Myths and Mythconceptions

What does it mean to be a programming language, anyhow?

MARY SHAW, Carnegie Mellon University, USA

Shepherd: Richard P. Gabriel (poet, writer, computer scientist), California

Modern software does not stand alone; it is embedded in complex physical and sociotechnical systems. It relies on computational support from interdependent subsystems as well as non-code resources such as data, communications, sensors, and interactions with humans. Both general-purpose programming languages and mainstream programming language research focus on symbolic notations with well-defined abstractions that are intended for use by professionals to write programs that solve precisely specified problems. There is a strong emphasis on correctness of the resulting programs, preferably by formal reasoning. However, these languages, despite their careful design and formal foundations, address only a modest portion of modern software and only a minority of software developers.

Several persistent myths reinforce this focus. These myths express an idealized model of software and software development. They provide a lens for examining modern software and software development practice: highly trained professionals are outnumbered by vernacular developers. Writing new code is dominated by composition of ill-specified software and non-software components. General-purpose languages may be less appropriate for a task than domain-specific languages, and functional correctness is often a less appropriate goal than overall fitness for task. Support for programming to meet a specification is of little help to people who are programming in order to understand their problems. Reasoning about software is challenged by uncertainty and nondeterminism in the execution environment and by the increasingly dominant role of data, especially with the advent of systems that rely on machine learning. The lens of our persistent myths illuminates the dissonance between our idealized view of software development and common practice, which enables us to identify emerging opportunities and challenges for programming language research.

CCS Concepts: • **Software and its engineering** → **General programming languages;** • **Social and professional topics** → *History of programming languages*.

Additional Key Words and Phrases: vernacular software developer, closed software system, open resource coalition, general-purpose programming language, domain-specific programming language, generality-power tradeoffs, formal specifications, software credentials, sufficient correctness, fitness to task, problem-solving design, problem-setting design, exploratory programming

ACM Reference Format:
Mary Shaw. 2021. Myths and Mythconceptions: What does it mean to be a programming language, anyhow? *Proc. ACM Program. Lang.* 4, HOPL, Article 234 (June 2021), 44 pages. https://doi.org/10.1145/3480947

Author's address: Mary Shaw, Institute for Software Research, Carnegie Mellon University, 5000 Forbes Av, Pittsburgh PA 15213 USA, mary.shaw@cs.cmu.edu

2475-1421/2021/6 - 234
https://doi.org/10.1145/3480947

Apollo killing the python [deBaudos 17th cent]

MYTHS AND MYTHCONCEPTIONS ABOUT SOFTWARE

Traditional general-purpose programming languages, or at any rate the ones of most interest to programming language researchers, are largely designed for generality, abstraction power, completeness, soundness, and other formally tractable properties. They are intended for implementing correct solutions to well-understood precisely specified problems, either as complete programs or as fixed compositions of program modules.

Much modern software, in contrast, implements complex cyberphysical and sociotechnical systems. This software rarely has precise specifications, and it is embedded in systems that involve rich interactions among subsystems that include not only software components but also non-code resources such as data, communications, sensors, and human interaction. The constituents of these software systems are often created by third parties, they are themselves typically not well-specified, and they may change without knowledge of the systems that incorporate them.

The traditional view of software is rooted in our mythology of software—the stories we tell ourselves to make sense of the world, to shape and share our view of our discipline, to distinguish good from evil. These, like most myths, are idealized and aspirational, they tell of heroes and villains, they set aside complexity to celebrate core values.

> "A myth isn't simply a refutable claim; in its purest form it's a heroic narrative illuminating the human condition. It can establish a model for behavior and uphold social traditions. Myths have enduring appeal—people want to believe in them, notwithstanding contradictory evidence. This 'wish to believe' elevates a myth from a simple, objectively testable statement to a phenomenon whose appeal and persistence require interpretation." [Jaspan et al 2009]

Myths are narratives that help us understand our place in the world. They are intended to capture deep truths while providing insight; they are based in history—or its retelling—but they

need not be historically accurate. Myths are stronger than tacit assumptions; they are embedded in our culture in a way that resists challenge.

Let's examine some of the principal cultural myths of programming languages and programming. As with many myths, these arose in a bygone era, one that is even more idealized in memory than it was in reality: an era with the belief that *programming is the use of general-purpose languages to create correct solutions to well-specified problems.*

As is common with myths, there has always been dissonance between the ideal world embodied in the myths and the real world of software and software developers. The myths capture a glorious past that may always have been aspirational rather than accurate. The dissonance has increased over time as software has become more deeply embedded in everyday life and essential functions.

The problem is not so much that the myths are untrue as that they are incomplete. They should continue to guide us where they apply, but we should be acutely aware of the risk that they're blinding us to other possibilities. Critically examining software practice through the lens of these myths reveals opportunities for programming language research to better serve modern software and its developers.

We identify six principal myths and examine how the world portrayed by the myths (*se mythos*) differs from actual software development practice (*se pragmos*).

myth	*se mythos*	*se pragmos*
Professional Programmer	Programs are written by highly skilled professional programmers.	Vernacular software developers vastly outnumber professional developers. Professional developers now (mostly) do things other than write code.
The Code IS the Software	Software is simply the symbolic program text.	Software systems are coalitions of many types of elements from many sources with sketchy specifications and unannounced behavior change.
The Purity myths:		
Mathematical Tractability	Soundness of programming languages is essential.	Task-specific expressiveness is more important than completeness or soundness.
Correctness	Correctness of software is also essential.	Fitness for task usually matters more than absolute correctness.
Specifications	Thus formal specifications are also essential.	Much software is developed to discover what it should do, not to satisfy a prior specification.
...and a new one... AI Revolution	*...a new one going around...* Artificial intelligence (actually machine learning) is so special that it will break normal software development.	Artificial intelligence has long been an incubator for disruptive programming ideas; the issues of current concern have variants in conventional software, where there are established ways to deal with them.

We'll see how these myths inspire mainstream practice, and we'll point out, with examples, many of the ways that the myths don't speak to actual software and software development. The dissonances present new challenges and opportunities for programming language design and research, opportunities that seem to be obscured by the myths.

Pandora opens the vessel [Flaxman 1910]

THE PROFESSIONAL PROGRAMMER MYTH

Mainstream programming languages are usually designed under the assumption that programmers are trained professionals, so language designers may assume some mathematical background and sophistication on the part of programmers, along with the willingness to spend time learning each new language, its underlying computational model, and its operating environment. The steep learning curve for these languages is a barrier to entry for many people. Part of the reward for this effort is an associated mystique about having mastered the special knowledge.

This myth also holds that the principal job of these professional software developers is mostly to write code that can be rigorously shown to correctly implement well-specified requirements.

As computation has become pervasive, more and more people who are not professionally trained as programmers are creating and tailoring software as a means to achieve goals of their own. These *vernacular software developers* are principally interested in their own tasks, not in the software as a primary objective.

Some are professionals in some other domain for whom software is a means to a domain-specific end. Others are hobbyists or tinkerers. They create and instantiate software in a specific context, their principal responsibility lies in that

A **professional software developer** creates software as a principal occupation, usually for use by others, and has professional training in software development.

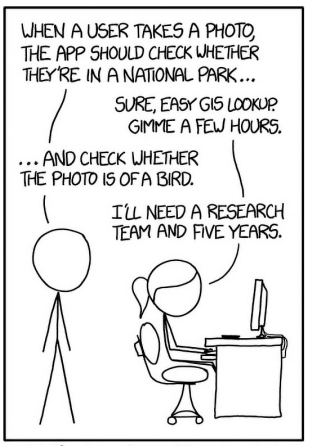
A **vernacular software developer**, on the other hand, creates software in a specific context, has principal responsibility in that context, and has professional training, if any, in that context rather than in software development; the software is a means to an end, not an end in itself. The alternative terms "end user" and "end user programmer" are pejorative; "non-professional" is misleading—many are professional, just at something else.

Here, "software development" includes not just the original creation of the software, but the quality and evolution of the software through its life cycle.

context, and their principal training is in that context rather than in software development. Some develop software in pursuit of personal interests; for others, software is a means to some professional objective.

Many people who develop spreadsheets, scripts, and databases professionally, for example, do not think of themselves as "doing programming." Nevertheless, their activity is commonly very like programming, requiring precise descriptions of the computations, comprehensive reasoning about the algorithms (not just a few examples), and evolution of the descriptions as understanding of the problem evolves. Indeed, vernacular software developers vastly outnumber highly trained computing professionals, by at least an order of magnitude or two [Scaffidi et al 2005, Scaffidi 2017]. Gartner Research says that over 40% of employees outside of IT departments customize or build data or technology solutions [Gartner Research 2021a].

Vernacular developers are typically not trained, nor interested in being trained, to use traditional general-purpose programming languages, nor do they necessarily share the cultural knowledge of the software domain or its engineering sensibilities about system integrity, maintenance, backup, and the like. Instead, they use spreadsheets, databases, scripts, web authoring tools, constraint systems, graphic composition tools, macros, and so on. Instead of writing sequences of programming language statements, they write sets of related formulas, develop macros and scripts with "follow me" examples, formulate sets of interdependent constraints, connect objects in diagrams, and develop user interfaces and web sites with graphical tools. Many develop their software incrementally, through trial and error. They may be quite sophisticated in the models and analysis techniques of their domains yet unsophisticated as software developers.



[xkcd 1425]

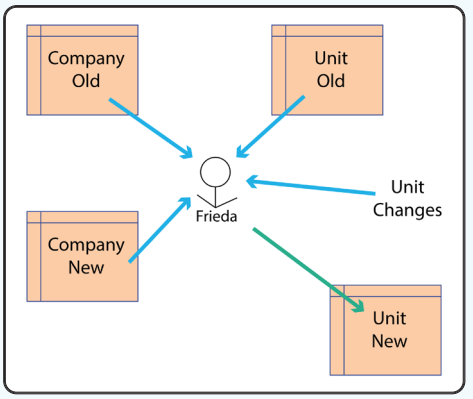
Is this programming? Yes, if you accept that a program is "a collection of specifications that may take variable inputs, and that can be executed (or interpreted) by a device with computational capabilities" [Ko et al 2011] rather than "a sequence of statements in a traditional programming language."

Spreadsheets are popular with vernacular programmers. They evolved from simple tools for elementary accounting to sophisticated modeling languages for propagation of data through complex secondary calculations, including visualizations. Their success is evident in their adoption as the tool of choice for myriad uses not anticipated by their designers, possibly because of their gentle learning curve as well as their functionality.

One common use is simply recording data, using the spreadsheet as a very simple unstructured database: in the EUSES spreadsheet corpus [Fisher and Rothermel 2005], a sample of 4498 spreadsheets consisting largely of spreadsheets found with web searches, the most common values in cells are integers and strings, and most of the spreadsheets contain no formulas. Language support for abstraction and correctness in spreadsheets is lacking, though checking tools are developed outside of the programming languages community [Ko et al 2011].

Frieda's Departmental Budget

Frieda is responsible for her department's budget tracking. Each year her company produces an updated spreadsheet that embeds new requirements. However, it doesn't quite match her department's needs, so every year she must adapt the company spreadsheet to her department. She works with both of last year's spreadsheets and this year's company spreadsheet to create this year's department spreadsheet. To do this, she has to rediscover (reverse-engineer) the changes she made last year and discover the changes the company made this year. Where the company has not made changes she can re-use last year's departmental changes, but she can't simply copy-and-paste because of the name binding rules. She must also make changes to incorporate new projects and adjust for the new elements of the company spreadsheet. At a high level, Frieda is inferring two transformations and trying to compose them—but she has neither notations nor concepts to treat this as a transformation problem, she must manually make the adjustments and hope for the best.



Thanks to Margaret Burnett for this example [Burnett and Myers 2014]. Frieda is a real person, interviewed for that study.

Scientific computing requires quite sophisticated calculations. A common pattern is to process the raw data in various ways and format it as input to a process in a scientific library, then process the output again to submit to another library process, and so on. This involves lots of scripting and data handling, plus calls to complex code written by other scientists. They must pick the right components, understand their interfaces (which may have complex configuration parameters), harness them together, and connect them to their data. They also face a challenge in deciding whether what they have written is correct. When, as is common, they do not have a precise specification of the desired output, their confidence in correctness co-evolves with the programs.

Scientific Data Analysis

Scientific programmers, unlike many professional programmers, frequently don't know quite what the correct answer is. Writing code to a specification that gives an equation relating the inputs to the outputs lets you check the outputs easily. However, if you're simulating a rapidly rotating black hole, the entire reason you're writing the code is that there isn't a closed-form equational solution.

So how do scientists create software to help them understand a phenomenon and figure out if that software is doing the right thing? The answer is spot checks: at each stage of the development process, the researchers eyeball the spot check, convince themselves it's OK, move on to the next stage.

Thanks to Greg Wilson, formerly of Software Carpentry, for this example [Wilson 2021].

> When people use Jupyter notebooks or the RStudio IDE to develop data analysis pipelines, a common workflow is:
>
> 1. Load the data and visualize it.
> 2. That looks OK, so filter the data and check there aren't any NAs ("not available", or result not returned).
> 3. That looks OK, so replace the check for NAs with a summarization step and visualize the groups.
> 4. Those groups look OK, so remove the visualization, add a pivoting step, and check the row and column totals.
>
> …and so on…
>
> [Wilson 2021]

This is not a new problem. Vernacular programmers have been using programming constructs for many years, without recognition or support. Many of them define their programs (or macros) by demonstration, for example by recording a series of actions that they perform on actual data. Although this would appear not to require programming skills such as abstraction, it suffers from difficulty in distinguishing which aspects of the demonstration constitute the algorithm being captured for future use and which aspects are incidental characteristics of the particular data on which the intended operations are being demonstrated.

Photoshop Action Recording

Even defining scripts or macros in interactive systems is programming-like. In Photoshop, for example, repetitive editing sequences can be captured as "actions" for future replay. However, the current context of the actions is captured in unexpected and unhelpful ways.

These actions are defined by example, that is by recording a series of operations. However, the documentation about recording actions is festooned with warnings about how current context such as file path, active layer, units, and default colors will be embedded—in other words, that unbound variables will be transported from one scope to another. These are scope issues that would clearly benefit from the attention of programming language designers.

Guidelines for recording actions

Results depend on file and program setting variables, such as the active layer and the foreground color. For example, a 3-pixel Gaussian blur won't create the same effect on a 72-ppi file as on a 144ppi file. Nor will Color Balance work on a grayscale file.

When you record actions that include specifying settings in dialog boxes and panels, the action will reflect the settings in effect at the time of the recording. If you change a setting in a dialog box or panel while recording an action, the changed value is recorded.

Modal operations and tools—as well as tools that record position—use the units currently specified for the ruler. A modal operation or tool is one that requires you to press Enter or Return to apply its effect, such as transforming or cropping. Tools that record position include the Marquee, Slice, Gradient, Magic Wand, Lasso, Shape, Path, Eyedropper, and Notes tools.

[Adobe 2021]

The myth of the **Professional Programmer** distracts programming language designers from opportunities to improve software overall by applying programming language design expertise to the notations and development processes so that they better match the needs of vernacular programmers.

Vulcan forging the thunderbolts of Jupiter [Rubens 1636–38]

THE CODE IS THE SOFTWARE MYTH

Historically, "program" was pretty much synonymous with "software." The mindset was that software systems are big programs constructed by compiling together smaller program modules, which are written in the same programming language and interact through procedure calls. Data was incidental, debugging involved direct interaction with the program text, maintenance was a necessary evil driven by factors that couldn't be predicted, documentation was missing or viewed as irrelevant. The behavior of the program depended only on explicitly invoked libraries and the underlying operating system, which changed only with explicit knowledge of the programmer. Programs embedded in physical devices were outside the mainstream. There was little sense of a program as part of a system that involved a development environment, distributed system infrastructure, user communities, and an operating ecosystem. We still see this **Professional Programmer** myth shaping software development processes that have a regular "build," which programmers worry about "breaking."

This myth persists into the present; the evidence for this includes the way that other forms of component interaction—such as concurrency,

A **program** is an executable definition of a calculation that accepts variable inputs. It is created by a **programmer**, using some kind of **programming system**, though not necessarily a general-purpose language.

A **software system** is a composition of programs and other resources, including possibly databases that provides a computational service. It is created by **software developers**, using some kind of **software development environment**. It differs from a program in scale, in scope, and in the developers' responsibility to the users of the system.

A **software development ecosystem** is a comprehensive software development environment that includes tools, languages, libraries, interface standards, and other resources that supports software developers. The software system may run in an equally rich execution ecosystem.

pipelines, client-server interactions, and so on—are invoked by procedure calls on code that implements those interactions. Such forms of component interaction are architectural in nature, but they have not entered general-purpose languages as first-class constructs. In many cases these interactions require multiple procedure calls in specific orders. Among the consequences of this second-class status are obfuscating the design of the system architecture and obscuring the number of technologies actually used. Recognizing that a software system comprises much more than just the code changes the game for both professional programmers and vernacular programmers.

Modern software has evolved beyond many of these assumptions. Even the simplest program runs on top of a deep infrastructure stack. Third-party components and large data sets are common. Many of these supporting elements are underspecified and subject to silent automatic update. Software involves not only programs written in traditional languages but also development environments and large datasets.

The discrepancies between myth and practice show up in different ways for professional and vernacular programmers

Professional software developers do many things other than write code

The *closed software system* view embedded in the myth **the Code IS the Software** was never completely accurate, of course, but the myth shaped the research direction of mainstream programming languages. As time has passed, the dissonance between myth and reality has only increased.

Modern software is made up not just of code in a single programming language, but also of code in multiple languages, supporting technologies, large datasets, synchronization for distributed execution, interface policies, scripts, real-time data feeds, dynamically selected components, automatic unannounced updates, and so on—much of which is not addressed by mainstream programming languages.

A **closed software system** is a composition of software elements that are under the control of the developer: the modules have reasonable specifications; updates are made explicitly and can be validated, the sources of components are known. The elements are mostly code modules, but not all elements are code.

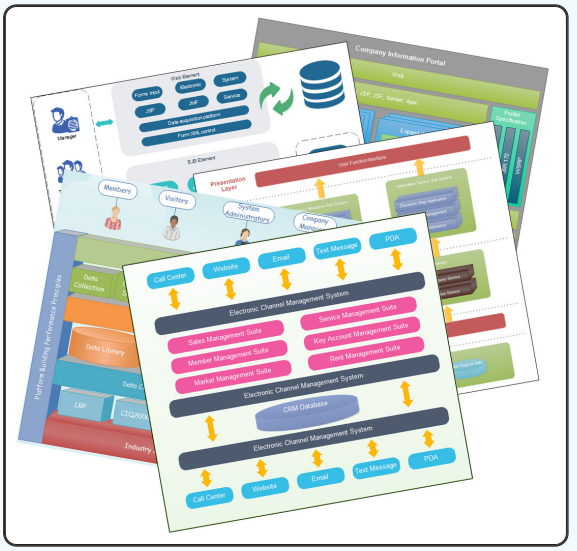
An **open resource coalition**, on the other hand, is a software system assembled from underspecified software components, unverified data, third-party resources, numerous layers of a software stack, and other resources that are under the control of third parties and may change dynamically and without notice.

These *open resource coalitions* transcend the classical closed software system concepts that a software system has boundaries, that it remains under the developer's control, and that it is dominated by lines of code written for the purpose. In addition, more than ever the software is created by using software tools such as scripting or generators to lash third-party components together, not by writing lines of code. The elements that are incorporated in the software may come from ad hoc, untrusted, unstable third party sources; they are underspecified and may change their behavior without notice. These coalitions must also be able to identify which versions of which elements are in use at a given time.

Programming-in-the Small and Programming-in-the-Large

This is not a new problem. In the mid-1970's DeRemer and Kron [1976] called out the difference between programming-in-the-small—that is, writing code modules and linking them together—and programming-in-the-large—that is, the overall organization of modules into components. Their language for describing software system organization was a simple provides-requires notation that showed dependencies among modules.

In the half-century since then, the abstractions, specification languages, and notations, especially graphical notations, for software architectures have been very significantly refined. Numerous architectural styles are recognized, and tools support diagramming. However, the integration with analysis tools and with programming languages in which the modules are written remains tenuous.



[Freeman 2021]

Traditional programming languages address only a modest portion of the task of assembling a resource coalition from components whose code was written by someone else and whose operation depends on continued predictable performance by arms-length strangers. The third-party code may not be available, and if it is available, its license may not permit modification or it may be written in an unfamiliar language. Software developers must scrutinize configuration parameters, wrangle APIs, set up data and communication protocols, manage the third-party bits (including the vagaries of unannounced changes), and generally worry about which bits not under their control are going to go awry next.

For example, a common architecture in modern software is a "software stack" that layers independent system components such as firmware, operating system, drivers, and middleware that lie between the application program and the hardware [Techtarget 2021]. This is so pervasive that "full stack developer" is a job category.

This sort of software system isn't constructed in the classical way, by linking modules together in a single build, but rather by enlisting available components (software, data sets, dynamic data feeds, etc.) into a coalition using tools like scripts and IDEs that you hope will serve your purpose. There may be many options available for a particular function such as a layer. A consequence of using

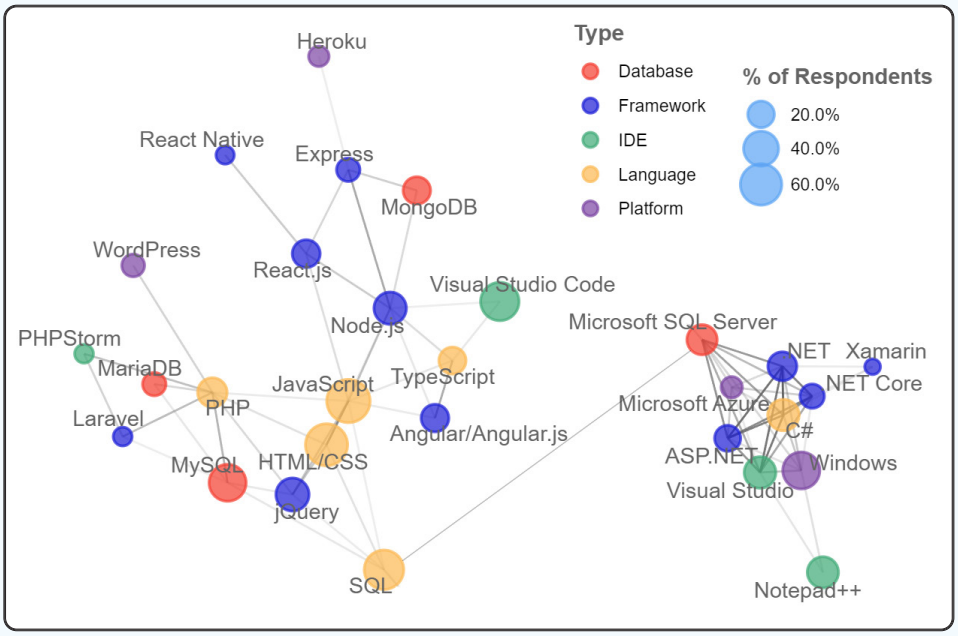
[xkcd 676]

independently developed components in this way is that meticulous record-keeping about sources and versions is required to know which components a system actually relies on. Version identification is now recognized as a "software bill of materials," an inventory of components and their versions including, recursively, the subcomponents and versions of these [NTIA 2021].

Even considering only the code components, the composition of a system from modules involves different abstractions and different composition mechanisms from code in a general-purpose programming language. The relationships among code modules—the software architecture—have long been recognized as an important design problem, but the languages for describing these are much less well-established than the language for code. Various other technologies support the management of the elements, but these have not been subjected to the same rigorous design and analysis typically accorded a traditional programming language. It is common for sets of related technologies, along with conventions about how they are used together, to form ecosystems in which developers contribute additional supporting technology.

Web Development Ecosystem

Stack Overflow [2017] identifies the ecosystem of technologies—the suite of complementary and compatible software technologies—used by members of the Stack Overflow community for web development and Microsoft technologies. The ecosystem identified in their 2019 survey shows a large web development cluster on the left connected via SQL to Microsoft technologies on the right. Note that the principal languages are HTML/CSS, JavaScript, and SQL, followed distantly by PHP, TypeScript, and C#.



[Stack Overflow 2019]

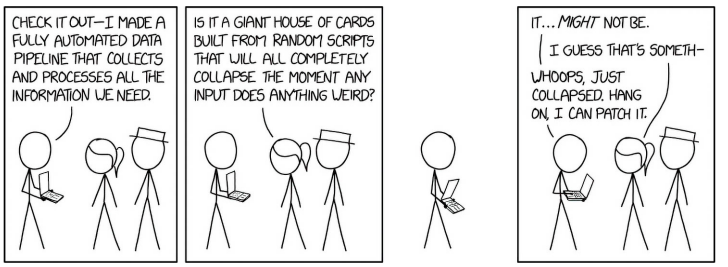
The myth that the **Code IS the Software** distracts programming language designers from opportunities to improve the overall process of software development—especially by professionals—by improving the design of the associated tools and frameworks as well as the programming languages.

The vastly more numerous vernacular programmers mostly use tools other than general-purpose programming languages

Many vernacular software developers are highly trained in their own domains and only lightly trained in traditional programming. They use a variety of tools and notations including spreadsheets, scripting, data schemas, markups, domain-specific languages, visual web development tools, and scientific libraries. Many of the tools don't represent the software as a static symbolic text; software development may involve progressive tailoring of a system, collection of formulas and constraints in spreadsheets or CAD systems, or visual programming. These developers need notations and tools tailored for their own problems, and they may not be able to justify the overhead of learning traditional general-purpose, high-skill languages with sophisticated (and complex) tools.

Vernacular programmers opportunistically explore possible ways to achieve their development goals, working to understand only as much as needed about unknown code fragments, scripts, data sources, and other elements, and exploring ways to fit some of the pieces together. Some of these elements may come from this vernacular programmer's own files, others may be discovered on the web. Some code fragments may have been written by this vernacular programmer, some by other vernacular programmers, and some by professional developers. The vernacular programmer explores different arrangements of these elements, evaluates how well each performs, works incrementally toward a solution, and may revisit some earlier variants tried before. Vernacular developers need effective ways to find useful fragments; understand, assess, and reuse those fragments; create variants; and understand the reasons for differences between variants. They need a variety of visualizations to help in these tasks [Burnett and Myers 2014].

It is important for the tools used by vernacular programmers to provide programming support that fits their own models of the application domain and that also respects the principles of programming language design. Many of these tools would benefit from application of programming language design principles such as precise syntax, abstractions, encapsulation, scope, and clear denotations —bearing in mind that the computation may be developed incrementally in the application and there may not be a single text file representing the program.



[xkcd 2054]

It is especially important for these tools to support programmers who are developing the software as a way of understanding their problems or the ramifications of their decisions—people who are using software to work *toward* a specification rather than *from* one. Programming in this exploratory mode involves getting feedback on whether initial ideas accurately capture the intention, then refining and extending the result to achieve a satisfactory program by progressive approximations while refining their understanding of their real intention.

Schön [1984] describes this design process as "a conversation with the materials of the situation." Many professional programmers work this way too, of course, but it's especially prevalent in vernacular programming.

Many vernacular developers are more concerned with collecting, aggregating, and visualizing information than with complex calculations, and many turn to web sites and online tools as their infrastructure of choice.

Web Mashups

Mashups are not new. Web mashups—which dynamically combine data from different sources to visualize in a browser—predate APIs that were actually designed for web integration. Google Maps has long been a popular base layer for mashups, dating at least to 2005 [Duvander 2010]. For example, during the 2008 US election, there was widespread concern about long queues at polling stations. A web mashup provided a current public display of wait times at polling locations based on reports collected from individuals via Twitter, SMS, and telephone in a stylized (and widely ignored) format, extracting polling locations and wait times, then combining the results to display over a Google map alongside the raw data feed [NPR 2008].

[Example collected in 2008 by Mary Shaw, reported by NPR [2008]]

The models and abstractions embedded in traditional programming languages are foreign to many vernacular software developers. These developers may become frustrated while trying to understand how to apply these foreign concepts to their own tasks. Conversely, the models of some non-language tools may be frustratingly foreign to professional developers.

Markup Languages

Web sites are among the most common types of vernacular software. Markup languages provide the backbone for web development, but markup languages existed long before there were computers, for example in the form of conventional copyeditor's annotations. The earliest computer markup languages annotated text for formatting. These evolved to TeX, in which the author essentially writes a program that produces typeset text. Document formatting has progressed beyond that, with visual editing tools providing abstractions that enable users to obtain good results without dealing directly in markup details, though some communities cling to the more complex programming model.

Meanwhile, Web formatting has also evolved from relatively simple static pages in HTML to much more sophisticated successors such as WordPress [2021]. The addition of site complexity such as dynamism and content management added new features to these tools, and the resulting systems suffer from lack of proper language design.

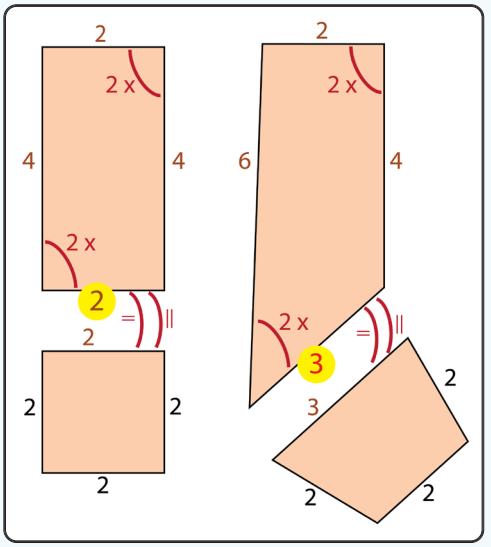
WordPress was originally developed as a simple blogging tool, a point on the power end of the generality/power curve. Its success in that domain led it to be used for many other types of web sites, but in many ways it has not escaped its roots in blogging, a model too specialized for general web site development. As a result, it has evolved from a simple visual web editor to a sophisticated system with templates, plugins, and dependencies. Its conceptual model can be confusing to users whose conceptual models arise from programming languages and document markup languages—the capabilities are there, but the interface is highly non-intuitive.

Some vernacular developers are quite sophisticated, especially about the mathematics and models of their own domains. They may develop highly interactive, highly visual applications. They may invest significant resources in the integrity of their applications over time and over different software infrastructure. Their programming language or tool of choice, however, is frequently not a traditional general-purpose programming language.

AutoCAD: Parametric Constraints

AutoCAD is a sophisticated computer-aided design software ecosystem [Autodesk 2021]. Its core is a rich parametric constraint system for graphically defining 3D models. The constraints may control the dimensions of individual objects or geometric relations between objects. The constraints may be formulas involving other constraints. Constraints interact, so changes made to one constraint may adjust other constraints automatically, frequently in unexpected ways. Writing sets of parametric constraints is very much a programming activity.

AutoCAD also includes toolsets supporting a variety of professions such as architecture, mechanical engineering, and mapping. These additional tools include libraries of standard parts, generators for standard 3D elements, drawing assistants, analysis tools, automated bills of materials, interfaces with other systems (e.g. ArcGIS), input and output data conversion. Using AutoCAD successfully requires more than the ability to visualize 3D objects. Orchestrating the interaction with other parts of the ecosystem is also a programming activity, for example when converting the 3D model to a toolpath for a CNC machine tool (which may require iteration to respect the capabilities of the milling machine) or preparing it for 3D printing by adjusting for the properties of the intended material and providing auxiliary support structure.



The diagrams illustrate a simple operation with parametric constraints. In the left diagram, two constraints require the long side of the quadrilateral to be twice the length of the short side. Two additional constraints require one side of the quadrilateral to be parallel to and of equal length to one side of the other quadrilateral.

Simply changing the highlighted “2” to a “3” causes changes in other parts of the figure as required to preserve these constraints.

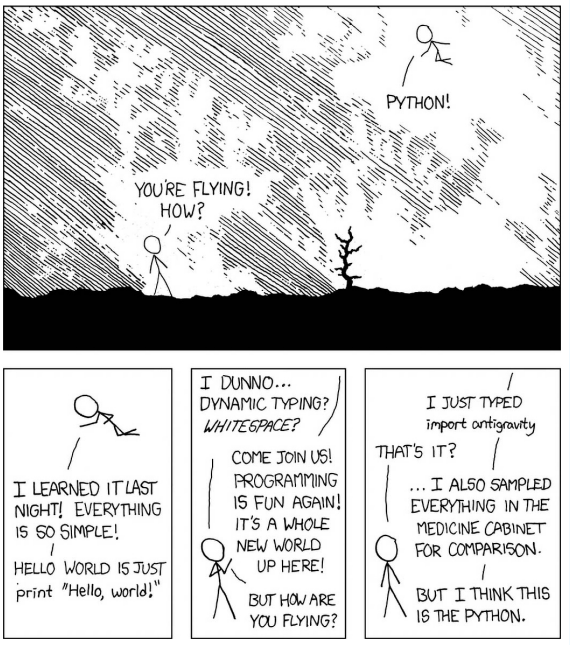
The right diagram shows one of many possible outcomes (the system is under-constrained, there are many others).

At the other end of the spectrum, an emerging trend of low-code and no-code development aims to provide easy entry to the programming profession and to address a perceived shortage of software. One might argue that these are professional programmers because their principal job is producing software for others, but their training and mode of operation resembles vernacular programmers. In any case, vernacular developers should benefit from the tooling.

Low-Code Development Environments

Gartner Research [2021b] recognizes a category of "citizen developers" who use low-code environments to develop applications for use by others. Gartner sees low-code application development as a way to address the shortage of developers, predicting 65% of applications will be developed in this way in the next few years [Bouhuijs 2019].

Indeed "low code development platforms" are the subject of an upcoming theme section in the Springer Journal of Software and Systems Modeling [2021]. "The growing need for secure, trustworthy, and cost-efficient software…and the shortage of highly skilled professional software developers have given rise to a new generation of low-code software development platforms, such as Google…and Microsoft PowerApps. Low-code platforms enable the development and deployment of fully functional applications using mainly visual abstractions and interfaces and requiring little or no procedural code. This makes them accessible to an increasingly digital-native and tech-savvy workforce who can directly and effectively contribute to the software development process, even if they lack a programming background. At the heart of low-code applications are typically models of the structure, the behavior and the presentation of the application. Low-code application models need to be edited (using graphical and textual interfaces), validated, version-controlled[,] and eventually transformed or interpreted to deliver user-facing applications."



[xkcd 353]

This call for papers focused on engineering activities such as interoperability and scalability, but not so much on the visual languages at the heart of the platforms.

The myth that the **Code IS the Software** distracts programming language designers from opportunities to apply language design, abstraction, and modeling principles to software that is developed incrementally, especially in forms other than text, by vernacular software developers.

The maiden and the unicorn [Domenichino 1602]

THE PROGRAMMING LANGUAGE PURITY MYTHS

Mainstream traditional programming language research focuses on **Purity**: symbolic notations with precise specifications and well-defined semantics that support provably correct solutions to well-specified problems. This has given rise to a pair of related myths, that **Mathematical Tractability** of programming languages and the **Correctness** of software are essential—and these two myths invoke a third (supporting, myth) that **Specifications** can and should be precise, complete, preferably formal, and available at the beginning of the programming effort.

In practice, though, programming languages are used to define practical software solutions to real-world problems, under time and budget constraints. In other words, they are for defining how to accomplish some computational task that some actual person cares about, with reasonable assurance that the result will be good enough. This purpose has become increasingly important over the years, as computation has become embedded in most aspects of everyday life. So, a programming language is for making

A **traditional programming language** is a symbolic notation with (a) a defined syntax; (b) language constructs including primitive elements, expressions for computing new elements, abstraction mechanisms for composing expressions, and perhaps a closure rule that compositions of expressions must yield legal elements; and (c) semantics that associate meaning in the application space with programs. [after Abelson et al 1996].

A **general-purpose programming language** is a traditional programming language that is agnostic as to the types of software it will be used for, emphasizing generality and completeness, in which programs are represented as lines of code. It is intended for creating correct solutions to well-specified problems, and it typically has a significant learning curve.

A **domain-specific language** is a language or tool that supports a specific domain, or a specific type of developer, or software development in a style other than (principally) writing lines of code.

software, especially real-world software. Mathematical tractability and correctness can certainly support this, but you need look no farther than your laptop or smartphone to see that software can be "good enough" without being "correct," and the discussion above shows that traditional programming languages do not address all of software development. Further, both professional and vernacular programmers commonly develop software in an exploratory mode, evolving the specifications along with the software and using that evolution to better understand what the software should do.

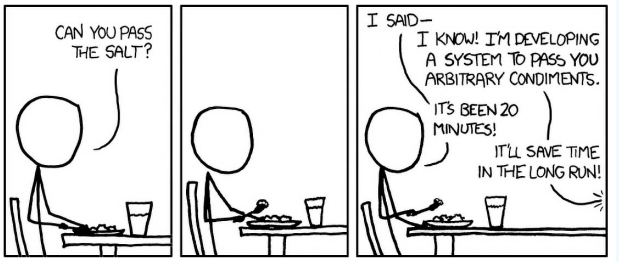
The myth of Mathematical Tractability favors mathematical elegance over expressiveness or domain-specific power

Traditional programming language research emphasizes generality, soundness, completeness, richness, abstraction mechanisms, elegance, and so on. The researchers have deep expertise in different programming models (procedural, functional, logic, data flow, object-oriented,...) and the formalisms that support reasoning about the programs. They design languages intended for use by professional software developers who are adept at generalization, modeling, abstraction, and symbolic reasoning. Using these languages requires programming expertise and, in many cases, mathematical sophistication and maturity.

But how widespread is that mathematical proficiency? According to the StackOverflow 2020 survey of professional programmers [StackOverflow 2020], only 3/4 have earned a college degree, and fewer than 2/3 of those received an undergraduate degree in computer science or mathematics. So fewer than half have college education that would suggest any proficiency in formal systems, even assuming that proficiency persists into their careers and is at a high enough level to handle type theory and category theory. Even more concerning, over 15% of the professional developers didn't think that college education is even needed to be a developer. In other words, there's significant dissonance between the level of formal proficiency expected by programming language researchers and that attained by developers.

Requirements for Mathematical Sophistication

A senior industrial developer with a PhD recently said, "Over several years, I've tried and failed to learn functional programming because that means learning a big pile of applied category theory. Despite claims that I can just use the parts I do understand, I find that reusing common libraries, like web serving, entails understanding the advanced stuff because it's revealed in their interfaces." [Confidential 2021]



[xkcd 974]

The sentiment is echoed by Syme [2021], whose response to a github request for type classes in F# included "Adding type-level programming of any kind can lead to communities where the most empowered programmers are those with deep expertise in certain kinds of highly abstract mathematics (e.g. category theory). Programmers uninterested in this kind of thing are disempowered. I don't want F# to be the kind of language where the most empowered person in the discord chat is the category theorist."

Independent of the formal proficiency of the software developers, traditional programming languages do not support the abstractions required to describe the large-scale organizations of code

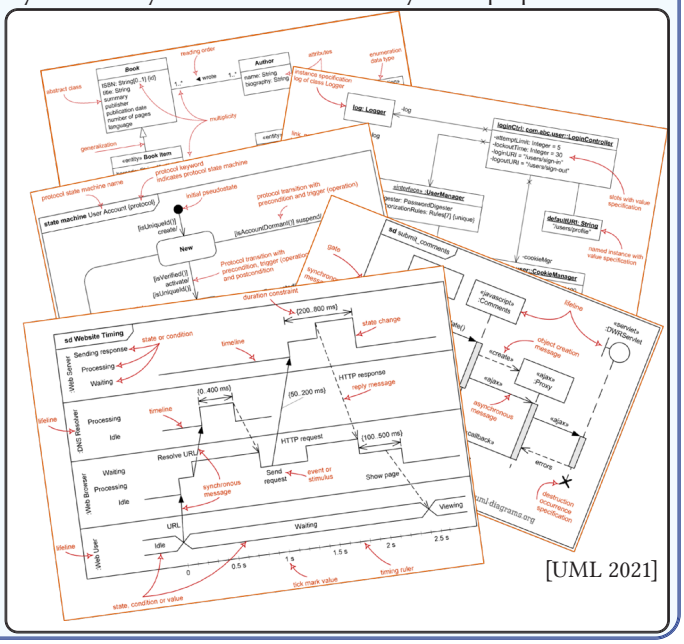
modules into systems; their primitive constructs don't reflect the diversity of real component types. As noted above, DeRemer and Kron [1976], building on early work by Parnas [1972a,b], called out the need for a language to describe module interconnections. The UML suite of notations followed.

UML: The Unified Modeling Language

The need for mathematical tractability extends beyond the module boundary to the properties of the system as a whole. In addition, the properties of interest go beyond functionality to timing, synchronization, communication, resource utilization, and interactions.

The Unified Modeling Language is a suite of a dozen or so notations, each addressing some system-level behavior or structure.

While each individual notation in the suite has a clear purpose and a formal basis, the suite as a whole is incomplete. Further, it lacks high-level abstractions, a way to coordinate models in the various notations (or even check consistency), and a strong connection to the code that implements the models.



[UML 2021]

Designers of general-purpose programming languages strive to make those languages broadly applicable. They choose primitive elements, control structures, and semantics based on common mathematical constructs, and they rely on software developers to create abstractions—typically in the form of procedure libraries—to add representations and computations for specific domains. In this way they favor formal elegance over expressiveness in the context of application.

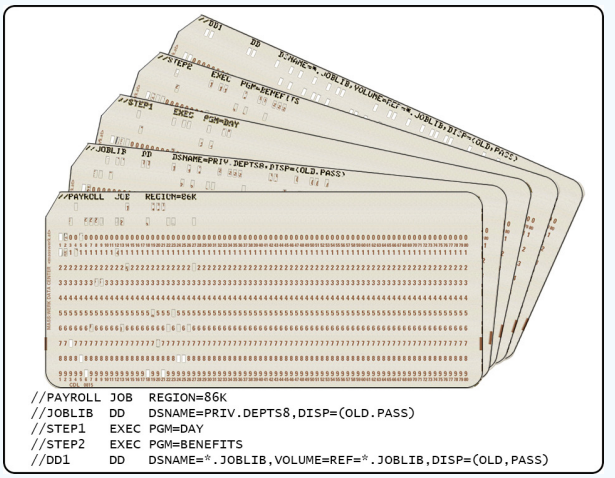
By striving for generality, language designers pass up the opportunity to provide first-class targeted support for specific domains. However, *domain-specific languages* have long provided direct support for more specialized models. Many have been designed ad hoc, and they would benefit from the formal rigor that's brought to the design of *general-purposes languages*. This separation of interests misses an opportunity to systematically explore tradeoffs between generality and power in programming languages.

Nevertheless, both professional and vernacular developers make extensive use of domain-specific languages—the former for the non-code and organizational aspects of systems, the latter in many cases as their principal tools. Few of these languages, however, have been subjected to the discipline of careful programming language design to support internal consistency, completeness, and even parseability.

IBM 360 Job Control Language: "The Worst Computer Language Ever"

Failure to treat less-than-fully general languages as real languages, with the inevitable consequences, is not a new problem; it has a long ignoble history. Take, for example, the Job Control Language (JCL) designed for IBM's OS/360 in 1965, described by Brooks [2010, pp 169–173] as "The Worst Computer Language Ever."

"A vivid example of expert failure is IBM's Operating System/360 Job Control Language (JCL).... It is, I am convinced, the worst computer programming language ever devised by anybody, anywhere.... It is instructive first to examine JCL's deficiencies as a programming language. Then one must inquire how a software team of real experts, having on call, for example, designers of the original Fortran and leading language theoreticians, could go so radically wrong. Although the mistakes were made 45 [now 55] years ago, JCL is still in use, in essentially the same form. The mistakes continue to curse us. And the lessons are timeless."



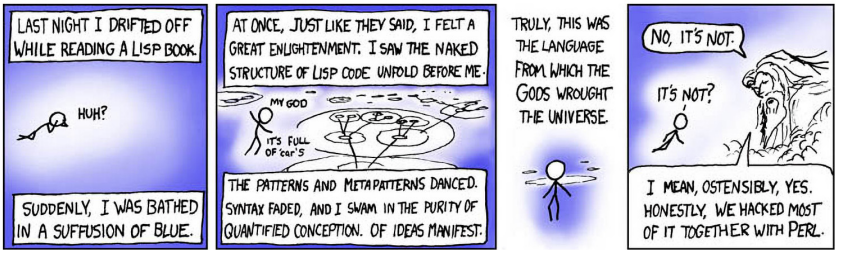
[IBM 1971 p.114, Landsteiner 2021]

Brooks goes on to describe JCL as a scripting language for batch jobs, one that is so hard to learn that people copied working scripts blindly. It forced all programmers to use a second language. It used a 72-column punch-card model when the rest of the system assumed terminal access. Its operator set (verbs) was not well matched to the complexity of the task. It provided no iteration and almost no branching, and the litany of shortcomings goes on.

"*The biggest flaw of all was that JCL is indeed a programming language, but it was not perceived as such by its designers*" — Frederick P. Brooks [2010].

Expert languages designers who favor generality and formal rigor miss the opportunity to provide those benefits to the domain-specific languages that support many vernacular programmers. However, rigor and formality are useful at many points on the generality-power tradeoff curve. So all the other notations, especially the ones used by vernacular programmers, desperately need the kind of loving care and attention that is regularly lavished on general-purpose languages.

There are two important reasons, both legitimate and respectable, for studying and designing programming languages. One emphasizes **Purity**: the theoretical basis of the languages and values generality, soundness, and mathematical purity. The other emphasizes the usefulness of languages to software developers for describing solutions to their current problems, including messy things like state, interfaces with the real world, richness of expression, and



[xkcd 224]

match to problem. Proponents of the former will, for example, value minimality of the language base for its mathematical properties while proponents of the latter will, for example, value richness of the constructs available as part of the language. Both approaches have merit, and misunderstandings arise when a proponent of one approach fails to appreciate the merits of the other.

This examination of our myths is an engineering view, focused on the need to produce practical software under engineering constraints—it's frankly utilitarian. It recognizes the role of theoretical research while seeking a balance between mathematical elegance and utility that will serve the full breadth of modern software.

There are lots of other examples. The point is: the programming language research community has, by and large, not taken them seriously. But the sheer volume of software development based on notations and tools other than traditional general purpose programming languages cries out for the same care and attention. The myth of **Mathematical Tractability** distracts programming language designers from opportunities to trade generality for power in a language while maintaining the formal rigor now largely reserved for general-purpose languages.

The myth of Correctness favors functional correctness over fitness for purpose

For almost half a century the dream of provably correct software has driven much of programming language research. Achieving this dream requires not only **Mathematical Tractability** of the programming language but also rigorous, complete, formal specifications against which to verify the code.

We have long paid lip service to the doctrine that **Specifications** are sufficient, complete, static, homogeneous, and usually purely functional—and that we can rely (exclusively) on those specifications when validating components and composing them into larger systems.

This has never actually been accurate for systems of any size, but this **Correctness** myth has compelling allure.

In fact, though, real software components have incomplete, evolving, heterogeneous, and nonmonotonic descriptions; a propensity to change without notice; and undocumented assumptions. Their descriptions evolve as new information is gathered from usage or analysis [Garlan et al 1995, Shaw 1996, Shaw and Scaffidi 2007, Garlan 2010]. To emphasize the difference between an *idealized formal specification* and the information available in practice, we call the latter *credentials*. Credentials are explicitly open-ended, heterogeneous, and dynamically updated representations of what is known about a component, with what confidence, and on what authority (provenance). They include extra-functional properties and evolve as new information is learned about a component. Unfortunately, this information is currently dispersed rather than consolidated and managed systematically, so it is hard to consult, to evaluate, and to update.

Specifications are formal (or at least rigorous), verifiable assertions about behavior of a software component, usually assumed to be sufficient, complete, static, homogeneous, and typically restricted to functional behavior.

Credentials are heterogeneous, incomplete, sets of partially verified information about a component, to be updated, possibly nonmonotonically, as new information is available, and of variable provenance and confidence.

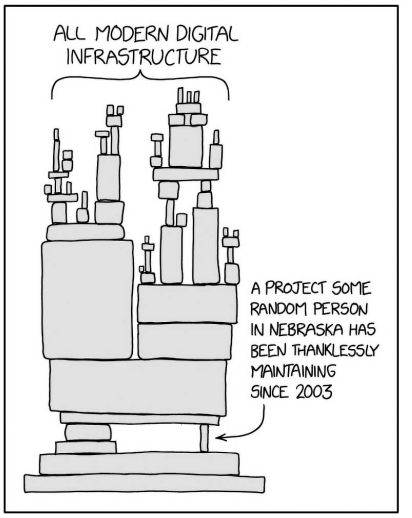
The completeness of specifications is particularly problematic. First, the list of properties that might be specified is open-ended, and it's hard to list them all, let alone to verify them. When observable properties are not specified, programmers may make tacit assumptions that run afoul of the actual implementation. For example, when three different components, all designed to be reusable, were reused together, the integration failed because all three assumed they had control of the main event loop [Garlan et al 1995]—and the authors of the reusable components had

not thought to include that in the specifications. Second, there is cost associated with writing and verifying the specifications or of empirically estimating specifications from existing code. Even if it were possible to enumerate all the properties that anyone might depend on, the cost of determining the specifications would be prohibitive.

Correctness is a siren song that distracts us from the more practical question "is this software fit for its intended purpose"—especially when the intended purpose requires properties other than functional correctness. Without adequate specifications for the components, there's no way to reason formally about the result. When owners of components can change them without notice, users have no assurances. As a result, formal verification is, on the whole, a minor player for real software.

Software correctness is intrinsically problematic because of many types of uncertainty:

- Even if individual modules can be formally verified, they execute in an unverified software environment.
- Even if the compiler and operating system were correct, it's not practical to verify the rest of the software stack, nor are the specifications of all the layers such that a given program's interactions with them can be formally verified.
- Simply identifying which versions of which software elements affect the behavior of a system can be challenging; hence the current interest in the "software bill of materials" [NTIA 2021].
- Understanding of software's requirements may coevolve with software development, which precludes a firm, fixed specification.
- Much software is now provided "in the cloud"; current practice updates it continually and without notice.



[xkcd 2347]

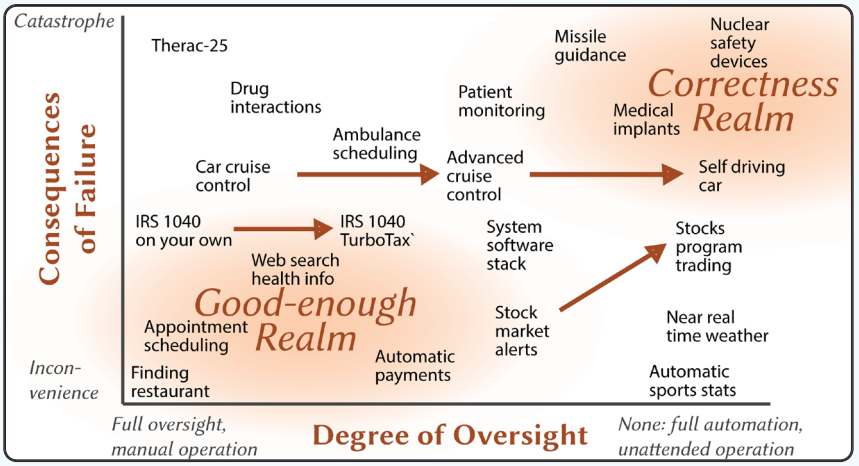
- Software embedded in physical systems is subject to the uncertainties arising from the physical systems.
- Solutions to societal problems are hard. These problems do not lend themselves to even forming consensus on the problem definitions, let alone definitions of success; they are called "wicked problems" [Rittel and Webber 1973].
- Acquiring specifications has cost, so it makes the most sense to get the specifications relevant to the task rather than trying to be complete.

These uncertainties introduce failure opportunities at many points of the system and call for end-to-end arguments [Saltzer et al 1984] about the system behavior. So for most practical software systems, the objective is not so much **Correctness**, especially functional correctness relative to a formal **Specification**, as fitness for purpose.

The criteria for "fitness for purpose"—whether the software is "good enough"—depend on things like the consequence of something going wrong and the likelihood that a problem will be detected and corrected before anything goes seriously wrong. Formal verification remains important for critical parts of critical software—expensive, but worth it because of the consequences. Security of common operating systems and other common infrastructure is a particularly important target for verification. Commodity and cloud software doesn't pretend to be "correct," you're pretty much on your own to figure out whether it's good enough for you. Responsible software developers understand where their applications lie in this space.

Good Enough is Often Good Enough

Some applications are sufficiently critical to justify the expense of verification, especially if they're fully automated and humans may not notice problems. For others—the vast majority—fitness for task is sufficient because the consequences of failure are fairly low and humans are in the loop, so "good enough" is good enough.



[Shaw 2015, 51:27–52:57]

The most serious problematic case is autonomous embedded software that's turned loose on the public (I'm looking at you, frenzied stampede for fully autonomous vehicles). An additional concern is creeping automation, when software that was originally used manually takes on new functionality and more autonomy and creeps toward the critical region without explicit review and requirement for better validation. We can at least imagine a model that makes explicit the level of need for validation, depending on the consequences of failure and the degree of human oversight, even though this is not totally satisfying.

How do software developers, especially vernacular developers, convince themselves their software is good enough for its intended use?

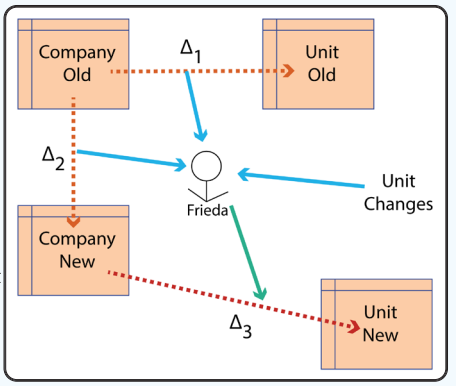
Professional developers' tools and development environments provide extensive support for testing and analysis. This is helpful to the extent that precise specifications are available, but in many cases "fitness for purpose" requires thoughtful discussion with clients. Many clients are not mathematically sophisticated, and discussions with such clients must take place in a common language. The translation of those discussions to precise, let alone formal, specifications presents an opportunity for miscommunication.

Vernacular developers, on the other hand, have little training or systematic support for reasoning about their programs. Especially when their software is not written in conventional programming languages, they have little alternative to observing the software in action to see whether the outcomes are plausible. Further, they may not know exactly what output they need at the outset, and with this evolutionary approach their understanding of the problem can coevolve with the solution captured in their programs. This is, of course, the basis of the trial-and-error or progressive-approximation approach to programming. The End User Software Engineering area [Ko et al 2011] seeks techniques for vernacular programmers to establish fitness for purpose of their software.

Frieda's Departmental Budget, revisited

Recall Frieda's annual task of adapting the company's budget tracking spreadsheet to her department. She is really constructing a transformation Δ_3 based on previous transformations and changes in the department.

How does she decide that her revised spreadsheet is correct? She makes the changes incrementally, and at each step she tries out some current numbers and eyeballs the values that come out. This helps her find the variable binding problems as well as errors in her recall of last year's adjustments. She does save some versions in case she needs to revert segments. But she has neither notations nor concepts for transformation functions, so she must rely on intuition to decide whether the changes that adapt the company's model to the department are correct, or even whether they are the same as last year. She has used reverse engineering, reuse, programming, testing, and debugging, mostly by trial and error.

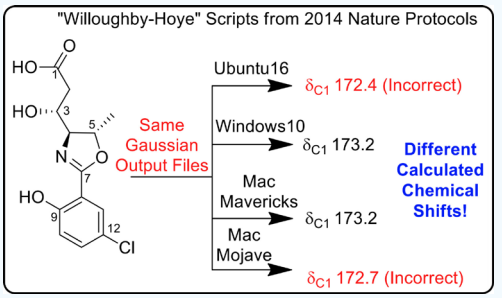


Thanks to Margaret Burnett for this example [Burnett and Myers 2014].

A recent example shows the perils of modern software in which scripts invoke code through APIs. A Python script commonly used by chemists delivered different results on different operating systems. In this case, a subtle reliance on an apparently unspecified property of an API led to inconsistent results across machines, possibly affecting many studies that used the script. If APIs were designed and specified as carefully as programming languages, assumptions like this could at least be visible.

Cross-Platform Scripting Inconsistency

"A programming error in a set of Python scripts commonly used for computational analysis of chemistry data returned varying results based on which operating system they were run on—throwing doubt on the results of more than 150 published chemistry studies.... The scripts, called the 'Willoughby-Hoye' scripts after their authors,...were found to return correct results on macOS Mavericks and Windows 10. But on macOS Mojave and Ubuntu, the results were off by nearly a full percent.



[Bhandari et al 2019, abstract]

"The reason for the variation was the scripts' use of Python's *glob* module, which searches for files matching a specific name pattern—the scripts generated a list of input files to read based on the glob results. But the module depends on the operating system for the order in which the files are returned. And the results of the scripts' calculations are affected by the order in which the files are processed."

Willoughby replied that the scripts had worked 6 years earlier and the problem emerged after changes in the operating systems [Gallagher 2019].

The myth of **Correctness** distracts programming language designers from considering the need to support reasoning with partial and unreliable information about extra-functional as well as functional properties—and from considering that it may be time to rethink what we mean by "correctness" for a complex software system, especially an embedded system.

The myth of Specification favors early specification over exploratory programming

The myth of **Correctness** engenders another myth: the myth that we have, or could have if we tried hard enough, a full **Specification**—either as a requirement on the software to be written, or as a commitment that a user of a component could rely on. After all, how can you prove correctness of the code without such a specification? So the **Correctness** myth is intimately bound up with the myth of full formal **Specification**. There is another challenge to the **Specification** myth, the notion that some sort of requirement or specification, even an informal one, should precede writing the software, and we turn to that myth here.

We've seen that an important, perhaps even dominant, mode for vernacular programmers is in fact to use the software development process as a vehicle for understanding what problem they're actually trying to solve. Sometimes they do not have access to ground truth or an oracle to check the results, so they write some code and make judgments about how well it meets their expectations.

Scientific Data Analysis

Scientific programmers gain confidence in their results by developing the software incrementally and spot-checking the results for plausibility at each stage. In this way, they develop confidence in the results as they explore the computation itself. So they develop confidence in the result with the same steps as for development: each time they produce an intermediate or final result, they scan a table, create a chart, or inspect some summary statistics to see if everything looks OK. Their heuristics are usually easy to state, like "there shouldn't be NAs at this point" or "the age range should be reasonable," but applying those heuristics to a particular analysis always depends on the data scientist's evolving insight into the data in question.

Unit testing frameworks aren't designed with this model in mind, and while researchers could leave the checks in place, they almost never do (in part because so many of them depend on visualization) The correctness challenge is exacerbated by the manual nature of the spot checks: the spot checks usually don't leave traces and aren't re-runnable, so it's not practical to write and reuse scripts.

Thanks to Greg Wilson, formerly of Software Carpentry, for this example [Wilson 2021].

The exploratory mode of development is especially, though not exclusively, important in research settings. This was certainly the case half a century ago, when artificial intelligence researchers were seeking to understand human intelligence by writing programs that performed tasks perceived as requiring intelligence. In addition, professional programmers also sometimes operate in this mode, and they have done so since time out of mind.

The Hardest Part of Design is Deciding What to Design

Brooks relates his experience as a summer student programmer, tasked with generating a certain type of report. Each morning he presented his client with a new sample of the report. Each morning his client studied the report, said that this is what he asked for, and requested another revision. Frustrated at first, Brooks eventually realized that the most useful service he was performing for his client was helping him to decide what he really wanted [Brooks 2010, pp.22–23].

Engineers recognize two general modes of design and development [Shaw 1990]:

- Routine, or precedented, design is appropriate in well-understood settings where a body of engineering and domain knowledge supports finding solutions that are similar to established solutions. Routine design involves solving familiar problems, reusing large portions of prior solutions. This largely aligns with code-to-specification for software, though some specifications might take the design outside the precedented envelope. In these settings, it is normal and expected to have at least precise requirements for the system performance, if not formal specifications for the behavior of components.
- Innovative, or unprecedented, design is appropriate when extensions to engineering and domain knowledge are required to find satisfactory solutions. Much innovative design involves finding novel solutions to unfamiliar problems. The engineered systems may be one-of-a-kind, or they may be breakthroughs that enable new classes of products. This largely aligns with exploratory programming, especially when the design is exploring what's actually possible.

Inkwell: a Program that Writes Poetry

Richard P. Gabriel is both an experienced, mathematically knowledgeable software developer (here known as Dick) and a poet (here known as Richard).

Dick implemented a system called Inkwell to help Richard better understand the nature of poetry. Inkwell is Dick's program that writes poetry (among many other things like that). It uses lots of weird-a-mundo mechanisms, most of which Dick invented. There are two questions Dick and Richard ask when trying out a new mechanism:

- Dick asks: is the result of my programming activity doing what I intended that result to do—that is, did I "correctly" implement my wacky new idea?
- Richard asks: is my program doing what I want it to do—that is, did the sum of all my wacky ideas in InkWell produce language the way I want or hope for?

For example, here is the "preferred" order of types of adjectives in English:

opinion-size-age-shape-colour-origin-material-purpose Noun

Dick didn't want to explicitly program in that order, and he tries to avoid blind machine learning for such things. So, he decided to try to rely on n-grams to sort things out. Therefore, an InkWell "program" might require deciding whether it should be "my fat big dog" or "my big fat dog." The frequency of the correct 2-gram ("fat big" versus "big fat") should tell the story. So Dick coded up an n-gram thing that works with the underlying optimization engine that does the hard work. If Inkwell produces get "my fat big dog," that's evidence for Richard that something might be wrong, but Dick can't a priori tell what it is: the n-gram thing he just coded, the optimization engine, the interaction between the previous two, some parameters in the "InkWell Program" for this generation task, the quality of n-gram data that's available.... Thus, it could be coding that Dick did wrong, the algorithm, the data, or the overall approach. The problem with InkWell is that usually the only hint of a bug is wording that Richard finds weird or unexpected. This was so prevalent (weird wording) that Dick had to write introspection code to show Richard why certain wording decisions were made. Or perhaps this should be called "consciousness code"? (What Richard learned was that just about all apparently ungrammatical sentences were in fact grammatical when he saw nuances of word definitions or when he saw that some oddball phrases were actually idioms he didn't know.) [Gabriel 2020]

Inkwell illustrates a contrast between two design paradigms. Designers work in two important modes: design as problem-solving and design as problem-setting [Visser 2006]. Our myths celebrate design as problem-solving, of which Simon [1996] was a leading proponent. In *Sciences of the Artificial* he presented this problem-solving sort of design as optimization (when the structure of the problem is well-known) or heuristic search (in other cases). But there's another side of

design, of which Schön [1984] was the leading proponent, which is design as problem-setting. In *The Reflective Practitioner*, he presented this mode as one in which the designer begins with partial understanding of what's actually needed and evolves that understanding by "conversation with the materials"—that is, by learning through exploration both what the real need is and what (possibly new) capabilities can be supported by the materials.

> "I shall consider designing as a conversation with the materials of the situation.... A designer makes things. Sometimes he makes the final product; more often he makes a representation—a plan, program, or image—of an artifact to be constructed by others. He works in particular situations, uses particular materials, and employs a distinctive medium and language.... He shapes the situation in accordance with his initial appreciation of it, the situation 'talks back', and he responds to the situation's back-talk. In a good process of design, this conversation with the situation is reflective." [Schön 1984]

We see this operating in the Inkwell example, where Dick manipulates the materials and serves as their voice, Richard is the voice of the task, and their dialog leads to exploration of avenues that neither has yet considered.

Brooks [2010] lays out the advantages and disadvantages of the "rational design process" beloved of engineers. This orderly process proceeds from goals to constraints to objective functions to design trees. It's tidy and organized. But—and it's a big but—the rational design process assumes a well-defined problem. If the goal is vague or incompletely specified, or if the design space is not well understood, or if constraints keep changing as the problem is better understood, then the rational design process should yield to progressive co-evolution of the problems statement and the design.

We see that there's a long tradition in design and engineering of progressively refining your understanding of both what's needed and of what's possible during the design/development process. The myths, particularly the myth of **Specification**, do not capture the significant role of exploratory programming. This continues into the present. Developers of systems using machine learning components frequently lack ground truth and must rely on subjective judgment and iterative development both for curating the data and evaluating the models.

Reports from Intelligent System Developers

A field study of experienced developers with years of intelligent systems development showed frustration with the ad hoc processes they found themselves using, referring to machine learning as akin to magic [Hill et al 2016]. Quoting from the interviews,

"Too many of our cases, the machine learning people are...in the background doing this like a black art."

(about problem solving) *"a process of going from obvious tricks to one level away from, like, voodoo."*

"A lot...is done manually by the high priests of machine learning."

The myth of **Specification** distracts programming language designers from two important things about specifications. In the context of **Correctness** we already confronted the impracticality of obtaining complete specifications—either as a requirement or as a description—and suggest that we would be better served by evolving, open-ended credentials.

We see here that the **Specification** myth also fails in focusing on the use of programming languages to write code to satisfy a specification and prove it correct. This distracts programming language designers from supporting the exploratory mode of software development.

The fall of Icarus [Anonymous, 17th cent]

THE AI REVOLUTION MYTH

The progress of technology has moved current software far beyond the idealized world of our myths. We are now in the midst of an "AI Revolution"—more specifically the advent of systems that incorporate not just artificial intelligence but machine learning algorithms that draw sophisticated inferences from very large datasets. According to this myth, machine learning systems are so different from conventional systems that we need a brand new software discipline to cope with them.

To the contrary, the artificial intelligence area has a long history as a wellspring and incubator for new programming ideas. These ideas initially seem impractical, even crazy, but some of them eventually acquire respectability and enter the mainstream—whereupon their roots in AI are largely forgotten.

In other words, many features and capabilities in modern programming language arose from AI challenges, if not revolutions. Language designers and practical use further refined the ideas, and our languages are richer for the influences from artificial intelligence.

AI is an incubator for programming ideas

Let's look at just a few of the ideas that have made their way from the artificial intelligence community to programming languages and mainstream software. AI brought us the view of design as problem-solving, sometimes eclipsing the complementary view of design as problem-setting. Search strategies were a mainstay of AI research, years before search became a major industry.

List processing languages originated in the AI community well over half a century ago, introducing list structures and functional programming.

Lisp: List Structures and Functional Programming

Lisp was developed for the IBM 704 in the late 1950's. The largest model of the 704 had a memory of 32K (yes, kilobytes) of 36-bit words. Program memory was very precious. Yet Lisp dedicated half of each word to the links that represented the list structures, taking advantage of machine instructions that made it efficient to extract the fields. It was mind-blowing to see the designers commit half of the available storage to the structure of the data rather the data itself, plus additional precious CPU time to garbage collection. This was even stunning to some of us who already worked on machines with non-contiguous array structures implemented with indirection and garbage collection.

In addition, Lisp was conceived as a mathematical notation for programming languages, in the spirit of Church's lambda calculus. It was the first in a long line of functional programming languages, several of which are now mainstream.

A couple of decades later, many artificial intelligence researchers were using exploratory programming in support of their attempts to write programs that could carry out particular tasks that were viewed as requiring "intelligence." Production systems emerged as a technique for implementing expert systems; these were typically developed incrementally. Notwithstanding this exemplar of exploratory programming, the prevailing sensibility of the programming language research community at that time remained implementing to specification and proving correctness.

Exploratory Programming in Expert Systems

In the 1980s, AI researchers strove to capture in software the ability of humans to make decisions. Without full knowledge of what the humans know, they turned to production systems. A production rule system is a set of condition-action pairs. Whenever the condition matches the current state of the world (as represented in the program), it triggers the action. There is no inherent sequencing.

Developers began by writing rules that expressed common cases and added new rules as they discovered discrepancies between the system behavior and their expectations. That is, they practiced programming by progressive approximation, a form of exploratory programming.

These systems did not scale well because of interactions among rules (and absence of scope mechanisms in the rule systems). Nevertheless, production systems were an early example of how iterating toward acceptability can be a legitimate alternative to "always correct, though perhaps incomplete."

A common theme of these examples is that when AI researchers build prototypes they are less constrained by performance considerations than are developers of production software. As a result they can tolerate inefficiency as they explore new programming paradigms, with some assurance that additional insights and Moore's Law may make the ideas practical someday.

Machine learning is the current offering from AI to the mainstream

So what about the current offering from AI to mainstream programming? The oft-voiced concerns about systems that incorporate machine learning (ML) start with the dominant role of data and its curation. There are also concerns about opacity, dynamic change, correctness, nondeterminism, and other extra-functional characteristics of their components: they are only approximately specified, they change dynamically as they are retrained, their algorithms are opaque. Software components have always behaved like this, but ML components do so with a vengeance.

The issues in machine learning resemble issues the software field has addressed before, and there are established approaches to dealing with them in software. Consideration of the other myths has shown that, independent of AI issues, we should embrace, rather than deny, the inherent

incompleteness and uncertainty of software systems—of their components, of their interactions with other systems, of the contexts in which they operate and, indeed, of the requirements they are intended to satisfy.

Perhaps machine learning will be the forcing function—the final straw—that finally breaks us free from the mystique of our myths, even though we should have done this long ago. Rather than treating ML as a novel challenge signaling an AI Revolution that invalidates existing software principles and processes, let us draw on what we have already learned from conventional software.

Collection and curation of large datasets plays a central role

Machine learning systems rely on automated collection, classification, cleansing, versioning, and analysis of very large quantities of data harvested in the wild. Moreover, the datasets are regularly updated and the models are regularly re-derived (re-learned). The inferences from these models depend on the quality of the data as well as the quality of the learning algorithms.

The programming language model of composing a system from code modules does not provide for these administrative operations. But database systems also have large datasets that require separate management from the code. The administration required for machine learning datasets is different from the administration required for databases, but both cases require system management of data as well as code.

We know from conventional data-intensive systems that data quality is often a major issue, so curation of datasets assumes an increasingly prominent role as datasets grow. The larger the datasets, and the more automatically they are collected and processed, the more vulnerable they are to noise and errors in the data.

Programming languages' type models focus on strict type checking,and they do not handle noisy data well. As a result, poor data quality poses a particular challenge to programming languages: formulating type models that deal appropriately with noisy or otherwise low-quality data.

Threats to data quality are of many kinds. Most basically, the data itself may be ill-formed: a dataset that is supposed to contain natural language text might contain executable files or TIFF/JPEG/PNG files or raw sensor data. Even if the data is well-formed, it may be internally inconsistent. If it's internally inconsistent, it may be simply incorrect. More seriously, datasets collected in the wild are vulnerable to intentionally or maliciously incorrect data. All these data quality problems appear in conventional systems, but they come to the fore in machine learning systems for which data is the central concern.

Data Noise in Inkwell

Continuing the Inkwell example, Dick reports that setting up his n-gram technique required processing 12TB of data that had many ill-formed elements: "For example, in InkWell, those n-grams I use are distilled from something like 12TB of data from Google. My poor computer with only 8 cores spent about three weeks of wall-clock time continuously running to clean up the noisy data in Google's n-gram set. And this cleaned up only approximately the most egregious line-noise-like noise. For example, the original texts they used were the results of scanning zillions of books. They apparently threw into their book-scanning hopper books in Russian (cyrillic) which were scanned as if in English (latin). Then all the resulting misspellings and nonsense had to somehow be match-up-able to the WordNet dictionary material, the CMU phonetic dictionary, other scanned dictionaries with both American and British spelling, words in random languages." [Gabriel 2021]

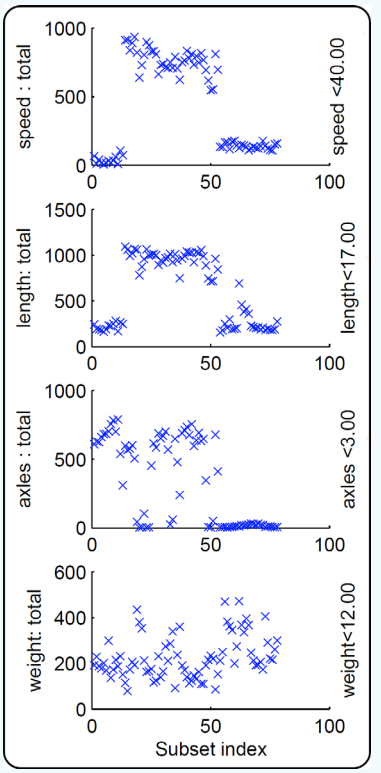
Even when data is syntactically well-formed, it may be incorrect. It may, for example, be the output of a processing pipeline that delivers well-formatted results but may not handle errors in the input properly.

Semantic Anomalies in Apparently Well-formed Data

The Minnesota Department of Transportation places sensors in the road to collect traffic data. This data is processed into a stream of time-stamped observations that include speed, length, number of axles, weight, and vehicle class, among other attributes. This "weigh-in-motion" data is then used for traffic analyses of various sorts including the effects of commercial truck traffic on pavement materials and designs. Clearly the data should be of good quality, but the large volume makes human checking impractical. Raz et al [2004] studied a 1998–2000 data and found high rates of anomalous observations. Left undetected, these anomalies could adversely impact the planned analyses.

For example, the data for three-axle trucks was a time-ordered set of 100 data subsets, each with 2000 observations. The figure shows for four different properties the number of data points in each subset that were either too high or too low. The number of axles is very noisy in early observations and very clean in later observations (and the data included a period in which 20% of the vehicles classified as three axle single unit trucks were reported as having a single axle). There is also a correlation between very low speed and over-length vehicle, which also got cleaner in the latter part of the period.

The study identified a period in which the anomaly rates were very high. Subsequent discussion with MinDOT showed that some internal processing inconsistencies had been corrected and some recalibration performed at approximately the times shown in the data.



[Raz et al 2004, Figure 4]

Health records for individual patients matter. Really, they do. However, the dirty little secret of electronic health records is that they're full of errors. Some of the errors are arguably minor. Some are consequences of poor software, like propagating stale data through cut-and-paste or inaccuracies resulting from forced selection from an inadequate dropdown list. Some are quite serious, like wrong or omitted diagnoses. Others are introduced to expedite insurance payments, for example by elevation of a secondary diagnosis to primary. Bell et al [2020] found that when patients read the notes on their doctor's visits, over 20% find mistakes, and they perceive over 42% of these as serious. Errors in diagnosis were most common serious errors, followed by errors in medical history and errors in medications or allergies.

In addition to the problems with entries in individual records, mismatches of records occur when multiple records are aggregated. A Pew Foundation [2018] study found that patient record matching within a facility could be as low as 80% (1 in 5 patients are not matched to all their records in a facility where they have been seen) and as low as 50% between organizations using the same electronic health record vendor. This situation was bad enough for paper records, when records for a given patient were consulted only by that patient's physicians. But the problem becomes exponentially more problematic with electronic health records, when inferences that affect many patients are made from huge numbers of these erroneous records.

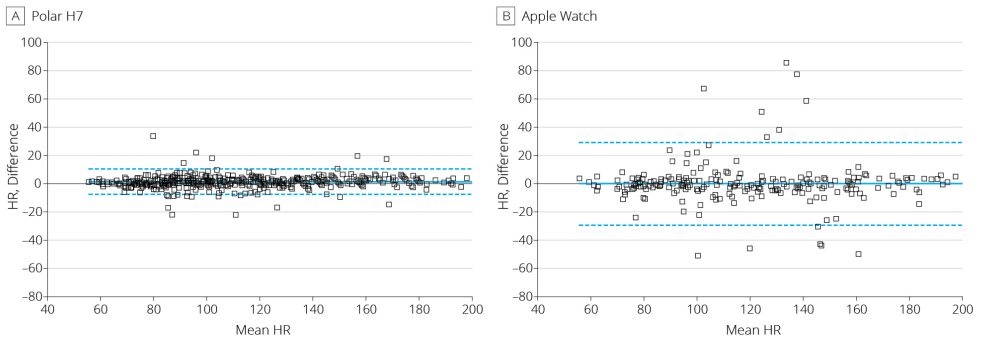
Modern consumer electronic monitoring devices have introduced an additional complexity in electronic health records. A patient's personal records, either manual notes or logs from electronic

monitoring devices, can provide useful diagnostic information. However, it is not in general as trustworthy as data collected in a medical setting with certified medical devices. This presents a challenge of tracking the confidence and provenance of the data in health records.

Data Quality in Consumer Heart Rate Monitors

Wang et al [2017] found variable accuracy among wrist-worn heart rate monitors, with none achieving the accuracy of the Polar H7 chest-strap monitor. Accuracy was generally better at rest, diminishing with exercise. They found 95% of the readings within about 35–40 beats per minute of ECG readings, varying by watch brand.

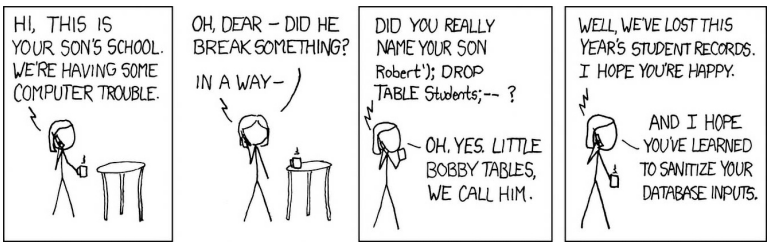
Figure. Bland-Altman Plots and 95% Limits of Agreement With Electrocardiographic Measured Heart Rate (HR)



[Wang et al 2017, figure [only]]

Nelson et al [2020] comment on the potential benefits, since these devices support continuous, scalable, and unobtrusive, "big data" collection of overall cardiac activity in real-world conditions with large samples (and the potential for p-hacking). They discuss factors that affect the accuracy of wrist-worn consumer devices and distinguish average accuracy from accuracy at any point in time. They also note "The lack of 'metadata standards'…for describing, reporting, and creating meaning from collected data in order to increase transparency, accountability, and interpretability of data in mHealth is likely to create a number of issues related to noise/signal ratio, contradictory results, lack of generalizability, and false positive and negative findings, which have historically undermined replicability (i.e., the ability to replicate a study with a new data set) and reproducibility (i.e., the ability to replicate findings from the same data set)."

Failure to verify that user-generated input actually conforms to the program's type definition is a long-standing source of security vulnerabilities, of which perhaps the most familiar is the SQL injection vulnerability. Well-formed data can also be malicious: for example, fraudulent reviews are a well-known problem in recommender systems.



[xkcd 327]

Ordinarily, data quality problems are handled with a combination of human oversight and software. In the case of Inkwell, Richard might notice problems in wording, for which Dick had to distinguish between bugs in Dick's code and noise in Google's data.

The MinnDOT data was used in research on anomaly detection, intended to find predicates that would detect when data feeds deviated significantly from expectations. In the case of electronic health records, a patient and an attentive physician might recognize and sort out errors—but there are also horror stories of inattentive physicians. The shift to electronic health records raises the stakes for improving data quality, and early data suggests reduction in error rates, though with the introduction of new issues of usability [Virginio and Marques Ricarte 2015].

These examples show that data quality, from ill-formed data to well-formed but erroneous data, is a long-standing problem in conventional systems. The problem is exacerbated in machine learning systems by the vastly larger quantities of data and the exploratory nature of model development. Even professional developers with extensive experience in machine learning systems struggle.

Reports from Intelligent System Developers

A field study of experienced developers with years of intelligent systems development showed frustration over their ability to obtain good data and to establish ground truth for that data [Hill et al 2016]. Quoting from the interviews:

About acquiring data:

Of the developers interviewed, 85% obtained data in some way other than collecting it themselves.

"This may be a very rich, good source of data or it may be junky...hard to sift through.... There may be so little uniformity in what we find...that non-uniformity will render the data basically useless."

"we still, years after this product is being sold...<have> fundamental data that's hard to get at."

About establishing ground truth in the data:

All of the developers interviewed reported difficulty establishing ground truth due to error and faulty data.

"even the canonical benchmark sets...have label noise or garbage data...."

"so ground truth is a huge problem, very labor intensive,...we're doing a lot of this very manually."

There are other problematic aspects of machine learning

Let's turn to the other unnerving aspects of machine learning, the characteristics that are said to make machine learning unique [Horneman 2019, Kästner and Kang 2020, Ozkaya 2020]:

- *Opacity:* ML algorithms are not capable of explaining how their outputs are related to their inputs. But we pretend that we could understand a conventional component if we tried hard enough, though for practical systems that isn't really the case.
- *Dynamically changing behavior and nondeterminism:* ML components are iteratively refined and continuously improved as models are re-trained from fresh data—that is, their effects can change behavior without notice. This effectively makes the systems nondeterministic. But ordinary components provided by third parties in the cloud—for example, ThingOfTheDay-as-a-service—also change behavior without notice, and nondeterminism is intrinsic to anything that's controlling a physical system.
- *Lack of specifications:* ML components are intended to model the phenomena that generated the input data. If we had oracles—precise models of those phenomena—we wouldn't need to use ML. In other words, we use ML precisely because we don't have good Specifications. So it's inherent that results will be judged against subjective criteria. But, despite the myth of Correctness, this is also true of many systems in the real world, especially sociotechnical systems—they have weak, incorrect, or incomplete specifications.

- *Criteria for correctness and bias:* Absent precise specifications, binary decisions about correctness are not possible. Instead, many systems are judged on the extent to which they deliver results that reasonably match expectations. But a similar situation occurs in conventional software, when fitness for task is accepted as the standard of evaluation. Similarly, developers of ML systems are properly concerned about biased outcomes—that is, outcomes that might match the data but do not match societal expectations. But, again, biased algorithms were incorporated in sociotechnical systems long before ML raised them to a level of public concern.
- *Unpredictable component interaction:* Partly because of the absence of specifications, ML components are prone to exhibit hidden dependencies, model interactions, and nonmonotonic behavior. But, again, hidden dependencies and nonmonotonic behavior are well-known in conventional software systems.

These are the characteristics that are said to make machine learning systems problematic. They all appear in conventional software systems as well, though in different forms, and there are established approaches to dealing with them. Why do we think those established techniques can't be adapted for ML? Let's look at some of the issues that arise from these characteristics.

Nondeterminism

Machine learning systems appear nondeterministic because each infusion of new data may change the learned models. When the models are updated automatically, in the background, the resulting system appears nondeterministic. This, of course, rides roughshod over the classical philosophy of **Correctness**, which holds that any change at all to a program invalidates what you know about its correctness.

This apparent nondeterminism also looms large in conventional systems, though. When software is a coalition of elements maintained by third parties, their specifications may not fully define the behavior of the software and their updates may change behavior that you depend on. When their updates are installed automatically, you may not even be aware of the cause of unexpected behavior. This is even more likely to occur if the third-party components are selected dynamically. In the world of enterprise software, the uncertainties are handled through software contracts—that is, in a form far removed from the programming language.

[xkcd 1838]

Real nondeterminism arises in software that controls physical systems. This software must deal with the inherent nondeterminism of physical systems—mechanical tolerances mean that actions are not precisely repeatable. General-purpose languages don't provide much help here. Instead, cyberphysical systems handle nondeterminism explicitly in a number of ways, primarily with feedback loops that inspect the state of the system and apply corrections to bring the system back into conformance with expectations. Here is another opportunity for better programming language support; even the numerous UML notations did not include feedback control.

Correctness and bias

Correctness can only be evaluated relative to some objective. This is true of conventional algorithms, but it is especially poignant when bias comes from inferring a predictive model from large aggregates of data. Is the standard of "correctness" simply the accuracy of prediction or replication

for observed data? Or in the absence of an oracle or ground truth, is it sufficient to mostly "seem right"? Or is there a higher standard about not supporting biased outcomes in the use of the data?

Discrepancies of the first two kinds might arise because the inferred model is not faithful to its input, or from the organization of the software system.

When software uses problematic data as input and the results are accepted uncritically, unfortunate consequences ensue. When the output depends on large-scale automation with opaque algorithms and the results are not repeatable because of dynamic data updates, the unfortunate consequences can compound rapidly. In the case of machine learning, the problem is exacerbated by apparent confusion about correctness. If a model accurately reproduces or predicts behavior of the people who generated the data, it is in some sense "correct." However, accepting those inferences uncritically can be socially unacceptable, because the behavior they have modeled (however accurately) is biased or otherwise undesirable.

A more serious concern, though, is bias of the latter kind. This lies outside the domain of programming languages and tools: the observed data is about real people making real decisions, and real people have real biases. It's inevitable, then, that inferred models capture the biases demonstrated by those people—though it's appalling that they're used uncritically. This is a problem of separating predictive power from socially desirable outputs. It's a very significant social problem, and it calls on the developers of this software to rethink what we mean by Correctness. This also reminds us that AI has wrestled over the years with the relations among data, information, and knowledge—the latter requiring interpretation in context, not in isolation.

Reliability

Machine learning systems appear to be unreliable, evidently because of hidden dependencies and model interactions [Kästner et al 2021]. Conventional software recognizes coupling and cohesion as a design concern precisely because of the risk of hidden dependencies; feature interactions have been studied for decades, and the infusion of fresh data can cause behavior to appear nonmonotonic.

Principles of reliable software support development of reliable systems from unreliable components with techniques such as modularization and architectural firewalls, redundancy, runtime checks and feedback, version control, checkpoints, checkable assertions, prioritization of failure severity, fault tolerance, graceful degradation, stochastic modeling, and so on. These localize or compensate for the vagaries of the unreliable components, provide intermediate sanity checks, define the envelope of allowable performance, support analysis of field variability, and indeed use machine learning techniques to detect anomalies in the system [Lyu 1996].

Abstractions

Integration of machine learning in programming languages and associated systems will require refinement of the abstractions appropriate to the domain. What are the principal types of components, and do we think of them as algorithms, models, datasets, functions, or something else? What architecture for the system provides intellectual control over data curation, training, and invoking the resulting models, which occur at different times in the operation of the software system? What are appropriate abstractions for handling uncertainty, nondeterminism, and opacity with the right level of detail?

Many of these characteristics lie outside the scope of traditional programming languages, of course. But the concerns of abstractions, nondeterminism, component interaction, and correctness (or fitness in the absence of precise specification) call out for adapting concepts from programming languages. Further, our entire field would benefit from languages that address uncertainty directly, for example with type systems rich enough to handle probabilistic values or anomaly checking.

Machine learning will not be the last challenge

So we see that the myth of the AI Revolution—that machine learning is essentially different from other software—doesn't stand up; established techniques and languages provide starting points for addressing the distinctive characteristics of machine learning. Machine learning is not alone in challenging the established norms of programming languages and software development.

Information Rules

Shapiro and Varian [1998] addressed a similar situation when they analyzed the apparent "new economy" of software, which was claimed to invalidate traditional economic. To the contrary, they argued, "we kept hearing that we are living in a 'New Economy.' The implication was that a 'New Economics' was needed as well, a new set of principles to guide business strategy and public policy. But wait, we said, have you read the literature on differential pricing, bundling, signaling, licensing, lock-in, or network economics?... Our claim: You don't need a brand new economics. You just need to see the really cool stuff, the material they didn't get to when you studied economics.... Even though technology advances breathlessly, the economic principles we rely on are durable. The examples may change, but the ideas will not go out of date." In other words, there is no need to seek a "New Economy" with new principles for business and public policy, the old ones will serve if you apply them thoughtfully.

We should take a cue from Shapiro and Varian: recognize the new technology, but ask how the old principles, perhaps with new parameters, still apply. The unruliness of machine learning components may be greater than that of more conventional components, but we should expect to apply variants of the techniques we've developed for conventional software.

The AI revolution of machine learning will not be the last word in new computing paradigms. Quantum computing is waiting in the wings. It will bring new challenges for programming languages and system design. Will it be possible to map quantum algorithms to the structure of conventional programming languages, or will it be more effective to remain at the quantum assembly language level? At what level of abstraction will the quantum computation be incorporated in the hybrid program? Will it be possible to define formal semantics in the classical way? How will languages describe the iterative cycle in which an algorithm begins with a good guess about a good input state, runs the quantum computation, checks the result, adjusts the input state, and runs it again? How will languages deal with components that deliver results in the form of probability clouds? How will developers determine correctness, especially if their problems are too large for a simulator? What about errors that arise from intrinsic properties of the quantum computer itself? How will reasoning about modularity address the possibility of quantum entanglement between two components? [Valiron 2015, Ribeiro da Rosa and de Santiago 2020, Metwalli 2021]

The myth of the AI Revolution distracts software developers from the long tradition of ideas from AI flowing into programming and programming languages, with varying degrees of disruption. In particular, it distracts us recognizing familiar properties in ML systems and, as we have done in the past, adapting languages and software technologies to address them. A particular lesson is that it's time to rethink what we mean by "correctness" for a complex software system, especially a system rich in real-world data or an embedded system. Although the forcing function may be components that embed inexplicable ML algorithms, this is something we should have done long ago. Doing so will embrace, rather than deny, the inherently incomplete and noisy information about components we use to design systems.

Prometheus brings fire to man [Vogel 1910]

OUR MYTHS SHOULD INSPIRE US, BUT THEY SHOULD NOT HOLD US CAPTIVE

The image of programming languages captured in our myths arises from the traditional setting, in which general-purpose programming languages with precise specifications and well-defined semantics are used to create correct solutions to well-specified problems. These problems, the main focus of mainstream programming language research, are now a minority of software development problems. How should programming language research respond?

We looked at the myths of programming:

- Examining the myth of the **Professional Programmer** showed us that vernacular programmers vastly outnumber professional programmers. The myth distracts programming language designers from opportunities to improve software overall by bringing programming language design expertise to the notations and development processes that support the needs of vernacular programmers.
- Examining the myth that the **Code IS the Software** shows us that developing software of practical scale involves building coalitions of many types of elements, which interact through mechanisms outside the programming language. The myth distracts programming language designers from opportunities to improve the overall process of software development, especially by professionals, by improving the design of the associated tools and frameworks as well as the programming language. It also distracts programming language designers from opportunities to apply language design, abstraction, and modeling principles to software that is developed incrementally in an exploratory mode, especially in forms other than text, by vernacular software developers
- Examining the **Purity** myth of **Mathematical Tractability** shows us that even among professional programmers a high level of mathematical sophistication is scarce and that non-general-pur-

pose languages suffer from not being treated as first-class languages. The myth distracts programming language designers from the urgent need to provide abstractions that encapsulate formal reasoning in a way that provides its benefits without requiring the programmer to master the mathematics. The myth also distracts from opportunities to trade generality for power in a language while maintaining the formal rigor now largely reserved for general-purpose languages.

- Examining the **Purity** myth of **Correctness** shows us that the idealized specifications that serve as correctness criteria are usually impractical, and in many cases the standard of "fitness for task" is more appropriate than full correctness. The myth distracts programming language designers from considering the need to support reasoning with partial and unreliable information about extra-functional as well as functional properties—and from considering that it may be time to rethink what we mean by "correctness" for a complex software system, especially an embedded system.
- Examining the **Purity** myth of **Specification** shows us that exploratory programming—discovering, not just implementing a specification—is as important a mode of software development as the usual mode of coding to a given specification. The myth distracts programming language designers from supporting the exploratory mode of software development.
- Examining the myth of the **AI Revolution** shows us that machine learning components present a somewhat new face to the software developer, together with challenges arising from collecting and curating massive data sets. However, it appears that established techniques provide a solid basis for addressing these new manifestations. The myth distracts software developers from recognizing familiar properties in machine learning systems and adapting languages and software technologies to respond to the specific new variants.

We discussed several ways in which software development in the world—*se pragmos* (in vivo) —is richer than the software of our myths—*se mythos*.

se mythos	*se pragmos*
Professional developers	Vernacular developers
Code dominates	Many types of resources
General-purpose languages	Domain-specific languages
Programming languages and compilers	Ecosystems of tools and frameworks
Closed systems	Open resource coalitions
Specifications	Credentials
Correctness	Fitness for purpose
Code to specifications	Code in order to understand

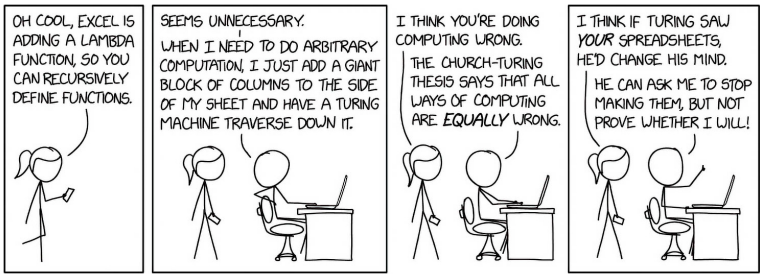
The pristine *se mythos* space lends itself to precise analysis, but opportunities for programming languages innovation abound in the larger *se pragmos* space. For example:

- Support the background and mindset of intended users. Hide the complexity and higher mathematics, for example with domain-specific abstractions. The advent of low-code environments suggests a niche in software development that presents an opportunity for providing better languages to a wide audience.
- Support exploratory programming that seeks understanding as well as programming to satisfy specifications.
- Bring the power of programming language design to scripting, markup, graphical, mashup, and other domain-specific languages and tools.
- Enrich type systems to elevate noisy or probabilistic data to first-class status in the language and to capture provenance of data from different sources. Make provisions for contingent

results that may change after the next time a machine learning model is retrained. Support contracts about behavior of data as part of the type system.

- Explore ways to determine how "good" (by whatever measure) software needs to be for a given application as well as how "good" a piece of software actually is, so that it's possible to evaluate "good enough" in context.

Above all, the programming language research community should make these improvements in a way that exploits the power of the formalism while still supporting the background, mindset, and workflow of the intended beneficiaries of these improvements. Capabilities grounded in formal mathematics have great power *se mythos*, but making them effective *se pragmos* requires packaging and presenting them in ways that provide the power without the obligation for users to master the formalism.



[xkcd 2453]

Myths have an enduring appeal that resists objective evidence. Sometimes changing the story behind a myth changes peoples' attachment to it. The programming language myths arise, I think, from a deep-rooted need for certainty. Indeed, it is possible to find certainty within a formal system. The world, however is uncertain—messy, nondeterministic, ambiguous, human. It does not lend itself to the precision required for formal definitions, and that prevents the certainties within our formal systems from being translated with confidence to the world. We should certainly continue to aspire to precision and completeness—but we should also provide support for practical software that runs in a context of inherent uncertainty.

Most of the challenges to our myths have been with us for a long time, but we have waved them off, treated them as special cases, apologized, or told ourselves that if we just tried really really hard we could bring our practical problems into the world imagined by our myths. It's long past time to move on from the myths—to embrace, rather than deny, incompleteness and uncertainty. It's time to recognize the prevalence of vernacular developers, the role of exploratory programming, and the need to embed theory in tools. Perhaps the **AI Revolution** will be the final straw that forces us to this point of view.

The **AI Revolution** will not destroy our model of software, I think, but we would benefit if the AI challenge motivates us to reconsider some of our myths and address some of the practical problems to which our myths may have blinded us.

Our myths should inspire us, but they should not hold us captive.

Athena holding spear and helmet, owl at left. [Brygos Painter 490–480 BCE]

ABOUT THE AUTHOR

Mary Shaw once designed traditional general-purpose programming languages, even languages with support for formal specification and verification, but realized that the real action is in the high-level organization of systems and turned to architecture description languages, whereupon the High Church of Programming Languages read her out as a heretic. In this journey to apostasy she

- declared global variables and aliasing to be evil, in the early 1970s
- renounced belief in the assignment axiom, in the mid 1970s
- accepted aliasing as dangerous but not evil, in the late 1970s
- recognized that types were actually added to programming languages to document broad intent and provide early warning of execution problems, in the early 1980s
- rejected the notion that objects are the one and true path, in the mid 1980s
- disavowed complete formally verified specifications of correctness, in the late 1980s
- embraced little languages for lightweight programming, in the early 1990s
- forsook formal elegance in favor of useful abstractions and hence advocated for first-class architectural connectors, in the mid 1990s
- prioritized fitness for task, relative to context, over pure correctness, in the late 1990s
- realized that most code doesn't matter enough to need rigorous verification, around 2000
- embraced problem-setting on a par with problem-solving in design, early in this century

She is the Alan J. Perlis University Professor of Computer Science at Carnegie Mellon University. She has received the United States' National Medal of Technology and Innovation, the ACM SIGSOFT Outstanding Research Award (with David Garlan), the IEEE Computer Society TCSE's Distinguished Educator and Distinguished Women in Software Engineering Awards, the George R. Stibitz Computer & Communications Pioneer Award, and CSEE&T's Nancy Mead Award for Excellence in Software Engineering Education. She is a Fellow of ACM and IEEE.

ACKNOWLEDGMENTS

This essay was written in tandem with my keynote talk for the 2021 History of Programming Languages Conference [Shaw 2021]. So it only got written because Richard Gabriel and Guy Steele provoked these reflections by inviting me to present that keynote and Richard shepherded the paper. Thanks to them and also to Marian Petre, Roy Weil, George Fairbanks, Eunsuk Kang, David Garlan, Ipek Ozkaya, David Newbury, Alison Langmead, Sara Metwalli, the software engineering group at Carnegie Mellon, and numerous participants in the 2021 History of Programming Language and the 2021 Pattern Languages of Programs Conferences for discussions of the issues, suggestions, and comments on various drafts. Thanks to Margaret Burnett, Greg Wilson, and a senior developer who wishes to remain unnamed for providing examples.

Bill Wulf started me on the path of looking skeptically at conventional wisdom and Anthony Hall reminded me that myths encode culture—that they are much, much more than mere tall tales.

Thanks to Randall Munroe for capturing important insights in xkcd and especially for publishing them under a Creative Commons attribution-noncommercial 2.5 license that makes them easy to incorporate here. Thanks also to The Met and Wikimedia Commons for providing a wealth of public domain or creative-commons-licensed artwork.

I deeply appreciate both moral and technical support from Roy Weil in production of the paper and support from Buck Caldwell and Josh Quicksall of the ISR staff in production of the HOPL video and the artwork in this paper.

Time for me to reflect on programming languages and our associated myths was available because of support from the Alan J. Perlis Chair of Computer Science at Carnegie Mellon University.

REFERENCES

Harold Abelson and Gerald Jay Sussman. 1996. *Structure and Interpretation of Computer Programs*, second edition. MIT Press, ISBN 0-262-01153-0. Also at https://mitpress.mit.edu/sites/default/files/sicp/full-text/book/book.html (accessed December 4, 2021).

Adobe. 2021. Creating Actions. Photoshop Documentation for recording actions. https://web.archive.org/web/20211018235502/https://helpx.adobe.com/photoshop/using/creating-actions.html Also at NON-ARCHIVAL https://helpx.adobe.com/photoshop/using/creating-actions.html (accessed December 4, 2021).

Autodesk. 2021. Overview: What isAutoCAD? https://web.archive.org/web/20211121040256/https://www.autodesk.com/products/autocad/overview Also at NON-ARCHIVAL https://www.autodesk.com/products/autocad/overview (accessed December 4, 2021).

Sigall K. Bell, Tom Delbanco, Joann G. Elmore, Patricia S. Fitzgerald, Alan Fossa, Kendall Harcourt, Suzanne G. Leveille, Thomas H. Payne, Rebecca A. Stametz, Jan Walker, and Catherine M. DesRoches. 2020. Frequency and Types of Patient-Reported Errors in Electronic Health Record Ambulatory Care Notes. *JAMA Network Open*. 2020;3(6):e205867 https://doi.org/10.1001/jamanetworkopen.2020.5867

Jayanti Bhandari, Ram P. Neupane, Yuheng Luo, Wesley Y. Yoshida, Rui Sun, and Philip G. Williams. 2019. Characterization of Leptazolines A–D, Polar Oxazolines from the Cyanobacterium Leptolyngbya sp., Reveals a Glitch with the "Willoughby–Hoye" Scripts for Calculating NMR Chemical Shifts. *Organic Letters* 2019 21 (20), 8449–8453. Publication date October 8, 2019, https://doi.org/10.1021/acs.orglett.9b03216

Just Bouhuijs. 2019. Gartner: *Citizen Developers strengthen impact of IT*, September 5, 2019. NON-ARCHIVAL https://wem.io/news/gartner-citizen-developers-strengthen-impact-of-it-2/ (accessed December 4, 2021).

Frederick P. Brooks Jr. 2010. *Design of Design: Essays from a Computer Scientist*. Pearson Education, ISBN 978-0201362985.

Margaret M. Burnett and Brad A. Myers. 2014. Future of end-user software engineering: beyond the silos. In *Future of Software Engineering Proceedings (FOSE 2014)*. Association for Computing Machinery, New York, NY, USA, 201–211. https://doi.org/10.1145/2593882.2593896

Confidential. 2021. Personal communication by email, May 25, 2021. NON-ARCHIVAL Email to Mary Shaw from a mathematically proficient senior developer at a major IT company who is well known to her but who wishes to remain unnamed. The comment summarizes a discussion about the frustration of dealing with programming languages that require mathematical sophistication to use the language.

Frank DeRemer and Hans Kron. 1975. Programming-in-the large versus programming-in-the-small. In *Proceedings of the International Conference on Reliable Software*. Association for Computing Machinery, New York, NY, USA, 114–121. https://doi.org/10.1145/800027.808431

Adam DuVander. 2010. 5 Years Ago Today the Web Mashup Was Born. April 8, 2010. https://web.archive.org/web/20200528234624/https://www.programmableweb.com/news/5-years-ago-today-web-mashup-was-born/2010/04/08 Also at NON-ARCHIVAL https://www.programmableweb.com/news/5-years-ago-today-web-mashup-was-born/2010/04/08 (accessed December 4, 2021).

Marc Fisher and Gregg Rothermel. 2005. The EUSES spreadsheet corpus: a shared resource for supporting experimentation with spreadsheet dependability mechanisms. In *Proceedings of the First Workshop on End-User Software Engineering (WEUSE I)*. Association for Computing Machinery, New York, NY, USA, 1–5. https://doi.org/10.1145/1083231.1083242

Richard P. Gabriel. 2020. Personal communication by email, March 11, 2020. NON-ARCHIVAL Email to Mary Shaw reporting on challenges of reasoning about correctness during exploratory programming.

Richard P. Gabriel. 2021. Personal communication by email, June 9, 2021. NON-ARCHIVAL Email to Mary Shaw reporting on the challenges of cleaning up noisy data in a publicly available data set.

Sean Gallagher. 2019. Researchers find bug in Python script may have affected hundreds of studies. *Ars Technica* October 15, 2019. https://web.archive.org/web/20210928233407/https://arstechnica.com/information-technology/2019/10/chemists-discover-cross-platform-python-scripts-not-so-cross-platform/ Also at NON-ARCHIVAL https://arstechnica.com/information-technology/2019/10/chemists-discover-cross-platform-python-scripts-not-so-cross-platform (accessed December 4, 2021).

David Garlan. 2010. Software engineering in an uncertain world. In *Proceedings of the FSE/SDP workshop on Future of Software Engineering Research (FoSER '10)*. Association for Computing Machinery, New York, NY, USA, 125–128. https://doi.org/10.1145/1882362.1882389

Gartner Research. 2021a. Gartner glossary: Citizen Developer. https://web.archive.org/web/20211102233208/https://www.gartner.com/en/information-technology/glossary/citizen-developer Also at NON-ARCHIVAL https://www.gartner.com/en/information-technology/glossary/citizen-developer (accessed December 4, 2021). Sets the scope for Gartner comments on low code development.

Gartner Research. 2021b. Gartner Forecasts Worldwide Low-Code Development Technologies Market to Grow 23% in 2021. February 16, 2021. https://web.archive.org/web/20211118153920/https://www.gartner.com/en/newsroom/press-releases/2021-02-15-gartner-forecasts-worldwide-low-code-development-technologies-market-to-grow-23-percent-in-2021 Also at NON-ARCHIVAL https://www.gartner.com/en/newsroom/press-releases/2021-02-15-gartner-forecasts-worldwide-low-code-development-technologies-market-to-grow-23-percent-in-2021 (accessed December 4, 2021). Describes anticipated growth in low-code development.

David Garlan, Robert Allen, and John Ockerbloom. 1995. Architectural mismatch; or, Why it's hard to build systems out of existing parts. In *Proceedings of the 17th International Conference on Software Engineering (ICSE '95)*. Association for Computing Machinery, New York, NY, USA, 179–185. https://doi.org/10.1145/225014.225031

Charles Hill, Rachel Bellamy, Thomas Erickson, and Margaret Burnett. 2016. Trials and tribulations of developers of intelligent systems: a field study. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 162–170. https://doi.org/10.1109/VLHCC.2016.7739680

Angela Horneman, Andrew Melinger, and Ipek Ozkaya. 2019. AI Engineering: 11 Foundational Practices. Carnegie Mellon Software Engineering Institute White Paper. https://web.archive.org/web/20210705174312/https://resources.sei.cmu.edu/asset_files/WhitePaper/2019_019_001_634648.pdf Also at NON-ARCHIVAL https://resources.sei.cmu.edu/asset_files/WhitePaper/2019_019_001_634648.pdf (accessed December 4, 2021).

IBM Systems Reference Library. 1971. IBM System/360 Operating System: Job Control Language Reference. Second edition, June 1971. https://web.archive.org/web/20210820005158/http://bitsavers.org/pdf/ibm/360/os/R20.1_Mar71/GC28-6704-1_OS_JCL_Reference_Rel_20.1_Jun71.pdf Also at NON-ARCHIVAL http://www.bitsavers.org/pdf/ibm/360/os/R20.1_Mar71/GC28-6704-1_OS_JCL_Reference_Rel_20.1_Jun71.pdf (accessed December 4, 2021).

Ciera Jaspan, Michael Keeling, Larry Maccerone, Gabriel L. Zenarosa, and Mary Shaw. 2009. Software Mythbusters Explore Formal Methods. *IEEE Software*, vol. 26, no. 6, pp. 60–63, November–December 2009, https://doi.org/10.1109/MS.2009.188

Christian Kästner and Eunsuk Kang. 2020. Teaching software engineering for AI-enabled systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET '20)*. Association for Computing Machinery, New York, NY, USA, 45–48. https://doi.org/10.1145/3377814.3381714

Christian Kästner, Eunsuk Kang, and Sven Apel. 2021. Feature Interactions on Steroids: On the Composition of ML Models. 2021. arXiv:2105.06449v1 [cs.SE]

Amy J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. 2011. The state of the art in end-user software engineering. *ACM Computing Surv*eys 43, 3, Article 21 (April 2011), 44 pages. https://doi.org/10.1145/1922649.1922658

Michael R. Lyu. 1996. *Handbook of Software Reliability Engineering*. IEEE Press and McGraw Hill, ISBN 978-0070394001. Also at http://www.cse.cuhk.edu.hk/~lyu/book/reliability/index.html

Sara Metwalli. 2021. Personal communication by email, May 3–18, 2021 NON-ARCHIVAL Email discussion with a member of the Keio University Quantum Computing Group about parallels between the challenges of AI and of quantum computing to the myths of programming languages.

Benjamin W. Nelson, Carissa A. Low, Nicholas Jacobson, Patricia Areán, John Torous, and Nicholas B. Allen. 2020. Guidelines for wrist-worn consumer wearable assessment of heart rate in biobehavioral research. *NPJ Digital Medicine*, 3, 90. https://doi.org/10.1038/s41746-020-0297-4

National Telecommunications and Information Administration (NTIA), U.S. Department of Commerce. 2021. Software Bill of Materials. https://web.archive.org/web/20211029190723/https://www.ntia.gov/SBOM Also at NON-ARCHIVAL https://www.ntia.gov/sbom (accessed December 4, 2021).

Ipek Ozkaya. 2020. What is really different in engineering AI-enabled systems? *IEEE Softw*are 37, 4 (July–August 2020), 3–6. https://doi.org/10.1109/MS.2020.2993662

David L. Parnas. 1972a. A technique for software module specification with examples. *Communications of the ACM* 15, 5 (May 1972), 330–336. https://doi.org/10.1145/355602.361309

David L. Parnas. 1972b. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 15, 12 (December 1972), 1053–1058. https://doi.org/10.1145/361598.361623

Pew Charitable Trusts. 2018. *Enhanced Patient Matching Is Critical to Achieving Full Promise of Digital Health Records.* October 2018. https://web.archive.org/web/20210707161718/https://www.pewtrusts.org/-/media/assets/2018/09/healthit_enhancedpatientmatching_report_final.pdf Also at NON-ARCHIVAL https://www.pewtrusts.org/-/media/assets/2018/09/healthit_enhancedpatientmatching_report_final.pdf (accessed December 4, 2021).

Orna Raz, Rebecca B. Buchheit, Mary Shaw, Philip Koopman, and Christos Faloutsos. 2004. Detecting semantic anomalies in truck weigh-in-motion traffic data using data mining. *Journal of Computing in Civil Engineering* 18 (2004): 291–300. https://doi.org/10.1061/(ASCE)0887-3801(2004)18:4(291)

Evandro Chagas Ribeiro da Rosa and Rafael de Santiago. 2020. Classical and Quantum Data Interaction in Programming Languages: A Runtime Architecture, May 2020. arxiv.org:2006.00131 [quant-ph] .

Horst W. J. Rittel and Melvin M. Webber. 1973. Dilemmas in a general theory of planning. *Policy Sciences* 4, 155–169 (1973). https://doi.org/10.1007/BF01405730

J. H. Saltzer, D. P. Reed, and D. D. Clark. 1984. End-to-end arguments in system design. *ACM Transactions on Computer Systems* 2, 4 (November 1984), 277–288. https://doi.org/10.1145/357401.357402

Christopher Scaffidi, Mary Shaw, and Brad Myers. 2005. Estimating the numbers of end users and end user programmers. 2005 *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '05)*, 2005, pp. 207–214, https://doi.org/10.1109/VLHCC.2005.34

Christopher Scaffidi and Mary Shaw. 2007. Developing confidence in software through credentials and low-ceremony evidence. *International Workshop on Living with Uncertainties*, at 23rd International Conference on Automated Software Engineering. Presented at workshop, not in digital library. Online at NON-ARCHIVAL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.407.4357&rep=rep1&type=pdf (accessed December 4, 2021).

Chris Scaffidi. 2017. Counts and earnings of end-user developers. NON-ARCHIVAL https://www.linkedin.com/pulse/counts-earnings-end-user-developers-chris-scaffidi/ (accessed December 4, 2021). Discussion of personal costs and benefits for vernacular programmers of learning new software skills; generally corroborates [Scaffidi et al 2005]).

Donald A. Schön. 1984. *The Reflective Practitioner: How Professionals Think in Action*. Basic Books (1984). ISBN 978-0465068784.

Carl Shapiro and Hal R. Varian. 1999. *Information Rules: A Strategic Guide to the Network Economy*. Harvard Business Review Press, 1999. ISBN 978-0875848631.

Mary Shaw. 1990. Prospects for an engineering discipline of software. *IEEE Software*, vol. 7, no. 6, pp. 15–24, November 1990. https://doi.org/10.1109/52.60586

Mary Shaw. 1996. Truth vs Knowledge: the difference between what a component does and what we know it does. In *Proceedings of the 8th International Workshop on Software Specification and Design (IWSSD '96)*. IEEE Computer Society, USA, 181. https://doi.org/10.1109/IWSSD.1996.501165

Mary Shaw. 2015. Progress toward an engineering discipline of software. Software Engineering Institute Architecture Technology User Network (SATURN) Conference, invited keynote. NON-ARCHIVAL https://www.youtube.com/watch?v=S03bsjs2YnQ (accessed December 4, 2021).

Mary Shaw. 2021. Myths and Mythconceptions: What does it mean to be a programming language, anyhow? Keynote presentation, *Fourth SIGPLAN History of Programming Languages Conference*. https://dl.acm.org/toc/pacmpl/2020/4/HOPL (accessed March 1, 2022).

Herbert A. Simon. 1996. *The Sciences of the Artificial*. MIT Press 1996. ISBN 13: 9780262691918.

Springer Journal of Software and Systems Modeling. 2021. Theme section on Modeling in Low-Code Development Platforms. Call for contributions. https://web.archive.org/web/20210703010301/https://sosym.org/theme_sections/cfp/cfp-SoSyM-TS-MLCDP21.pdf Also at NON-ARCHIVAL https://www.sosym.org/theme_sections/cfp/cfp-SoSyM-TS-MLCDP21.pdf (accessed December 4, 2021).

Stack Overflow. 2017. Mapping Ecosystems of Software Development. https://web.archive.org/web/20210506153925/https://stackoverflow.blog/2017/10/03/mapping-ecosystems-software-development/ Also at NON-ARCHIVAL https://stackoverflow.blog/2017/10/03/mapping-ecosystems-software-development (accessed December 4, 2021).

Stack Overflow. 2020. Annual survey 2020. https://web.archive.org/web/20211121035038/https://insights.stackoverflow.com/survey/2020 Also at NON-ARCHIVAL https://insights.stackoverflow.com/survey/2020#education (accessed December 4, 2021).

Don Syme. 2021. Comment on issue #243, Support type classes or implicits, September 9 2021. https://web.archive.org/web/20210914010432/https://github.com/fsharp/fslang-suggestions/issues/243 Also at NON-ARCHIVAL https://github.com/fsharp/fslang-suggestions/issues/243#issuecomment-916079347 (accessed December 4, 2021). Don Syme explains the downsides of type classes and the technical and philosophical reasons for not implementing them in F#.

Techtarget. 2021. Definition of software stack. https://web.archive.org/web/20210816151809/https://searchapparchitecture.techtarget.com/definition/software-stack Also at NON-ARCHIVAL https://searchapparchitecture.techtarget.com/definition/software-stack (accessed December 4, 2021).

Benoît Valiron, Neil J. Ross, Peter Selinger, D. Scott Alexander, and Jonathan M. Smith. 2015. Programming the quantum future. *Communications of the ACM* 58, 8 (August 2015), 52–61. https://doi.org/10.1145/2699415

Luiz A. Virginio Jr and Ivan Luiz Marques Ricarte. 2015. Identification of patient safety risks associated with electronic health records: a software quality perspective. In *Proceedings of MEDINFO 15 eHealth-enabled Health*. https://doi.org/10.3233/978-1-61499-564-7-55 *Studies in Health Technology and Informatics* 2015;216:55–9. Also at National Library of Medicine https://pubmed.ncbi.nlm.nih.gov/26262009/ PMID: 26262009.

Willemien Visser. 2006. *The Cognitive Artifacts of Designing*. CRC Press 2006. ISBN 978-0805855111.

Robert Wang, Gordon Blackburn, Milind Desai, et al. 2017. Accuracy of Wrist-Worn Heart Rate Monitors. *JAMA Cardiol*ogy 2017;2(1):104–106. https://doi.org/10.1001/jamacardio.2016.3340

Greg Wilson. 2021. Personal communication by email, May 2, 2021. NON-ARCHIVAL Email to Mary Shaw about how scientific programmers develop confidence in their software.

Wordpress. 2021. Product introduction. NON-ARCHIVAL https://wordpress.com/ (accessed December 4, 2021).

ARTWORK AND FIGURES: SOURCES AND PERMISSIONS

Anonymous. 17th century. The fall of Icarus, Musee Antoine Vivenel. Photo by Wmpearl 2010, public domain image via Wikimedia Commons. https://commons.wikimedia.org/wiki/File:%27The_Fall_of_Icarus%27,_17th_century,_Mus%C3%A9e_Antoine_Vivenel.JPG (accessed November 17, 2021).

Jayanti Bhandari, Ram P. Neupane, Yuheng Luo, Wesley Y. Yoshida, Rui Sun, and Philip G. Williams. 2019. Characterization of Leptazolines A–D, Polar Oxazolines from the Cyanobacterium Leptolyngbya sp., Reveals a Glitch with the "Willoughby-Hoye" Scripts for Calculating NMR Chemical Shifts. *Organic Letters* 2019 21 (20), 8449–8453. Publication date October 8, 2019, https://doi.org/10.1021/acs.orglett.9b03216 (figure from abstract, fair use for purpose of criticism, accessed November 17, 2021).

Brygos Painter (attr). 490-480 BCE. Terracotta lekythos (oil flask). Athena holding spear and helmet, owl at left. ca 490–480 BCE. Held by The Met (Metropolitan Museum of Art), New York, accession Number: 09.221.43. Public domain image provided by the The Met https://www.metmuseum.org/art/collection/search/248181 (accessed November 17, 2021).

Robert Willemsz de Baudous (attr). 17th century. Apollo Killing the Python (at Delphi). Engraving possibly by Robert Willemsz de Baudous (Netherlandish, 1574/5–1659) after Hendrick Goltzius (Netherlandish, Mühlbracht 1558–1617 Haarlem). Held by The Met (Metropolitan Museum of Art), New York, accession Number: 2012.136.479. Public domain image provided by The Met https://www.metmuseum.org/art/collection/search/398703 (accessed November 17, 2021).

Domenichino. 1602. The Maiden and the Unicorn. Palazzo Farnese, Rome. Public domain image via Wikipedia commons. https://commons.wikimedia.org/wiki/File:DomenichinounicornPalFarnese.jpg (accessed February 19, 2022).

John Flaxman. 1910. Pandora öffnet das Gefäß, 1910. Hesiod, *Werke und Tage* V. 94. "Pandora opens the vessel in which sufferings and evils were locked. You defocus, only hope remains". Schwab, Legends of classical antiquity I. 5. Public domain image via Wikipedia commons. https://commons.wikimedia.org/wiki/File:Flaxman%27s_Zeichnungen_1910_007.jpg (accessed November 17, 2021).

James Freeman. 2021. Complete Guide to Architecture Diagrams. https://web.archive.org/web/20211107180232/https://www.edrawsoft.com/architecture-diagram.html Also at NON-ARCHIVAL https://www.edrawsoft.com/architecture-diagram.html (fair use for the purpose of criticism, accessed November 17, 2021).

Norbert Landsteiner. 2021. The Virtual Card Read-Punch. https://web.archive.org/web/20210726130535/https://www.masswerk.at/card-readpunch/ Also at NON-ARCHIVAL https://www.masswerk.at/card-readpunch/ App for creating images of punch cards from text. Permission to use images produced with this tool granted by Norbert Landsteiner in email September 22, 2021. Text from [IBM 1971, p.114].

NPR. 2008. Vote Report: Help NPR Identify Voting Problems. Nov 4, 2008. NON-ARCHIVAL https://www.npr.org/templates/story/story.php?storyId=96349881 (fair use for purpose of criticism, accessed November 17, 2021).

Peter Paul Rubens. 1636–38. Vulcan forging the Thunderbolts of Jupiter, ca 1636–38. Museo del Prado, accession number P001676. Public domain image via Wikimedia Commons. https://commons.wikimedia.org/wiki/File:Rubens_-_Vulcano_forjando_los_rayos_de_J%C3%BApiter.jpg (accessed November 17, 2021).

Stack Overflow. 2019. Developer Survey Results 2019: Correlated Technologies. https://web.archive.org/web/20220208133924/https://insights.stackoverflow.com/survey/2019#correlated-technologies Also at NON-ARCHIVAL https://insights.stackoverflow.com/survey/2019#correlated-technologies (fair use for purpose of criticism, accessed November 17, 2021).

UML. 2021. UML Diagrams https://web.archive.org/web/20211006104724/https://www.uml-diagrams.org/uml-25-diagrams.html Also at NON-ARCHIVAL https://www.uml-diagrams.org/uml-25-diagrams.html (fair use for purpose of criticism, accessed November 17, 2021).

Hugo Vogel. 1910. Prometheus bringt den Menschen das Feuer, 1910. Wall painting in the industrial hall of the German Department of the 1910 World Exhibition in Brussels. Public domain image via Wikipedia Commons. https://commons.wikimedia.org/wiki/File:Hugo_Vogel_-_Prometheus_bringt_den_Menschen_das_Feuer,_1910.jpg (accessed November 17, 2021).

Robert Wang, Gordon Blackburn, Milind Desai, et al. 2017. Accuracy of Wrist-Worn Heart Rate Monitors. *JAMA Cardiology* 2017;2(1):104–106. https://doi:10.1001/jamacardio.2016.3340 (fair use for purpose of criticism, accessed November 17, 2021).

xkcd. 224-2453 Randall Munroe. *xkcd. A webcomic of romance, sarcasm, math, and language.* NON-ARCHIVAL https://xkcd.com/ This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. (all accessed November 5, 2021)
224: *Lisp* https://xkcd.com/224/ February 16, 2007.
327: *Exploits of a Mom* https://xkcd.com/327/ October 10, 2007.
353: *Python* https://xkcd.com/353/ December 5, 2007.
676: *Abstraction* https://xkcd.com/676/ December 16, 2009.
974: *The General Problem* https://xkcd.com/974/ November 7, 2011.
1425: *Tasks* https://xkcd.com/1425/ September 24, 2014.
1838: *Machine Learning* https://xkcd.com/1838/ May 17, 2017.
2054: *Data Pipeline* https://xkcd.com/2054/ October 3, 2018.
2347: *Dependency* https://xkcd.com/2347/ August 17, 2020.
2453: *Excel Lamda* https://xkcd.com/2453/ April 21, 2021.