled, and which operations are reversible?

A.1.2 Description. A piece of computer memory can typically be overwritten any number of times. Such writes, by default, do not set aside and save the old value beforehand; thus, information destruction is the norm. Conceptually, each cell has a history of different values, but practically only its latest value exists.

Destructive overwrites occur frequently, from ordinary incrementation of a counter to re-use of garbage-collected or freed memory blocks.

Dimension: bi-directionality. Computation ordinarily proceeds uni-directionally: complex expressions are gradually reduced down to simpler values, or subroutines are invoked on input parameters to produce an output. Furthermore, intermediate values are quickly destroyed as in discarded stack frames or heap blocks.

Instance: provenance. When data is computed by code, a user stepping through in a debugger might know what code computed the data, but otherwise this is unknowable from the data itself. A time when this might be useful would be tracking down a bug in procedurally drawn graphics. If a shape is being drawn in the wrong way, it would be convenient if each shape had been tagged with, say, the line number of its draw call. Then one could click on the shape and immediately narrow down the relevant code.

A.2 Customizability

A.2.1 Summary. Considering an existing program created using the programming system, what aspects of the program can be extended and modified and in what ways is this achieved?

A.2.2 Description. Programming is a gradual process. We start either from nothing or from an existing program and gradually extend and refine it until it serves a given purpose. Programs created using different programming systems can be refined to a different extent, in different ways, at different stages of their existence.

Consider three different examples. First, a program in a conventional programming language like Java can be refined only by modifying its source code. However, you may be able to do so by just adding new code, for example a new interface implementation. Second, a spreadsheet can be modified at any time by modifying the formulas or data it contains. There is no separate programming phase. However, you have to modify the formulas directly in the cell—there is no way of modifying it by specifying a change in a way that is external to the cell. Third, a programmable editor such as Emacs can be extended at runtime by writing extensions that are separate from the core system but that extend its functionality through various provided extension points.

The way a program created using a particular programming system is customizable can be illuminated through the following three characteristics.

Dimension: staging of customization. For programming systems that distinguish between different stages, e.g. writing source code and running a program, the ways in which a program can be customized may be different for each stage. In traditional programming languages, customization is done by modifying or adding source code at the programming stage, but there is no (automatically provided) way of customizing the created programs once they are running.

In systems where a (strict) stage distinction does not exist, programs can be customized not just during development, but also while running. For example, when running a Smalltalk program from an image that includes the programming environment, users of a program can open this and customize the program by modifying the logic behind it.

There are a number of interesting questions related to staging of customization. First, what is the notation used for customization? This may be the notation in which a program was initially created, but a system may also use a secondary notation for customization (e.g., Emacs using Emacs Lisp). For systems with a stage distinction, an important question is whether such changes are *persistent*. For example, modifying a DOM of a web page returned from a web application through Developer Tools is not normally a persistent change. Webstrates [Klokmose et al. 2015], however, makes exactly this kind of modification persistent by synchronizing the DOM between the server and the client. Teitelman's PILOT system for Lisp [Teitelman 1966] offers an interactive way of correcting errors when a program fails. Such corrections are then applied to the source code of the running program.

Dimensions: externalizability and addressibility. A program is typically represented through some structure. This may be its source code (in conventional programming languages), a graph structure in memory (e.g., object graph in a Smalltalk image) or perhaps a document (as in spreadsheet or a Boxer program). When customizing a program, an interesting question is whether a customization needs to be done by modifying the original structure (*internally*) or whether it can be done by *adding* something alongside the original structure, *externally* [Basman et al. 2016].

Examples of the former abound. A case of the latter is, for example, the Cascading Style Sheets (CSS) mechanism on the web. Given a web page, it is possible to modify (almost) any aspects of its look simply by *adding* additional rules to a CSS file. A similar effect is achieved in object-oriented programming where functionality can be added to a system by defining a new class (although injecting the new class into existing code without modification requires some form of configuration such as a dependency injection container).

When customizing programs through *external* means, the customization is done by *addressing* some aspect in the existing program and specifying additional or replacement behaviour. An important issue is how are such addresses specified and what extension points in the program can they refer to. The programming system may offer an automatic mechanism that makes certain parts of a program addressable or this task may be delegated to the programmer. In Aspect Oriented Programming (AOP) systems such as AspectJ, it is possible to add functionality to the invocation of a specific method (among other options) by using the *method call pointcut*.

A.2.3 Examples.

- *Smalltalk, Newspeak and similar*: in programming systems derived from Smalltalk, there is generally no strict distinction between stages and so a program can be customized during execution in the same way as during development. This is typically done *internally*, i.e., by navigating to a suitable object or a class (which serve as the addressable extension points) and modifying that. Lisp-based systems such as *Interlisp* follow a similar model.
- *Extensible text editors* including vim, Emacs and Atom: although those are not (exactly) programming systems, they serve as an interesting example. They can be modified during execution. In some, this is done using a secondary notation (explicitly designed for this purpose), but e.g., in Atom the customization is done through the same mechanism in which it is implemented (JavaScript). The customization is done *externally* by writing an extension separate from the editor itself and it can address extension points offered by the editor. For example, in case of Atom, this is an explicitly defined API, although the dynamic nature of JavaScript (and CSS) makes it possible to address other (less directly exposed) aspects of the system.
- *Cascading Style Sheets*: Although not a programming system, CSS is a prime example of a system that offers external customizability with rich addressability mechanisms that is partly automatic (e.g., when referring to tag names) and partly manual (e.g., when using element IDs and class names). The Infusion project[13] is a research programming system that offers similar customizability mechanism, but for behaviour rather than just styling.
- *Document Object Model (DOM) and Webstrates*: In the context of web programming, there is traditionally a stage distinction between programming (writing the code and markup) and running (displaying a page). However, the DOM can be also modified by browser Developer Tools—either manually, by running scripts in a console, or by using a userscript manager such as Greasemonkey. Such changes are not persistent except in Webstrates [Klokmose et al. 2015], which automatically synchronizes DOM edits to the server and to all other clients viewing the page/application.
- *Object and Aspect Oriented Programming*: in conventional programming languages, customization is done by modifying the code itself. OOP and AOP make it possible to do so *externally* by adding code independently of existing program code. In OOP, this requires manual definition of extension points; AOP improves on this by providing a richer addressing mechanism.
- *Subtext [Edwards 2005] and Boxer [diSessa and Abelson 1986]* are two example research programming systems that do not have explicit stage distinctions. Both of them represent programs as a document that is visible to the user in its entirety (called *naive realism* in Boxer), allowing the user to customize the program by finding the relevant aspect in the document and modifying it.

[13]https://fluidproject.org/infusion.html

A.2.4 Relations.

- *Factoring of complexity* (Section A.6): related in that "customizability" is a form of creating new programs from existing ones; factoring repetitive aspects into a reusable standard component library facilitates the same thing.
- *Interaction structure* (Section 4.1): this determines whether there are separate stages for running and writing programs and may thus influence what kind of customizability is possible.

A.3 Level of automation

A.3.1 Summary. To what extent and in what ways does the programming system remove the need to specify the program implementation in minute detail?

A.3.2 Description. In order to execute a computer program, computers require a full and exact specificiation of the instructions to run. Ever since the 1940s, programmers envisioned that some form of "automatic programming" will alleviate the need for repeatedly specifying all details. The computer still requires full details today, but many aspects of the task of programming can be automated.

Automation can take a number of forms. Extracting common functionality into a library may be merely good use of *factoring of complexity* (Section A.6), but to the user of the library, this may appear as automation. In high-level programming languages, many details are also omitted; those are filled in by the compiler. Finally, the program may also be executed by a more sophisticated runtime that fills in details not specified explicitly, such as when running an SQL query.

Remark: notations. Even with high-level of automation, programming involves manipulating some program notation. In high-level functional or imperative programming languages, the programmer writes code that typically has clear operational meaning. When using more declarative programming like SQL, Prolog or Datalog, the meaning of a program is still unambiguous, but it is not defined operationally—there is a (more or less deterministic) inference engine that solves the problem based on the provided program. Finally, systems based on programming by example step even further away from having clear operational meaning—the program may be simply a collection of sample inputs and outputs and a (typically non-deterministic) engine infers the minute details of a program based on those.

Dimension: degrees of automation. This analysis suggests that there are many degrees of automation in programming systems. The basic mechanism is however always the same—given a program, some logic is specified explicitly and some is left to a reusable component that can specify the details. In the case of library reuse, the reusable component is just the library. In the case of higher-level programming languages, the reusable component may include a memory allocator or a garbage collector. In case of declarative languages or programming by example, the reusable component is a general purpose inference engine.

It is worth noting that higher levels of automation require significantly more complex *reusable components* than lower levels. This is a difference between *level of automation* and *factoring of complexity*—producing systems with higher level of automation requires more than simply extracting (factoring) existing code into a reusable component. Instead, it requires doing more work and introducing a higher level of indirection between the program and the reusable component.

There is also an interesting (and perhaps inevitable?) trade-off. The higher the level of automation, the less explicit the operational meaning of a program. This has a wide range of implications. Smaragdakis [Smaragdakis 2019] notes, for example, that this means the implementation can significantly change the performance of a program.

Remark: fragmentation. An interesting issue is that reusable components that enable higher levels of automation are often specific to each system. This, arguably, limits what we can achieve as components that enable higher-levels of automation are increasingly complex to implement. As noted in [Kell 2009, 2013], incompatible reusable components that exist for multiple systems also limit compositionality. One possible exception from the rule is the Z3 theorem prover, which is used as an implementation mechanism by multiple programming systems including F#.

Instance: next-level automation. Throughout history, programmers have always hoped for the next level of "automatic programming". As observed by Parnas [Parnas 1985], "automatic programming has always been a euphemism for programming in a higher-level language than was then available to the programmer".

We may speculate whether Deep Learning will enable the next step of automation. However, this would not be different in principle from existing developments. We can see any level of automation as using *artificial intelligence* methods. This is the case for declarative languages or constraint-based languages—where the inference engine implements a traditional AI method (GOFAI, i.e., Good Old Fashioned AI).

A.3.3 Relations.
Factoring of complexity: One typically automates the thing at the lowest level in one's factoring (by making the lowest level a thing that exists outside of the program—in a system or a library)

A.4 Handling of errors

A.4.1 Summary. What does the system consider to be an error, and how does it approach their prevention and handling?

A.4.2 Description. In general, a *program error* is when the system cannot run in a normal way and needs to resolve the situation. This raises a number of questions for system design: What can cause a program error? Which program errors can be prevented from happening? How should the system react to a program error?

A computer system is not aware of human intentions. There will always be human mistakes that the system cannot recognize as errors. However, there are many human mistakes that a system can recognize. The design of a system can determine what human mistakes can be detectable program errors.

We distinguish between four kinds of errors: slips, lapses, mistakes and failures. A *slip* is a unintended error caused by a human attention failure such as a typo in source code. A *lapse* is also an unintended error, but caused e.g., by a memory failure such as an incorrectly remembered method name. A *mistake* is a logic error of the human such as bad design of an algorithm. Finally, a *failure* is a system error caused by the system itself that the programmer has no control over, e.g. a hardware or a virtual machine failure.

Dimensions: error detection and prevention. Errors can be identified in any of the *feedback loops* that the system implements. This can be done either by a human or the system itself, depending on the nature of the feedback loop. Consider three examples. First, in live programming systems, the programmer immediately sees the result of their code changes. Error detection is done by a human and the system can assist this by visualizing as many consequences of a code change as possible. Second, in a system with static checking feedback loop (e.g. syntax checking, static type system), potential errors are reported as the result of the analysis. Third, errors can be detected when the developed software is run, either when it is tested by the programmer or when it is run by a user.

Error detection in different feedback loops is suitable for detecting different kinds of errors. Many slips and lapses can be detected by the static checking feedback loop, although this is not always the case. For example, a more compact *expression geography* can make it easier for slips and lapses to produce hard to detect errors.

Mistakes are easier to detect through live feedback loop, but they can also be partly detected by more advanced static checking.

A common aim is to eliminate *latent errors*, i.e. errors that occur at some earlier point during execution, but will manifest themselves through an unexpected behaviour later (for example, dereferencing an incorrect memory address to obtain a value that will be stored in a database and accessed later). Latent errors can be prevented differently in different feedback loops. In the live feedback loop, this can be done by visualizing effects that would normally remain hidden. When running software, latent errors can be prevented through a mechanism that detects errors as early as possible (e.g. initializing pointers to `null` and stopping on `null` dereferencing).

Dimension: error response. When an error is detected by a system, there are a number of typical ways in which the system can respond. The following applies to systems that admit some kind of error detection during execution.

- A system may attempt to automatically recover from the error as best as possible. This may be possible for simpler errors (slips and lapses), but also for certain mistakes (e.g., a mistake in an algorithm's concurrency logic may often be resolved by restarting the code).
- A system may proceed as if the error did not happen. This can eliminate expensive checks, but may lead to latent errors later.
- A system may ask a human how to resolve the issue. This can be done interactively, by entering into a mode where the code can be corrected, or non-interactively by stopping the system.

Orthogonally to the above options, a system may also have a way to recover from latent errors by tracing back through the execution in order to find the root cause. It may also have a mechanism for undoing all actions that occurred in the meantime, e.g. through transactional processing.

A.5 Background knowledge

A.5.1 Summary. What background knowledge does the system demand in order to be judged on its own merit?

A.5.2 Description. Every system rests on some requirement of background knowledge in order to be *learnable* in the first place. Some requirements, like literacy in a written natural language or familiarity with the de facto GUI conventions of modern computing, are easy to miss because they are so ubiquitous. But above the bare necessities for software in general, each system will sit on its own specific foundations that are deemed *outside of its scope* for teaching the user. These can range from widely known skills like the above, to specialist topics in mathematics and sciences, or even the unwritten cultural history of the system itself.

Background knowledge is not only necessary to get *benefit* from *using* the system; it is also necessary to be able to give it a truly fair *evaluation.* If someone without the necessary background attempts and fails to get the use promised by the system, this is not a point against the system (although the fact that it *does* depend on this prerequisite may be criticized). On the other hand, someone who is familiar with the background is in a position to judge the learnability of the system.

The matter is complicated, however, by the fact that a system may not give any clear outward indication of what it expects users to already know. Instead this will have to be *inferred* along the way, making it unclear whether difficulties should be attributed to newly revealed prerequisites, or to shortcomings of the system design.

Thus, to the extent that a system explicitly sets out its background knowledge, it may be judged on whether it works "as advertised". In the more common case of this being implicit, this will invite a complex debate involving what the system is for and what can be inferred about its prerequisites.

A.5.3 Examples.

- As mentioned, familiarity with GUI items like windows, menus, buttons and text fields and their use via keyboard and mouse (or touch) is so ubiquitous a requirement for software that one may wish to declare it *universal.* The only possible exception to this may be Bret Victor's DynamicLand, which relies primarily on everyday motor skills and familiarity with physical objects. However, it appears DynamicLand is programmed in a textual programming language, so keyboard (as well as programming) familiarity may be required even there.
- While the GUI is the ubiquitous background requirement for most "end-user"-facing software and also much "programming" software, programming has its own ubiquitous assumed base: the concepts and conventions of the *UNIX operating system* (more precisely, of UNIX *at the latest*—no doubt, it imported many concepts and conventions from its predecessors and contemporaries): hierarchical file systems; files as streams and vice versa; command-line argument spelling conventions; parsing and serialising of ASCII renderings of structured data.
- *Spreadsheets* are notable for embodying a dataflow computation model on a greater proportion of GUI background to traditional programming:
  - Instead of a textual rendering of symbolic names for variables, spreadsheets present a 2D matrix analogous to computer memory, with spatially "addressed" locations instead of named ones. This substitutes vision and spatial reasoning for abstract name management.
  - Formulas, at their simplest, rest on basic arithmetical knowledge along with two extras: the de-facto ASCII names for multiplication and division (`*` and `/`), and the syntax parsing rules of traditional programming.
  - More complex formulas require using functions in a semi-mathematical notation; this leverages the mathematical concept of functions and its typical notation. However, as tasks move beyond the arithmetically simple, even spreadsheets cannot avoid gradually importing more and more concepts of traditional programming (conditionals, sequencing, arrays, etc.)

A.6 Factoring of complexity

A.6.1 Summary. What are the primitives? How can they be combined? How is *common structure* recognized and utilized?

A.6.2 Description. There is a large space of possible things we might want to do with a system. The question is, how do we "get" to all the possible locations in this space? We can have a very flat structure where different tools, methods or features are used to reach individual points in this space. We can also have a more structured approach where we need to compose individual components to "get" to individual locations.

Dimension: composability. In short, *you can get anywhere by putting together a number of smaller steps.* There exist building blocks which span a range of useful combinations. Such a property can be analogized to *linear independence* in mathematical vector spaces: a number of primitives (basis vectors) whose possible combinations span a meaningful space. Composability is, in a sense, key to the notion of "programmability" and every programmable system will have some level of composability (e.g. in the scripting language.)

Dimension: convenience. In short, *you can get to X, Y or Z via one single step.* There are ready-made solutions to specific problems, not necessarily generalizable or composable. Convenience often manifests as "canonical" solutions and utilities in the form of an expansive standard library.

Composability without convenience is a set of atoms or gears; theoretically, anything one wants could be built out of them, but one must do that work.

Composability *with* convenience is a set of convenient specific tools *along with* enough components to construct new ones. The specific tools themselves could be transparently composed of these building blocks, but this is not essential. They save users the time and effort it would take to "roll their own" solutions to common tasks.

Dimension: commonality. Humans can see Arrays, Strings, Dicts and Sets all have a "size", but the software needs to be *told* that they are the "same". Commonality like this can be factored out into an explicit structure (a "Collection" class), analogous to database *normalization.* This way, an entity's size can be queried without reference to its particular details: if `c` is declared to be a Collection, then one can straightforwardly access `c.size`.

Alternatively, it can be left implicit. This is less upfront work, but permits instances to *diverge*, analogous to *redundancy* in databases. For example, Arrays and Strings might end up with "length", while Dict and Set call it "size". This means that, to query the size of an entity, it is necessary to perform a case split according to its concrete type, solely to funnel the diverging paths back to the commonality they represent:

```
if (entity is Array or String)  size := entity.length
else if (entity is Dict or Set) size := entity.size
```

Remark: flattening and factoring. Data structures usually have several "moving parts" that can vary independently. For example, a simple pair of "vehicle type" and "color" might have all combinations of (Car, Van, Train) and (Red, Blue). Here, we can programmatically change the color directly: `pair.second = Red` or `vehicle.colour = Red`.

In some contexts, such as class names, a system might only permit such multi-dimensional structure as an *exhaustive enumeration*: RedCar, BlueCar, RedVan, BlueVan, RedTrain, BlueTrain, etc. The system sees a flat list of atoms, whereas the user can see the sub-structure encoded in the string. In this world, we cannot simply "change the color to Red" programmatically; we would need to case-split as follows:

```
if (type is BlueCar) type := RedCar
else if (type is BlueVan) type := RedVan
else if (type is BlueTrain) type := RedTrain
...
```

Thus we say there is implicit structure here that remains *un-factored*, similar to how numbers can be expressed as singular expressions (16) or as factor products (2,2,2,2).

A.6.3 Examples. In relational databases, there is an opposition between *normalization* and *redundancy.* When data is duplicated in order to fit into a flat table structure, there is the corresponding redundancy. When data is factored into small tables as much as possible, such that there is only one place each piece of data "lives", the database is in *normal form* or *normalized.* Redundancy is useful for read-only processes, because you do not need joins. But it makes writing risky, because you need to modify one thing in multiple places.

A.7 Abstraction mechanisms

A.7.1 Summary. What is the relationship between concrete and abstract in the system? What are the different abstraction mechanisms supported by the system? How are abstract entities created from concrete things and vice versa?

A.7.2 Description. Abstraction refers to deriving general concepts from specific examples. An *abstraction* is the result of such process. Almost all programming systems support some kind of abstraction. This is necessary, because we often want to instruct the computer to perform the same operation on multiple different inputs or produce many things using the same schema. This dimension captures what is the nature of *abstractions* used in the programming system, how are they created, modified and used.

Dimension: abstraction construction. As suggested above, abstractions can be constructed from concrete examples, first principles or through other methods. A part of the process may happen in the programmer's brain—for example, they can think of concrete cases and use that to come up with an abstract concept and then directly encode the abstract concept in the programming system. But a programming system can directly support different ways of producing abstractions.

One option is to construct abstractions *from first principles*. Here, the programmer starts by defining an abstract entity such as an interface in object-oriented programming languages. To do this, they have to think what the required abstraction will be (in the brain) and then encode it (in the system).

Another option is to construct abstractions *from concrete cases*. Here, the programmer uses the system to solve one or more concrete problems and, when they are satisfied, the system guides them in creating an abstraction based on their concrete case(s). This can be done in a programming language (e.g. through the "extract function" refactor, or by adding a parameter), but also through other approaches (e.g. a form of macro recording)

A.7.3 Examples.

- *Pygmalion*: In Pygmalion [Smith 1975], all programming is done by manipulating concrete icons that represent concrete things. To create an abstraction, you can use "Remember mode", which records the operations done on icons and makes it possible to bind this recording to a new icon.
- *Jupyter notebook*: In Jupyter notebooks, you are inclined to work with concrete things, because you see previews after individual cells. This discourages creating abstractions, because then you would not be able to look inside at such a fine grained level.
- *Spreadsheets*: Up until the recent introduction of lambda expressions into Excel, spreadsheets have been relentlessly concrete, without any way to abstract and reuse patterns of computation other than copy-and-paste.

A.8 Learnability and sociability

A.8.1 Summary. How does the system facilitate or obstruct adoption by both individuals and communities? Individual adoption is facilitated by designing the system to be easily learned by a targeted audience. Community adoption is less about technical issues and more about economic/social/cultural factors that are harder to design for. One important factor is the tenor of online communities, whose norms are best established at the start.

A.8.2 Description. Human and social factors greatly affect the extent to which a technology is used. Even the simplest software technologies require substantial effort to learn. All else equal, a technology that is easier to learn is more likely to be adopted. Individuals with free choice will decide whether the learning investment is worth the benefits. We can make a technology more learnable in several ways:

- Specialize to a specific domain of usage or population of users.
- Design for teaching beginners rather than professional usage.
- Stage levels for learners to advance through.
- Provide more scaffolding and faster feedback in the programming experience.
- Reduce the number of things that need to be learned by unifying concepts (See *conceptual structure*, Section 4.3).

However social factors can easily override individual ones. Our increasing appreciation of the importance of social factors leads us to discuss software technolgies in terms of communities. The nature of the community can be the decisive factor in adopting a technology. How big and old is the community? Do we feel safe that future problems will be solved and new requirements will be met? Do we feel the community shares our values and that we belong in it? Does it give us purpose and meaning? We cannot claim to have a good understanding of how to design a sociable programming system, but we do observe a number of factors that have been important historically:

- Easy integration into existing technology stacks, allowing incremental adoption, and also easy exit if needed.
- Contrary to the previous point, a cloistered community that turns its back on the wider world can give its members strong feelings of belonging and purpose. Examples are Smalltalk, Racket, Clojure, Haskell. These communities bear some resemblance to cults, including guru-like leaders. As such we can expect them to last a long time, but to always remain a fringe counter-culture to the mainstream.
- Backing by large corporations or large capital investments to ensure longevity.
- Making the limits of the technology clear up front, so that adopters are not led astray. For example, no one expects spreadsheets to create web apps. On the other hand, many "no-code" systems have vague and shifting limits.
- Easy sharing of code via package repositories or open exchanges. Prior to the open source era, commercial marketplaces were important, like VBX components for VisualBasic.
- Avoiding fragmentation into sub-communities by establishing standard libraries and practices. Scheme and Haskell are counter-examples. See the Lisp Curse [Winestock 2011].
- Establishing an online community that is welcoming and supportive of newcomers. This is best done by setting community norms from the beginning, as they can be intractable to change later. A good example of this is Elm.

A.8.3 Examples.

- *Spreadsheets, HyperCard*: Small self-sufficient systems that can be mastered in a matter of months.
- *Logo, Boxer, Basic, Pascal, Scratch*: Designed primarily for teaching, prioritizing simplicity.
- *HyperCard, Racket*: Levels of progressively increasing complexity to assist learning
- *Live programming research*: this pursues the dream of relieving programmers from having to simulate execution in their head.
- *Python, Ruby, Elm*: languages that prioritize friendliness to beginners, both technically and in community norms. Leadership that is self-effacing.
- *Lisp, Scheme, Haskell*: Math-like languages appealing strongly to certain people but repelling the general public, along with a culture that celebrates the divide.
- *Software ecosystems*: VisualBasic VBX marketplace. Package managers: CPAN, RubyGems, npm. Scratch website.
- *Elm*: an intentionally sociable language. Friendly error messages, a dramatic usability improvement over other ML-like languages—not technically hard, but other language communities never cared enough to fix it. Principled exclusion of unsafe features that can cause runtime errors, to the point of breaking external interoperability. Prioritized establishing friendly community norms from the beginning.

REFERENCES

Antranig Basman, L. Church, C. Klokmose, and Colin B. D. Clark. 2016. Software and How it Lives On: Embedding Live Programs in the World Around Them. In *PPIG*.

FP Brooks. 1995. Aristocracy, Democracy and System Design. In *The Mythical Man Month: Essays on Software Engineering*. Addison-Wesley.

Hasok Chang. 2004. *Inventing temperature: Measurement and scientific progress.* Oxford University Press, Oxford.

A. A diSessa and H. Abelson. 1986. Boxer: A Reconstructible Computational Medium. *Commun. ACM* 29, 9 (Sept. 1986), 859–868. https://doi.org/10.1145/6592.6595

Jonathan Edwards. 2005. Subtext: Uncovering the Simplicity of Programming. *SIGPLAN Not.* 40, 10 (Oct. 2005), 505–518. https://doi.org/10.1145/1103845.1094851

Jonathan Edwards, Stephen Kell, Tomas Petricek, and Luke Church. 2019. Evaluating programming systems design. In *Proceedings of 30th Annual Workshop of Psychology of Programming Interest Group* (Newcastle, UK) *(PPIG 2019)*.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2018. A Programmable Programming Language. *Commun. ACM* 61, 3 (Feb. 2018), 62–71. https://doi.org/10.1145/3127323

John Foderaro. 1991. LISP: Introduction. *Commun. ACM* 34, 9 (Sept. 1991), 27. https://doi.org/10.1145/114669.114670

Richard P. Gabriel. 1991. *Worse Is Better.* https://www.dreamsongs.com/WorseIsBetter.html

Richard P. Gabriel. 2008. Designed as Designer. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications* (Nashville, TN, USA) *(OOPSLA '08)*. Association for Computing Machinery, New York, NY, USA, 617–632. https://doi.org/10.1145/1449764.1449813

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* (Tucson, Arizona, USA) *(Onward! 2012)*. Association for Computing Machinery, New York, NY, USA, 195–214. https://doi.org/10.1145/2384592.2384611

Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Mass.

T. R. G. Green and M. Petre. 1996. Usability Analysis of Visual Programming Environments: a 'cognitive dimensions' framework. *JOURNAL OF VISUAL LANGUAGES AND COMPUTING* 7 (1996), 131–174.

Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology* (New Orleans, LA, USA) *(UIST '19)*. Association for Computing Machinery, New York, NY, USA, 281–292. https://doi.org/10.1145/3332165.3347925

Daniel Ingalls. 1981. *Design Principles Behind Smalltalk.* https://archive.org/details/byte-magazine-1981-08/page/n299/mode/2up

Stephen Kell. 2009. The mythical matched modules: overcoming the tyranny of inflexible software construction. In *OOPSLA Companion.*

Stephen Kell. 2013. The Operating System: Should There Be One?. In *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems* (Farmington, Pennsylvania) *(PLOS '13)*. Association for Computing Machinery, New York, NY, USA, Article 8, 7 pages. https://doi.org/10.1145/2525528.2525534

Stephen Kell. 2017. Some Were Meant for C: The Endurance of an Unmanageable Language. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Vancouver, BC, Canada) *(Onward! 2017)*. Association for Computing Machinery, New York, NY, USA, 229–245. https://doi.org/10.1145/3133850.3133867

Ursula Klein. 2003. *Experiments, Models, Paper Tools: Cultures of Organic Chemistry in the Nineteenth Century.* Stanford University Press, Stanford, CA. http://www.sup.org/books/title/?id=1917

Clemens N. Klokmose, James R. Eagan, Siemen Baader, Wendy Mackay, and Michel Beaudouin-Lafon. 2015. Webstrates: Shareable Dynamic Media. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology* (Charlotte, NC, USA) *(UIST '15)*. Association for Computing Machinery, New York, NY, USA, 280–290. https://doi.org/10.1145/2807442.2807446

John McCarthy. 1962. *LISP 1.5 Programmer's Manual.* The MIT Press.

Stephen L. Michel. 1989. *Hypercard: The Complete Reference.* Osborne McGraw-Hill, Berkeley.

Donald A. Norman. 2002. *The Design of Everyday Things.* Basic Books, Inc., USA.

Dan R. Olsen. 2007. Evaluating User Interface Systems Research. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology* (Newport, Rhode Island, USA) *(UIST '07)*. Association for Computing Machinery, New York, NY, USA, 251–258. https://doi.org/10.1145/1294211.1294256

Cyrus Omar, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues, Jonathan Aldrich, and Matthew A. Hammer. 2017. Toward Semantic Foundations for Program Editors. In *2nd Summit on Advances in Programming Languages (SNAPL 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 71)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 11:1–11:12. https://doi.org/10.4230/LIPIcs.SNAPL.2017.11

David Lorge Parnas. 1985. Software Aspects of Strategic Defense Systems. (1985). http://web.stanford.edu/class/cs99r/readings/parnas1.pdf

A. J. Perlis and K. Samelson. 1958. Preliminary Report: International Algebraic Language. *Commun. ACM* 1, 12 (Dec. 1958), 8–22. https://doi.org/10.1145/377924.594925

Kragen Javier Sitaker. 2016. *The Memory Models That Underlie Programming Languages.* http://canonical.org/~kragen/memory-models/

Yannis Smaragdakis. 2019. Next-Paradigm Programming Languages: What Will They Look like and What Changes Will They Bring?. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Athens, Greece) *(Onward! 2019)*. Association for Computing Machinery, New York, NY, USA, 187–197. https://doi.org/10.1145/3359591.3359739

D. C. Smith. 1975. Pygmalion: a creative programming environment.

Guy L. Steele and Richard P. Gabriel. 1993. The Evolution of Lisp. In *The Second ACM SIGPLAN Conference on History of Programming Languages* (Cambridge, Massachusetts, USA) *(HOPL-II)*. Association for Computing Machinery, New York, NY, USA, 231–270. https://doi.org/10.1145/154766.155373

Steven L. Tanimoto. 2013. A Perspective on the Evolution of Live Programming. In *Proceedings of the 1st International Workshop on Live Programming* (San Francisco, California) *(LIVE '13)*. IEEE Press, 31–34.

Philip Tchernavskij. 2019. *Designing and Programming Malleable Software.* PhD thesis. Université Paris-Saclay, École doctorale nº580 Sciences et Technologies de l'Information et de la Communication (STIC).

Warren Teitelman. 1966. *PILOT: A Step Toward Man-Computer Symbiosis.* Ph.D. Dissertation.

David Ungar and Randall B. Smith. 2013. *The thing on the screen is supposed to be the actual thing.* http://davidungar.net/Live2013/Live_2013.html

Bret Victor. 2012. *Learnable Programming.* http://worrydream.com/#!/LearnableProgramming

Larry Wall. 1999. *Perl, the first postmodern computer language.* http://www.wall.org//~larry/pm.html

Rudolf Winestock. 2011. *The Lisp Curse.* http://www.winestockwebdesign.com/Essays/Lisp_Curse.html