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. This fails to form a coherent body of research, since it is unclear how to build on past work. In the research world, much has been said and done that allows comparison of *programming languages*, yet no similar theory exists 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* which capture the underlying system characteristics and provide a means for conceptualizing and comparing programming systems. Since these characteristics may be compared or advanced independently, it should be easier to talk about programming systems in a way that can be shared and constructively debated rather than relying solely on personal impressions. By providing foundations for more systematic research in this area, we can help the designers of future programming systems to stand, at last, on the shoulders of giants.

ACM Reference Format:
Joel Jakubovic, Jonathan Edwards, and Tomas Petricek. 2022. Technical Dimensions of Programming Systems. In *Proceedings of* . ACM, New York, NY, USA, 32 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

> A systematic presentation removes ideas from the ground that made them grow and arranges them in an artificial pattern.
> — Paul Feyerabend, *The Tyranny of Science*, Polity Press (2011)
>
> Irony is said to allow the artist to continue his creative production while immersed in a sociocultural context of which he is critical.
> — Emmanuel Petit, *Irony or, the Self-Critical Opacity of Postmodernist Architecture*, Yale (2013)

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.) Beyond this, the rise

, 2021,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 artifact 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 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[1] or low-code programming tools like Dark[2] and Glide.[3] 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 this lack of common vocabulary. While we can thoroughly assess programming languages, as soon as we add interaction or graphics into the picture, we get stuck on how 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*

[1] https://replit.com/
[2] https://darklang.com/
[3] https://www.glideapps.com/

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; it is hard for practitioners to situate their work in the context of others' so that subsequent work can improve on it. Our aim is to provide foundations for the study of programming systems that would allow such development.

1. We present the dimensions in detail, organised into related clusters: *interaction*, *notation*, *conceptualization*, *customizability*, *level of automation*, *errors*, *factoring of complexity*, *representation*, and *adoptability*.
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 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 recognize programming languages as a *small region* 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 "Adoptability" (Section 4.9). 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 have ensured that UNIX has a place in our framework.

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[4] that make data analysis more amenable to scientists by combining code snippets and their numerical or graphical output in a convenient document format.

[4] https://jupyter.org/

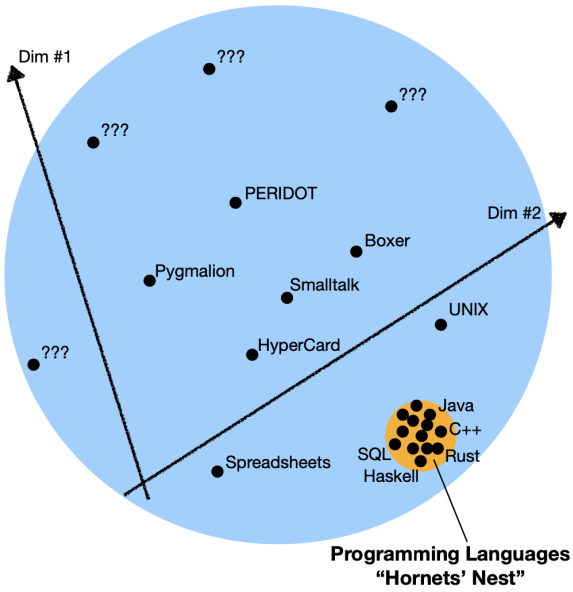

Fig. 1. A speculative sketch of one 2-dimensional slice of the space of possible systems.

- "Low code" end-user programming systems that present a simplified GUI for developing applications. One example is Coda,[5] 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, which creates cloud API services with a "holistic" programming experience including a language and direct manipulation editor with near-instantaneous building and deployment.
- Even for general purpose programming with conventional tools, systems like Replit 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:

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

[5] https://coda.io/welcome

2.3 Characteristics already identified elsewhere

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.
- The *Design Principles of Smalltalk* [Ingalls 1981] documents the philosophical goals and dicta 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[6] 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.

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

3 PROGRAMMING SYSTEMS

We intentionally use the term *programming system* to refer to a broad range of systems which are programmable to a varying degree. This section highlights a number of example families of programming systems, following a roughly chronological order. This serves three 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. Third, studying the difference between systems in individual families is one way of identifying and motivating interesting 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 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-8 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,[7] 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, online IDEs, or virtual machine and container images.

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 human beings of all ages" [Kay and Goldberg 1977]. The authors saw computers as *meta-media* that could become a range of other media for education, discourse, creative arts, simulation and other applications not yet invented. Smalltalk was designed for single-user workstations with a graphical display, and pioneered this display not just for applications but also for programming itself. This evolved over the history of Smalltalk. In Smalltalk 72, one wrote code in the bottom half of the screen. When editing a definition, the window became a

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

structure editor logically similar to that of Lisp, but controlled using a mouse and menu instead of a teletype. Smalltalk 76 completed the transition from a terminal-based interface to a graphical interface and introduced the *class browser* that allowed navigating through the classes that exist in the system and modifying their code.

Smalltalk shared a number of other characteristics with Lisp, although its key concept was one of *objects* and *message passing* rather than *lists*. 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 fact that much of the Smalltalk environment was implemented in itself, a property shared with many later Lisp 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* the system.[8] 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 "Adoptability" (Section 4.9), 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. Although UNIX is historically closely linked to the C programming language, it developed a language-agnostic set of abstractions that make it possible to use multiple programming languages in a single system. 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, not present in Smalltalk or Lisp: UNIX distinguishes between volatile (RAM) structures, which are lost when the system is shut down, and non-volatile (disk) structures that are preserved. The ontology of files, however, enables an open pluralistic environment.

3.2 Application platforms

The previously discussed programming systems were either universal, in that they did not focus on any particular kind of application, or they were focused on broad application areas. For example, Lisp was designed for symbolic data manipulation in the context of Artificial Intelligence whereas the focus of FORTRAN was scientific computing. However, as computers became more widely used, it became clear that there are more narrow typical kinds of applications that need to be built. For those, 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

[8] 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.

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 platforms

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. The boundaries of these *developer platforms* are less well-defined and we acknowledge that the exact delineation we choose significantly affects our analysis.

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.

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. Webstrates make the DOM collaborative, but not (automatically) *live* because of the complexities this implies for event handling.

3.3.2 REPLs and notebooks. Another kind of developer ecosystem which evolved from simple scripting tools consists of modern data science tools, such as Jupyter, whose roots date back to "On-Line Systems" developed in the 1960s. The style was exemplified in 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, an idea pioneered in work on *literate programming* [Knuth 1984].

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 some notebook systems 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 to record the session to a file.)

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 of 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 "Errors" (4.6).

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 proposed technical dimensions grouped under *clusters*. 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 longer *description*, and closes with a list of any *relations* to other clusters along with any *references* if applicable. Within the main description, individual *dimensions* are listed. 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 *examples*. Finally, interspersed discussion that is neither a *dimension* nor an *example* is introduced as a *remark*.

4.1 Interaction

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

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 read through error messages they get. After fixing the code to pass compilation, a similar process might happen with runtime errors.

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].

4.1.1 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*.

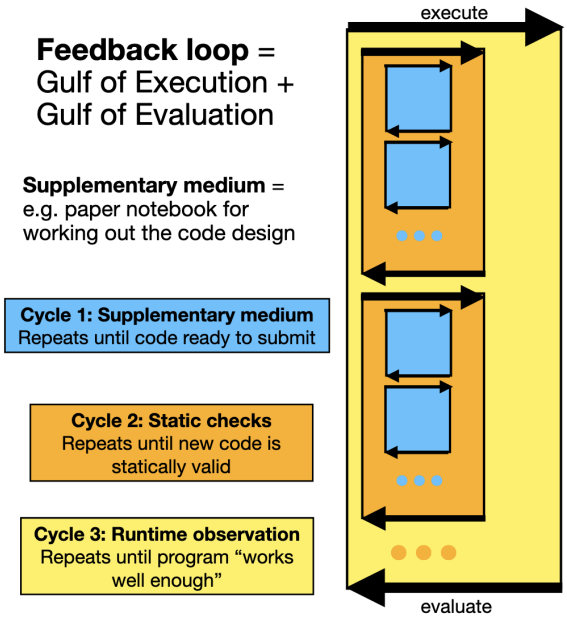

Fig. 2. The nested feedback loops of a statically-checked programming language.

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.

For example, we can analyze statically checked *programming languages* (e.g. Java, Haskell) into several feedback loops (Figure 2):

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 (like 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.

4.1.2 Example: 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*. For convenience, we choose to include the latter criterion—automatic demand of result—in our definition of immediate feedback.

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 generally only get reported *after* the entire line is submitted. This evaluation gulf has been reduced in the JavaScript console of web browsers, where the line is "run" in a limited manner on every keystroke. Thus, simple commands without side-effects (such as calls to pure functions) can give instant previewed results, though partially typed expressions and syntax errors will not trigger previews.

4.1.3 Example: 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. If, on the other hand, 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.

Spreadsheets contain a feedback loop for direct manipulation of values and formatting, as in any other WYSIWYG application. They also contain another 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,

[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!

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.

4.1.4 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.

4.1.5 Relations.

- *Errors* (Section 4.6) 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.
- *Adoptability* (Section 4.9): The *execution* gulf is concerned with software using and programming in general. The time taken to realize an idea in software is affected by the user's familiarity and the system's *learnability*, as well as the *expressive power* of the system's ontology.
- *Representation* (Section 4.8): The motto "The thing on the screen is supposed to be the actual thing" [Pawson 2004], adopted in the context of live programming, suggests that *liveness* also relates to representation. Objects that users interact with should be equipped with faithful behavior rather than being intangible shadows cast by the hidden *real* object.
- *Notation* (Section 4.2): Feedback loops are related to *notational structures*. In a system with multiple notations, each notation may have different associated feedback loops.

4.2 Notation

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

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.

4.2.1 Dimension: notational structure. In practice, most programming systems use multiple notations. Different notations can play different roles in the system. On the one hand, multiple *overlapping notations* can be provided as different ways of programming the same aspects of the system. In this case, each notation may be more suitable to different kind of users, but may have certain limitations (for example, a visual notation may have a limited expressive power). On the other hand, multiple *complementing notations* may be used as the means for programming different aspects of the system. In this case, programming the system requires using multiple notations, but each notation may be more suitable for the task at hand, such as HTML for describing the document structure and JavaScript for specifying its behavior.

565 *4.2.2 Example: overlapping notations.* A programming system may provide multiple notations for programming
566 the same aspect of the system. This is typically motivated by an attempt to offer easy ways of completing different
567 tasks, e.g., a textual notation for defining abstractions and a visual notation for specifying concrete structures.
568 The crucial issue in this kind of arrangement is synchronization between the different notations. If the notations
569 have different characteristics, this is not a straightforward mapping. For example, source code may allow a more
570 elaborate abstraction mechanism (such as a loop) which will appear as visible repetition in the visual notation.
571 What should such a system do when the user edits a single object that resulted from such repetition? Similarly,
572 textual notation may allow incomplete expressions that do not have an equivalent in the visual notation. For
573 programming systems that use *overlapping notations*, we need to describe how the notations are synchronized.
574 *Sketch-n-Sketch* [Hempel et al. 2019] employs overlapping notations for creating and editing SVG and HTML
575 documents. The user edits documents in an interface with a split-screen structure that shows source code on the
576 left and displayed visual output on the right. They can edit both of these and changes are propagated to the other
577 view. The code can use abstraction mechanisms (such as functions) which are not completely visible in the visual
578 editor (an issue we return to in *expression geography* below). Sketch-n-sketch can be seen as an example of a
579 *projectional editor*, although traditional projectional editors usually work more directly with the abstract syntax
580 tree (AST) of a programming language.
581 *UML Round-tripping.* Another example of a programming system that utilizes the *overlapping notations* structure
582 are UML design tools that display the program both as source code and as a UML diagram. Edits in one result in
583 automatic update of the other. An example is the Together/J system.. To solve the issue of notation synchronization,
584 such systems often need to store additional information (e.g. location of classes in UML diagram after the user
585 rearranges them) in the textual notation, typically using a special kind of code comment.
586
587 *4.2.3 Example: complementing notations.* A programming system may also provide multiple complementing
588 notations for programming different aspects of its world. Again, this is typically motivated by the aim to make
589 specifying certain aspects of programming easier, but it is more suitable when the different aspects can be more
590 clearly separated. The key issue for systems with complementing notations is how the different notations are
591 connected. The user may need to use both notations at the same time, or they may need to progress from one to
592 the next level when solving increasingly complex problems. In the latter case, the learnability of progressing
593 from one level to the next is a major concern.
594 *Spreadsheets and HyperCard.* In Excel, there are three different complementing notations that allow users to
595 specify aspects of increasing complexity: (i) the visual grid, (ii) formula language and (iii) a macro language such
596 as Visual Basic for Applications. The notations are largely independent and have different degrees of expressive
597 power. Entering values in a grid cannot be used for specifying new computations, but it can be used to adapt
598 or run a computation, for example when entering different alternatives in What-If Scenario Analysis. More
599 complex tasks can be achieved using formulas and macros. A user gradually learns more advanced notations, but
600 experience with a previous notation does not help with mastering the next one. The approach optimizes for easy
601 learnability at one level, but introduces a hurdle for users to surmount in order to get to the second level. The
602 notational structure of *HyperCard* is similar and consists of (i) visual design of cards, (ii) visual programming (via
603 the GUI) with a limited number of operations and (iii) HyperTalk for arbitrary scripting.
604 *Boxer and Jupyter.* Boxer [diSessa and Abelson 1986] uses *complementing notations* in that it combines a visual
605 notation (the layout of the document and the boxes of which it consists) with textual notation (the code in the
606 boxes). Here, the textual notation is always nested within the visual. The case of Jupyter notebooks is similar. The
607 document structure is graphical (edited by adding and removing cells); code and visual outputs are nested as cells
608 in the document. This arrangement is common in many other systems such as Flash or Visual Basic, which both
609 combine visual notation with textual code, although one is not nested in the other.
610
611

4.2.4 Dimension: primary and secondary notations. In practice, most programming systems use multiple notations. Even in systems based on traditional programming languages, the *primary notation* of the language is often supported by *secondary notations* such as annotations encoded in comments and build tool configuration files. In systems that use the *complementing* or *overlapping* notational structure, multiple notations can be primary.

Programming languages. Programming systems built around traditional programming languages 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 a multi-step mechanism for running C code via the terminal, assisted by secondary notations such as shell scripts. 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]

Haskell. In 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. The idea of having such 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" using Greek letters and subscripts.

4.2.5 Dimension: expression geography[11]*.* 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*? Visual notations may provide a 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.

C/C++ expression language. 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).

4.2.6 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 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, 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 are 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.

[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.

[11] TODO: Add citations. The idea of "expression geography" is due to Antranig, but I cannot find a reference to cite. Similarly "pedantic notations" is due to Thomas Green, but again, I cannot find a citation.

Notations are closely linked to representation in that the notation may mirror the structures used for program representation. For example, 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 system in Lisp, with syntax indistinguishable from Lisp. Moreover, it is possible to write a program that constructs another Lisp program and not only run it interpretively (using the `eval` function) but compile it at runtime (using the `compile` function) and 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]

4.2.7 References. Cognitive Dimensions of Notation [Green and Petre 1996] provide a comprehensive framework for analysing individual notations, while our focus here is on how multiple notations are related and how they are structured. It is worth noting that the Cognitive Dimensions also define *secondary notation*, but in a different sense to ours. For them, secondary notation refers to whether a notation allows including redundant information such as color or comments for readability purposes.

The importance of notations in the practice of science, more generally, has been studied by [Klein 2003] as "paper tools". These are formula-like entities which can be manipulated by humans in lieu of experimentation, such as the aforementioned mathematical notation in Haskell: a "paper tool" for experimentation on a whiteboard. Programming notations are similar, but they are a way of communicating with a machine; the experimentation does not happen on paper alone.

4.2.8 Relations.

- *Factoring of complexity* (Section 4.7): Using multiple complementing notations implicitly factors complexity by expressing different aspects of a program using different notations.
- *Interaction* (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.
- *Adoptability* (Section 4.9): Notational structure can affect learnability. In particular, complementing notations may require (possibly different) users to master multiple notations. Overlapping notations may improve learnability by allowing the user to edit the program in one way (e.g. visually) and see the effect in the other notation (e.g. in code).
- *Representation* (Section 4.8): The two dimensions are related in that the internal representation of program structures may be more or less closely reflected in the notation.

4.3 Conceptualization

What tradeoffs are made between conceptual integrity vs. compatibility with established technologies?

> 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 intricate 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 established by 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.

4.3.1 Dimension: conceptual integrity. Perhaps the apotheosis of conceptual integrity was in early Smalltalk and Lisp machines, which were complete programming systems built around a single language. They incorporated capabilities commonly provided outside the programming language by operating systems and databases. Everything was done in one language, and so everything was represented with the datatypes of that language. The benefit of all code being in one language was that it became a *lingua franca*, reusable across all contexts.

In addition to Smalltalk and Lisp, many programming languages focus on one kind of data structure [Sitaker 2016]:

- In COBOL, data consists of nested records as in a business form.
- In Fortran, data consists of parallel arrays.
- In SQL, data is a set of relations with key constraints.
- In scripting languages like Python, Ruby, and Lua, much data takes the form of string-indexed hash tables.

Many languages are *imperative*, staying close to the hardware model of addressable memory, lightly abstracted into primitive values and references into mutable arrays and structures. On the other hand, *functional* languages hide references and treat everything as immutable structured values. This conceptual simplification benefits certain kinds of programming, but can be counterproductive when an imperative approach is more natural, such as in external input/output.

Python and Perl showcase opposite extremes of 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 a *system*, properly speaking, but rather a *toolkit for improvising* assemblages of *found* software. Perl could be seen as championing the value of Pluralism over Conceptual Integrity. This philosophy has been called *Postmodern Programming* [Noble and Biddle 2004].

4.3.2 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.

Richard Gabriel first described this dilemma in his influential 1991 essay *Worse is Better* [Gabriel 1991] analyzing the defeat of Lisp by UNIX and 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 easily to the evolving conditions of the external world. The tradeoff was decreased conceptual integrity, such as the undefined behaviours of C, the junkyard of working directories, and the proliferation of special purpose programming languages to provide a complete development environment.

Another case is that of C++, which added to C the 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.

4.3.3 Remark: the end of 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 systems have become programming ecosystems — not designed but evolved. How can we noticeably improve programming in the face of 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, with the result that 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.

4.3.4 Relations.

- *Composability* (Section 4.7.1): Conceptual integrity strives to reduce complexity at the source, by creating unified concepts that compose orthogonally to generate diversity.

4.4 Customizability

> *For an existing program created using the system, which aspects can be extended and modified and how is this achieved?*

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, such as 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 self-sustaining programming system, such as Smalltalk, does not make an explicit distinction between "programming" and "using" phases, and it can be modified and extended via itself. It gives developers the power to experiment with the system and, in principle, replace it with a better system from within.

4.4.1 Dimension: staging of customization. For systems that distinguish between different stages, such as writing source code versus running a program, customization methods 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.

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 (consider Emacs using Emacs Lisp). For systems with a stage distinction, an important question is whether such changes are *persistent*.

Smalltalk, Interlisp and similar. In image-based programming systems, there is generally no strict distinction between stages and so a program can be customized during execution in the same way as during development. The program image includes the programming environment. Users of a program can open this, navigate to a suitable object or a class (which serve as the *addressable extension points*) and modify that. Lisp-based systems

such as *Interlisp* follow a similar model. Changes made directly to the image are persistent. The PILOT system for Lisp [Teitelman 1966] offers an interactive way of correcting errors when a program fails during execution. Such corrections are then applied to the image and are thus persistent.

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 in the default browser state, but are made so by Webstrates [Klokmose et al. 2015] which synchronize the DOM between the server and the client.

4.4.2 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 (an 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].

When customizing programs through *external* means, the customization is done by *addressing* some aspect in the existing program and specifying additional or replacement behaviour. Of particular importance is how such addresses are specified and what extension points in the program they can refer to. The system may offer an automatic mechanism that makes certain parts of a program addressable, or this task may be delegated to the programmer.

Cascading Style Sheets (CSS): CSS is a prime example of a system that offers external customizability with rich addressability mechanisms that are partly automatic (when referring to tag names) and partly manual (when using element IDs and class names). Given a web page, it is possible to modify (almost) any aspects of its appearance by simply *adding* additional rules to a CSS file. The Infusion project [Basman 2021] offers similar customizability mechanisms, but for behaviour rather than just styling.

Object Oriented Programming 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, i.e. interfaces and abstract methods. Functionality can then 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). AOP systems such as AspectJ [Kiczales et al. 2001] provides a richer addressing mechanism. In particular, it makes it possible to add functionality to the invocation of a specific method (among other options) by using the *method call pointcut*. This functionality is similar to *advising* in Pilot [Teitelman 1966].

4.4.3 Dimension: self-sustainability. For most programming languages, programming systems, and ordinary software applications, if one wants to customize beyond a certain point, one must go beyond the facilities provided in the system itself. Most programming systems maintain a clear distinction between the *user level*, where the system is used, and *implementation level*, where the source code of the system itself resides. If the user level does not expose control over some property or feature, then one is forced to go to the implementation level. In the common case this will be a completely different language or system, with an associated learning cost. It is also likely to be lower-level—lacking expressive functions, features or abstractions of the user level—which makes for a more tedious programming experience.

It is possible, however, to carefully design systems to expose deeper aspects of their implementation *at the user level*, relaxing the formerly strict division between these levels. For example, in the research system *3-Lisp* [Smith 1982], ordinarily built-in functions like the conditional `if` and error handling `catch` are implemented in 3-Lisp code at the user level.

The degree to which a system's inner workings are accessible to the user level, we call *self-sustainability*. At the maximal degree of this dimension would reside "stem cell"-like systems: those which can be progressively evolved to arbitrary behavior without having to "step outside" of the system to a lower implementation level. In a sense, any difference between these systems would be merely a difference in initial state, since any could be turned into any other.

The other end, of minimal self-sustainability, corresponds to minimal customizability: beyond the transient run-time state changes that make up the user level of any piece of software, the user cannot change anything without dropping down to the means of implementation of the system. This would resemble a traditional end-user "application" focused on a narrow domain with no means to do anything else.

The terms "self-describing" or "self-implementing" have been used for this property, but they can invite confusion: how can a thing describe itself? Instead, a system that can *sustain itself* is an easier concept to grasp. The examples that we see of high self-sustainability all tend to be *Operating System-like*. UNIX is widely established as an operating system, while Smalltalk and Lisp have been branded differently. Nevertheless, all three have shipped as the operating systems of custom hardware, and have similar responsibilities. Specifically: they support the execution of "programs"; they define an interface for accessing and modifying state; they provide standard libraries of common functionality; they define how programs can communicate with each other; they provide a user interface.

UNIX: Self-sustainability of UNIX is owed to the combination of two factors: (1) the system is implemented in binary files (via ELF[12]) and text files (for configuration); (2) these files are part of the user-facing filesystem, so users can replace and modify parts of the system using UNIX file interfaces.

Smalltalk and COLAs: Self-sustainability in Smalltalk is similar to UNIX, but at a finer granularity and with less emphasis on whether things reside in volatile (process) or non-volatile (file) storage. The analogous points are that (1) the system is implemented as objects with methods containing Smalltalk code, and (2) these are modifiable using the class browser and code editor. Combined Object Lambda Architectures, or COLAs [Piumarta 2006], are a theoretical system design to improve on the self-sustainability of Smalltalk. This is achieved by generalizing the object model to support relationships beyond classes.

4.4.4 *References.*

In addition to the examples discussed above, the proceedings of self-sustaining systems workshops [Hirschfeld et al. 2010; Hirschfeld and Rose 2008] provides numerous examples of systems and languages that are able to bootstrap, implement, modify, and maintain themselves; Gabriel's analysis of programming language revolutions [Gabriel 2012] uses *advising* in PILOT, related Lisp mechanisms, and "mixins" in OOP to illustrate the difference between the "languages" and "systems" paradigms.

4.4.5 *Relations.*

- *Factoring of complexity* (Section 4.7): 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* (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.

4.5 Automation

To what extent and in what ways does the system remove the need to spell out implementation in minute detail?

Ultimately, at the hardware level, computers are primitive calculating machines. They require a full and exact specificiation of the instructions to run. Ever since the 1940s, programmers have envisioned that some form of

[12] Executable and Linkable Format.

"automatic programming" will alleviate the need for tediously specifying details at this level. While this level still remains today, many aspects of the task of "programming" can and have been *automated.*

Automation can take a number of forms. Extracting common functionality into a library may be merely good use of *factoring of complexity* (Section 4.7), 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 or using a logic programming system like Prolog.

4.5.1 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 description. 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, from which a (typically non-deterministic) engine infers the concrete steps of execution.

4.5.2 Dimension: degrees of automation. There are many degrees of automation in programming systems, but the basic mechanism is always the same—given a program, some logic is specified explicitly and some is left to a reusable component that can do the rest. 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.

Higher levels of automation require 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.

4.5.3 Example: programming by example. An interesting case of automation is *programming by example* [Lieberman 2001]. In this case, the user does not provide even a declarative specification of the program behavior, but instead specifies sample inputs and outputs. A more or less sophisticated algorithm then attempts to infer the relationship between the inputs and the outputs. This may, for example, be done through program synthesis where an algorithm composes a transformation using a (small) number of pre-defined operations. Programming by example is often very accessible and has been used in spreadsheet applications [Gulwani et al. 2012].

4.5.4 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. For example, the resolution algorithm that is at the core of a Prolog system is typically tightly bound to the particular system and cannot be easily reused by another programming system.

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 Dafny and F*, as well as by numerous program verification tools.

4.5.5 Example: 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).

4.5.6 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)

4.6 Errors

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

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.

Following the standard literature on errors [Reason 1990], we distinguish four kinds of errors: slips, lapses, mistakes and failures. A *slip* is an error caused by transient human attention failure, such as a typo in the source code. A *lapse* is similar but caused by memory failure, such as an incorrectly remembered method name. A *mistake* is a logical error 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.

4.6.1 Dimensions: error detection. 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 a static checking feedback loop (such as syntax checks, static type systems), 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 (manually or through automated testing) 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, consider a "compact" *expression geography* where small changes in code may result in large changes of behaviour. This makes it easier for slips and lapses to produce hard to detect errors. Mistakes are easier to detect through a live feedback loop, but they can also be partly detected by more advanced static checking.

4.6.2 Example: static typing. In statically typed programming languages like Haskell and Java, types are used to capture some information about the intent of the programmer. The type checker ensures code matches the lightweight specification given using types. In such systems, types and implementation serve as two descriptions of programmer's intent that need to align; what varies is the extent to which types can capture intent and the way in which the two are constructed; that is, which of the two comes first.

4.6.3 Examples: TDD, REPL and live coding. Whereas static typing aims to detect errors without executing code, approaches based on immediate feedback typically aim to execute (a portion of) the code and let the programmer see the error immediately. This can be done in a variety of ways.

In case of *test-driven development*, tests play the role of specification (much like types) against which the implementation is checked. Such systems may provide more or less immediate feedback, depending on when tests are executed (automatically in the background, or manually). Systems equipped with a read-eval-print loop (REPL) let programmers run code on-the-fly and inspect results. For successful error detection, the results need to be easily observable: a printed output is more helpful than a hidden change of system state. Finally, in live coding systems, code is executed immediately and the programmer's ability to recognize errors depends on the extent to which the system state is observable. In live coded music, for example, you *hear* that your code is not what you wanted, providing an easy-to-use immediate error detection mechanism.

4.6.4 Remark: eliminating latent errors. A common aim of error detection is to prevent *latent errors*, i.e. errors that occured at some *earlier* point during execution, but only manifest themselves through an unexpected behaviour later on. For example, we might dereference the wrong memory address and store a junk value to a database; we will only find out upon accessing the database. Latent errors can be prevented differently in different feedback loops. In a 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 if they are dereferenced.)

Elm and time-travel debugging. One notable mechanism for identifying latent errors is the concept of *time-travel debugging* popularized by the Elm programming language. In time-travel debugging, the programmer is able to step back through time and see what execution steps were taken prior to a certain point. This makes it possible to break execution when a latent error manifests, but then retrace the execution back to the actual source of the error.

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

- It may attempt to automatically recover from the error as best as possible. This may be feasible for simpler errors (slips and lapses), but also for certain mistakes (a mistake in an algorithm's concurrency logic may often be resolved by restarting the code.)
- It may proceed as if the error did not happen. This can eliminate expensive checks, but may lead to latent errors later.
- It 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.

Interlisp and Do What I Mean (DWIM). Interlisp's DWIM facility attempts to automatically correct slips and lapses, especially misspellings and unbalanced parentheses. When Interlisp encounters an error, such as a reference to an undefined symbol, it invokes DWIM. In this case, DWIM then searches for similarly named symbols frequently used by the current user. If it finds one, it invokes the symbol automatically, corrects the source code and notifies the user. In more complex cases where DWIM cannot correct the error automatically, it starts an interaction with the user and lets them correct it manually.

4.6.6 Relations.

- *Feedback loops*: Error detection always happens as part of an individual feedback loop. The feedback loops thus determine the structure at which error detection can happen.

- *Level of automation:* A semi-automatic error recovery system (such as DWIM) implements a form of automation. The concept of antifragile software [Monperrus 2017] is a more sophisticated example of error recovery through automation.
- *Expression geography:* In an expression geography where small changes in notation lead to valid but differently behaved programs, a slip or lapse is more likely to lead to an error that is difficult to detect through standard mechanisms.

4.6.7 References. The most common error handling mechanism in conventional programming languages is exception handling. The modern form of exception handling has been described in [Goodenough 1975]; [Ryder et al. 2005] documents the history and influences of Software Engineering on exception handling. The concept of *antifragile software* [Monperrus 2017] goes further by suggesting that software could improve in response to errors. Work on Chaos Engineering [Chang et al. 2015] is a step in this direction.

[Reason 1990] analyses errors in the context of human errors and develops a classification of errors that we adopt. In the context of computing, errors or *miscomputation* has been analysed from a philosophical perspective [Floridi et al. 2015; Fresco and Primiero 2013]. Notably, attitudes and approaches to errors also differ for different programming subcultures [Petricek 2017].

4.7 Factoring of complexity

> *What are the primitives? How can they be combined? How is* common structure *recognized and utilized?*

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.

4.7.1 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.)

UNIX shell commands are a standard example of composability. The base set of primitive commands can be augmented by programming command executables in other languages. Given some primitives, one can "pipe" one's output to another's input (`|`), sequence (`;` or `&&`), select via conditions, and repeat with loop constructs, enabling full imperative programming. Furthermore, command compositions can be packaged into a named "script" which follows the same interface as primitive commands, and named subprograms within a script can also be defined.

In *HyperCard*, the user level just before writing code is *non*composable for programming buttons: there is simply a long list of predefined behaviors to choose from.

The *Haskell type system*, as well as that of other functional programming languages, exhibits high composability. New types can be defined in terms of existing ones in several ways. These include records, discriminated unions, function types and recursive constructs (e.g. to define a `List` as either a `Nil` or a combination of element plus other list.) The C programming language also has some means of composing types that are analogous in some ways, such as structs, unions, enums and indeed even function pointers. For every type, there is also a corresponding "pointer" type. It lacks, however, the recursive constructs permitted in Haskell types.

4.7.2 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. This could characterize low-level programming systems or systems in a "blank canvas" state.

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.

For example, let us turn to a convenience factor of *UNIX* shell commands, having already discussed their composability above. Observe that it would be possible, in principle, to pass all information to a program via standard input. Yet in actual practice, for convenience, there is a standard interface of *command-line arguments* instead, separate from anything the program takes through standard input. Most programming systems similarly exhibit both composability and convenience, providing templates, standard libraries, or otherwise pre-packaged solutions, which can nevertheless be used programmatially as part of larger operations.

4.7.3 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
```

4.7.4 Examples: 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). In this *factored* representation, 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, even though a human 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
...
```

The *commonality* between RedCar, RedVan, BlueCar, and so on has been *flattened*. 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). *Factoring* this commonality gives us the original design, where there is a pair of values from different sets.

In *relational databases*, there is an opposition between *normalization* and *redundancy*. In order to fit multi-table data into a *flat* table structure, data needs to be duplicated into redundant copies. 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 there is no need to join different tables together based on common keys. Writing, however, becomes risky; in order to modify one thing, it must be synchronized to the multiple places it is stored. This makes highly normalized databases optimized for writes over reads.

4.7.5 Dimension: abstraction construction. 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 how are they created, modified and used.

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 mind—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 system. But a 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 mind) 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)

- *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.

4.7.6 Relations.

- *Notational structure* 4.2. abstractions are typically done at a level of individual notations.

4.7.7 References.

- How to Design a Good API and Why it Matters [Bloch 2007]

4.8 Representation

> *Which structures are available to programmatic manipulation, and which ones must be extracted by a non-trivial process?*

Representations may be flat sequences or structured graphs. There may be multiple representations of the same thing. What are they and how do they relate?

4.8.1 Dimensions: Source Of Truth and Derived Representation. A human might recognize individual edges and objects in an image, but the image is a mere array of pixels. Similarly, an audio file might consist of a single, long waveform, yet a human listener could easily pick out individual instruments, sounds or words. In each of these cases there is (a) a messy, real-world *source of truth* easily understood by humans yet hard to formalize, and (b) a

derived representation of computer-friendly repeating primitive units. The latter produces comparable stimuli for a human, but is merely a recording of the source of truth in a specific context.

4.8.2 Example: String Typing. The epithet "stringly typed" refers to data with clear sub-structure which has been encoded in a string. This sub-structure may be evident to a human, but all a computer program sees is an opaque list of characters. For example, a citation might be displayed as "Example Paper (John Doe, 1993)". Clearly, it contains the title of a work, an author name, and a year. However, while this may be clear to us, a machine simply sees a list of characters. In order to work with these individual fields, it must be programmed to *extract* them by *parsing* the string.

Alternatively, the citation could be programmatically represented in a structured form, such as a dictionary:

```
{
  title: "Example Paper", year: 1993,
  author: {forename: "John", surname: "Doe"}
}
```

This structure could then be *rendered* into a string for display or output to a document. The code to do this could well be simpler than the code required for the reverse transformation, parsing a string into the structure. The benefit would be that there is straightforward programmatic access to the individual sub-structures, for free, from the beginning.

4.8.3 Examples: Vector and Raster Graphics. Raster graphics organises graphics as a rectangular array of colored pixels, while *vector graphics* works on a tree or graph of shape descriptions. In a classic raster graphics program like MS Paint, a user can can see the pixels resulting from using the curve tool, but can't change the curve afterwards—they can only undo or erase the pixels and start again. In contrast, in a vector graphics tool, the curve's control points and parameters would be always present, only *incidentally* rendering the shape to pixels for display purposes.

This is analogous to the String Typing example. However, the problem of starting from a block of pixels and wishing to *programmatically* work with sub-structure—parsing, so to speak—is much more complicated in this case. Extracting meaningful units of structure from digital images, such as recognizing shapes, is the domain of advanced research in AI and computer vision.

Thus, using a structured representation (vector graphics) for graphic design is the only technically feasible option. In the case of strings, the issue is less obvious due to decades of established techniques in parsing formal languages. The missing link between the two is that formal languages, as far as we are aware, have never been extended to formal structures of pixels in images, where parsing would not need to rely on artificial intelligence yet would still break on incorrect pixel placement.

4.8.4 Examples: Text vs. Structure Editing. There are two ways in which text editing can be analysed for its Source of Truth and Derived Artefact. In the first, we can consider text editing, and word processing more generally, as a special case of Vector Graphics. Howver, in another sense, text editing shares a resemblance to Raster Graphics.

Text editing as Vector Graphics. Text glyphs are shapes like any others, and need rasterizing to pixels on a screen. Text editors, word processors and other publishing software do not require a typist to correct their mistakes by manipulating blocks of pixels. These tools have the user work at the level of *characters* instead of *pixels*. The usual advantages apply, such as scaling to different sizes and easily navigating, selecting and modifying at the character level. Even though text needs rendering to pixels, it is usually inappropriate to manipulate text as pixels.

Text editing as Raster Graphics. On the other hand, what exactly is meant by *text*? Viewing written language as merely a list of characters may be just as naïve as viewing glyphs as pixel blocks. Written language clearly has internal structure. Documents comprise paragraphs and headings. Paragraphs contain sentences, sentences contain clauses, clauses contain words and words themselves contain syllables and hyphenation points. These

same structural categories are concretely rendered with different punctuation characters in different languages. Furthermore, in formal languages such as program code, this property is explicit at the level of its formal grammar. A program might be made up of declarations, containing statements, containing expressions, containing tokens, which are ultimately made up of textual characters.

The possibility of making *typographical errors* in an English document, or *syntax errors* in some source code, is proof that the character level has degrees of freedom that our languages do not have. This is the motivation for *structure editing* as opposed to text editing—the approach of making errors *unrepresentable* in the first place.

Structure editors do for linguistic structures (like program code) what vector graphics editors do for diagrams. From a *notational* point of view, there is no reason why structure editing cannot be made arbitrarily close to text editing; the text can be rendered straightforwardly as in a text editor, and it can be entered by ordinary typing. The key difference is in what is being built up in computer memory: in structure editing, a *tree* or *graph* is being built up as the user works, whereas a text editor merely builds a long character sequence.

In order to write programs to transform, analyze, or otherwise work with the digital artefact the user has created, one can trivially navigate the structure output by a structure editor. For the output of a text editor, one must first parse it *into* such a structure, and deal with any errors just before use, instead of at the moment the user introduced the error.

Consider an interface to enter a personal name made up of a forename and a surname. The user could be presented with a single text field, in which they type the the names separated with a space; or two fields, where the names are entered separately. We can also ask what is built up in memory: is it a single string, or two separate strings? Four *representation paradigms* are illustrated by the combinations that result:

1. *Sequence editing:* single field, single string. All the editor has to do is straightforwardly store whatever is in the text field. This corresponds to text editors, and more generally sensors and recorders for different data (video, photographs, and audio.)
2. *Sequence rendering:* two fields, single string. Here, there is a structured interface writing through to a string in memory. The information about the separation of the two strings, present in the interface, is not quite "thrown away" but it is made *implicit* as, say, a space character in the string. In order to recover this information, the string must be parsed later on. This combination corresponds to code generation from GUI forms in Visual Basic, as well as MIDI sequencers (which render musical notes played by virtual instruments to a single combined waveform), video editors, and renderers for 2D or 3D graphics. Another example is line-based diff tools, which provide side-by-side views and related interfaces, yet must ultimately forward the user's changes to the underlying text file.
3. *Structure editing:* two fields, two strings. As in sequence editing, both parts match, so there is not much work to do. This corresponds to structure editors, vector graphics editors, and 3D modelling tools.
4. *Structure recovery:* single field, two strings. Parsing needs to happen each time the input changes. This style is found in the DOM inspector in browser developer tools, where HTML can be edited as text to make changes to the document tree structure. More generally, this is the mode found in compilers and interpreters which accept program source text yet internally work on tree and graph structures.

4.8.5 Relations.

- *Level of Automation.* The human eye can see a rectangular array of pixels and straightforwardly recognise shapes. Programming a computer to do this is the subject of highly advanced research in *computer vision* and *artificial intelligence*. The human eye can also recognise structures in a formal language from the character sequences used to render them. However, getting a computer to do the same thing—parsing—is a matter of ordinary programming instead of advanced probabilistic techniques. The parsing of natural language, perhaps, lies between these two degrees of formalization.

- *Errors* (Section 4.6). A process that merely records user actions in a sequence (such as text editing) will, in particular, record any *errors* the user makes and defer their handling to later use of the data, keeping the errors *latent*. A process which instead treats user actions as edits to a structure, with constraints and correctness rules, will be able to catch errors at the moment they are introduced and ensure the data coming out is error-free.

4.9 Adoptability

How does the system facilitate or obstruct adoption by both individuals and communities?

We consider adoption by individuals as the dimension of *Learnability*, and adoption by communities as the dimension of *Sociability*.

4.9.1 Dimension: learnability. Mainstream software development technologies require substantial effort to learn. Systems can be made easier to learn in several ways:

- Specializing to a specific application domain.
- Specializing to simple small-scale needs.
- Leveraging the background knowledge, skills, and terminologies of specific communities.
- Supporting learning with staged levels of complexity and assistive development tools [Fry 1997]. Better *Feedback Loops* can help (Section 4.1).
- Collapsing heterogeneous technology stacks into simpler unified systems. This relates to the dimensions under *Conceptualization* (Section 4.3).

FORTRAN was a breakthrough in programming because it specialized to scientific computing and leveraged the background knowledge of scientists about mathematical formulas. COBOL instead specialized to business data processing and embraced the business community by eschewing mathematics in favor of plain English.

LOGO was the first language explicitly designed for teaching children. Later BASIC and Pascal were designed for teaching then-standard programming concepts at the University level. BASIC and Pascal had second careers on micropocessors in the 90's. These microprocessor programming systems were notable for being complete solutions integrating everything necessary, and so became home schools for a generation of programmers. More recently languages like Racket, Pyret, and Grace have supported learning by revealing progressive levels of complexity in stages. Scratch returned to Logo's vision of teaching children with a graphical programming environment emphasizing playfulness rather than generality.

Some programming languages have consciously prioritized the programmer's experience of learning and using them. Ruby calls itself *a programmer's best friend* by focusing on simplicity and elegance. Elm targets the more specialized but still fairly broad domain of web applications while focusing on simplicity and programmer-friendliness. It forgoes capabilities that would lead to run-time crashes. It also tries hard to make error messages clear and actionable.

If we look beyond programming languages *per se*, we find programmable systems with better learnability. The best example is spreadsheets, which offer a specialized computing environment that is simpler and more intuitive. The visual metaphor of a grid leverages human perceptual skills. Moving all programming into declarative formulas and attributes greatly simplifies both creation and understanding. Research on Live Programming [Hancock and Resnick 2003; Victor 2012] has sought to incorporate these benefits into general purpose programming, but with limited success to date.

HyperCard and Flash were both programming systems that found widespread adoption by non-experts. Like spreadsheets they had an organizing visual metaphor (cards and timelines respectively). They both made it easy for beginners to get started. Hypercard had layers of complexity intended to facilitate gradual mastery.

Smalltalk and Lisp machines were complex but unified. Once having overcome the initial lerning curve their environments provided a complete solution for building entire application systems of arbitrary complexity without having to learn other technologies. Boxer [diSessa 1985] is notable for providing a general-purpose programming environment (albeit for small-scale applications) along with an organizing visual metaphor like spreadsheets.

4.9.2 Dimension: sociability. Over time, especially in the internet era, social issues have come to dominate programming. Much programming technology is now developed by open-source communities, and all programming technologies are now embedded in social media communities of their users. The nature of these communities often trumps purely technical and individual considerations [Meyerovich and Rabkin 2012]. Some of the specific concerns of sociability are:

- Easy integration into standard technology stacks, allowing incremental adoption, and also easy exit if needed. This dynamic was discussed in the classic essay *Worse is Better* [Gabriel 1991] about how UNIX beat Lisp.
- Backing by large corporations or widespread industry investments that ensures economic sustainability.
- An open-source community of volunteers investing their time, which has proven to be as viable as financial support.
- 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. Sharing is impeded when languages have competing dialects and libraries, like Scheme [Winestock 2011].
- Friendly and helpful user communities on social media, for example Stack Overflow. Such communities have to some extent replaced the traditional role of documentation.

The tenor of the online community around a programming system can be its most public attribute. Even before social media, Flash developed a vibrant community of amateurs sharing code and tips. The Elm language invested much effort in creating a welcoming community from the outset [Czaplicki 2018]. Attempts to reform older communities have introduced Codes of Conduct, but not without controversy.

On the other hand, 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, and Haskell. These communities bear some resemblance to cults, with guru-like leaders, and fierce group cohesion.

The economic sustainability of a programming system can be even more important than strictly social and technical issues. Adopting a technology is a costly investment in terms of time, money, and foregone opportunities. Everyone feels safer investing in a technology backed by large corporations that are not going away, or in technologies that have such widespread adoption that they are guaranteed to persist. A vibrant and mature open-source community backing a technology also makes it safer.

Unfortunately, all of these issues of sociability create barriers to new programming systems targeting non-experts, and indeed the entire dimension of learnability. Large internet corporations have invested mainly in technologies relevant to their high-end needs. Open-source communities have only flourished around techologies for expert programmers "scratching their own itch". While there has been a flow of venture funding into "no-code" and "low-code" programming systems, it is not clear how they can become economically and socially sustainable. By and large, the internet era has seen the ascendancy of expert programmers.

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 design 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.

REFERENCES

Antranig Basman. 2021. *Infusion Framework and Components.* https://fluidproject.org/infusion.html

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*.

Joshua Bloch. 2007. *How to Design a Good API and Why it Matters.* http://www.cs.bc.edu/~muller/teaching/cs102/s06/lib/pdf/api-design

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.

Michael Alan Chang, Bredan Tschaen, Theophilus Benson, and Laurent Vanbever. 2015. Chaos monkey: Increasing sdn reliability through systematic network destruction. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 371–372.

Evan Czaplicki. 2018. . https://www.youtube.com/watch?v=uGlzRt-FYto

Andrea A. diSessa. 1985. A Principled Design for an Integrated Computational Environment. *Human–Computer Interaction* 1, 1 (1985), 1–47. https://doi.org/10.1207/s15327051hci0101_1

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

Luciano Floridi, Nir Fresco, and Giuseppe Primiero. 2015. On malfunctioning software. *Synthese* 192, 4 (2015), 1199–1220.

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

Nir Fresco and Giuseppe Primiero. 2013. Miscomputation. *Philosophy & Technology* 26, 3 (2013), 253–272.

Christopher Fry. 1997. Programming on an Already Full Brain. *Commun. ACM* 40, 4 (apr 1997), 55–64. https://doi.org/10.1145/248448.248459

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.

John B. Goodenough. 1975. Exception Handling: Issues and a Proposed Notation. *Commun. ACM* 18, 12 (1975), 683–696. https://doi.org/10.1145/361227.361230

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.

Sumit Gulwani, William R Harris, and Rishabh Singh. 2012. Spreadsheet data manipulation using examples. *Commun. ACM* 55, 8 (2012), 97–105.

C. Hancock and M. Resnick. 2003. *Real-time programming and the big ideas of computational literacy.* Ph.D. Dissertation. Massachusetts Institute of Technology. https://dspace.mit.edu/handle/1721.1/61549

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

Robert Hirschfeld, Hidehiko Masuhara, and Kim Rose (Eds.). 2010. *Workshop on Self-Sustaining Systems, S3 2010, Tokyo, Japan, September 27-28, 2010.* ACM. https://doi.org/10.1145/1942793

Robert Hirschfeld and Kim Rose (Eds.). 2008. *Self-Sustaining Systems, First Workshop, S3 2008, Potsdam, Germany, May 15-16, 2008, Revised Selected Papers.* Lecture Notes in Computer Science, Vol. 5146. Springer. https://doi.org/10.1007/978-3-540-89275-5

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

A. Kay and A. Goldberg. 1977. Personal Dynamic Media. *Computer* 10, 3 (1977), 31–41. https://doi.org/10.1109/C-M.1977.217672

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

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An Overview of AspectJ. In *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2072)*, Jørgen Lindskov Knudsen (Ed.). Springer, 327–353. https://doi.org/10.1007/3-540-45337-7_18

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

Donald Ervin Knuth. 1984. Literate programming. *The computer journal* 27, 2 (1984), 97–111.

H. Lieberman. 2001. *Your Wish Is My Command: Programming by Example.* Morgan Kaufmann.

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

Leo A. Meyerovich and Ariel S. Rabkin. 2012. Socio-PLT: Principles for Programming Language Adoption. 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, 39–54. https://doi.org/10.1145/2384592.2384597

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

Martin Monperrus. 2017. Principles of Antifragile Software. In *Companion to the First International Conference on the Art, Science and Engineering of Programming* (Brussels, Belgium) *(Programming '17)*. Association for Computing Machinery, New York, NY, USA, Article 32, 4 pages. https://doi.org/10.1145/3079368.3079412

James Noble and Robert Biddle. 2004. Notes on Notes on Postmodern Programming. *SIGPLAN Not.* 39, 12 (dec 2004), 40–56. https://doi.org/10.1145/1052883.1052890

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

Richard Pawson. 2004. *Naked Objects.* PhD thesis. Trinity College, University of Dublin.

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

Tomas Petricek. 2017. Miscomputation in software: Learning to live with errors. *Art Sci. Eng. Program.* 1, 2 (2017), 14. https://doi.org/10.22152/programming-journal.org/2017/1/14

Ian Piumarta. 2006. *Accessible Language-Based Environments of Recursive Theories.* http://www.vpri.org/pdf/rn2006001a_colaswp.pdf

James Reason. 1990. *Human error.* Cambridge university press.

Barbara G. Ryder, Mary Lou Soffa, and Margaret Burnett. 2005. The Impact of Software Engineering Research on Modern Programming Languages. *ACM Trans. Softw. Eng. Methodol.* 14, 4 (oct 2005), 431–477. https://doi.org/10.1145/1101815.1101818

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

Brian Cantwell Smith. 1982. *Procedural Reflection in Programming Languages.* Ph.D. Dissertation. Massachusetts Institute of Technology. https://dspace.mit.edu/handle/1721.1/15961

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 n°580 Sciences et Technologies de l'Information et de la Communication (STIC).

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

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