Could No-Code be Code?

Toward a No-Code Programming Language for Citizen Developers

Assaf Avishahar-Zeira
assaf@too.software
TOO.Software
Bnei Atarot 6099100, Israel

David H. Lorenz
lorenz@openu.ac.il
Dept. of Mathematics and Computer Science
Open University of Israel
Ra'anana 4353701, Israel

Abstract

By 2030 for each filled position in *Software Engineering*, two positions would remain unfilled. This already apparent loss of productivity has the software industry scrambling to fill the missing positions with *citizen developers*—technical people with little or no programming skills—who would be using No-Code platforms to program various software solutions in specific domains. However, currently available platforms have fairly limited abstractions, lacking the flexibility of a *general purpose* programming language.

To break the No-Code abstraction barrier, a very simple yet expressive general purpose No-Code programming language might provide citizen developers with an alternative to domain-specific No-Code platforms. Unfortunately, these requirements seem contradictory. Making a language very simple and specific might render it crippled, thus limited to a certain domain of problems. Conversely, making a language very expressive and general, might render it too complicated for citizen developers.

In this work we argue that a multi-paradigm minimalist approach can bridge the gap between simplicity and expressiveness by including only abstractions considered intuitive to citizens. As a concrete proof-of-concept, we present a general purpose programming language designed for citizen developers that is on the one hand very powerful and on the other hand very simple. In fact, this language is so simple that the entire development is accomplished by flowcharts using mouse actions only, without typing a single line of code, thus demonstrating a general purpose No-Code programming language candidate for citizen developers.

CCS Concepts: • **Software and its engineering** → *Visual languages*; *Multiparadigm languages.*

Onward! '23, October 25–27, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0388-1/23/10.
https://doi.org/10.1145/3622758.3622893

Keywords: Citizen Developers, Golang, No-Code Software Development, Projectional Editing, Programming Language Design.

ACM Reference Format:
Assaf Avishahar-Zeira and David H. Lorenz. 2023. Could No-Code be Code? Toward a No-Code Programming Language for Citizen Developers. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '23), October 25–27, 2023, Cascais, Portugal*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3622758.3622893

1 Introduction

Today and for the foreseeable future, the supply of professional programmers cannot meet the demand for software engineers [5]. SlashData,[1] a leading analyst company in the developer economy, projects a total number of 45 million software engineers globally by year 2030. The U.S. Labor Department further estimates a shortage of 85 million engineers by that time, meaning that for each filled position two would remain unfilled, and that because of this shortage companies may lose $8.4 trillion revenue.

This shortage in programmers is pushing the software industry toward *No-Code tools* that enable software development by novice programmers and even non-programmers, generally referred to as *citizen developers* [29] (hereafter, *citizens*). These No-Code tools keep the promise of creating a solution without typing code, but they are by far less expressive than a full blown *general purpose programming language (GPL)*, lacking any pretension to be *Code*.

In fact, No-Code tools resemble hardware more than software development, missing the most important property of software being "*soft*." In order to retain the "softness" property, they must be expressive like a programming language; that is, be *Code*. At the same time they must be simple enough for citizens; that is, to also be *No-Code*. This begs the question: *could No-Code be Code?*

1.1 General Purpose Programming Languages

The past 70 years have witnessed an enormous investment in programming languages. Thousands of books and scientific papers were written, and hundreds of languages were

[1] https://www.slashdata.co

developed. Noticeably, on average, one in five Turing Award citations attributes contribution to programming languages.

Furthermore, development of programming languages is not an episode of the past as new languages are constantly being invented. For example, TypeScript[2] and Rust [22] were invented in the last decade. Rust gained popularity for emphasizing performance, type safety, and concurrency, and TypeScript gained popularity for adding syntax on top of JavaScript [9], allowing developers to add types.

However, the evolution of programming languages is not necessarily going in the direction of simplification. TypeScript and Rust, for instance, can be considered among the most complicated ever invented. It seems therefore that since the invention of the first compiled programming language, all popular GPLs share roughly the same complexity [18], moving in the path set by Böhm [4] in 1951 in his Ph.D. dissertation.

1.2 Simplicity is Complicated

For No-Code development, language simplicity is an essential albeit elusive property. There were some attempts to create very simple languages by keeping the language small. However, a small size of a programming language does not necessarily make it easy to use or learn. Brainfuck [7], for example, is regarded as one of the smallest programming languages with only eight commands. However, it is no more than an esoteric attempt. Coding in Brainfuck is a nightmare, the generated code is completely unreadable, and the performance is poor.

A language can be very small and highly expressive and still lack simplicity. For example, a Turing machine has only few language constructs while exhibiting Turing-complete expressiveness, although Turing machines are far from being easy to use for everyday programming.

A successful attempt to create a simple and practical GPL was put forward by the three software engineers that invented the Go language [6]. In order to keep the language simple, Griesemer, Pike, and Thompson, each had the power to veto new features [24]. This kept Go at about two times smaller than other popular languages at that time, such as C++ [31] and Java [3]. Notably, they reported that simplicity is complicated [26].

Go excels in the domain of scalable, cloud-based servers that are optimized for performance; its light-weight *go routines* enable massive multithreading and performance under pressure. Go creates an efficient code, although C [16] and Rust are better in that respect. Go is fairly simple; an experienced programmer can pick it up over a weekend.

In contrasts to Go, our aim is to define a GPL that would make a big difference in how citizens use No-Code to build software solutions. But could a pragmatic GPL be made any simpler than Go?

[2] https://github.com/microsoft/TypeScript

1.3 Contribution

We present a novel, very simple, general purpose, No-Code programming language, named *Too (Things Object-Oriented)*.[3] The Too language enables both citizen developers and professional programmers to create software solutions in a simple manner, using drag-and-drop components, without having to write code explicitly.

Too was influenced by the powerful and compact Go language, but it is not a subset of Go. It contains the key features of Go, such as concurrency, interfaces, inheritance, as well as features that Go does not support, such as function overloading, default values for formal arguments, and error handling by catch/throw-like events (whose absence in Go is reported as drawbacks).[4] With all these additional features and more, Too is about 10 times smaller than Go (Tab. 1).

Creating a new language is by no means a lightweight decision. An alternative could have been to start with Go, or some other GPL, and strip off features that are deemed inessential for citizens or unmanageable for No-Code development. Shrinking an existing language may be a more effective approach than devising a completely new language. However, the resulted language would be limited to only features found in Go. Instead, the features selected for the Too language and for the software development methodology at the core of the No-Code solution are geared toward producing programs that citizens can read, develop, debug, and evolve. For example, the Too language adopts software architectures such as *event-driven*, *component-based*, and *cloud programming* that Go lacks.

Outline. Sect. 2 states the goal of Too, setting criteria for an ideal GPL for citizens. Sect. 3 reviews the underlying principles for the language design. Sect. 4 describes the structure of programs in the language. Sect. 5 describes the cloud development environment and its use of projectional editing and thing broker. Sect. 6 reviews the language internals. Sect. 7 describes deployment and current use.

2 Objective

The goal of Too is to provide citizen developers with a No-Code GPL. No-Code and a GPL may sound like two things that cannot both be achieved, an oxymoron. It entails two seemingly opposite requirements: domain-specific-like simplicity and general-purpose-like expressibility. On the one hand, in order for the language to be accepted by citizens, it better be simple; it should enable easy entry, creating simple programs in minutes. On the other hand, assuming that citizens' software needs are like any other software needs, the language must also be expressive enough, enabling citizen developers to evolve and create high-end solutions that

[3] https://too.software
[4] https://www.toptal.com/go/4-go-language-criticisms

Table 1. Language comparison [8]

Language	Keywords	Operators	Syntax rules	Year
Fortran [23]	39	16	~170	1957
C [16]	32	27	~100	1970
C++ [31]	90	35	~200	1979
Python [35]	33	39	~90	1991
Java [3]	51	34	~250	1995
Ruby [34]	13	28	~60	1995
C# [12]	78	41	~220	2000
Go [6]	25	34	~100	2009
TypeScript[5]	50	35	~150	2012
Rust [22]	35	45	~250	2015
Too	*0*	*5*	*10*	*2022*

involve concurrency, synchronization, complex algorithms, and complex data-structures.

Preferably, we would like to see citizens and professionals alike appreciating the many benefits a No-Code GPL brings. Citizens would find the visual projectional editing easy and productive to use, metaphorically as easy and as productive as using a spreadsheet, with only mouse actions, such as drag-and-drop and selection from drop-down menus. Professionals would also use the line-oriented text-based structural editing and advanced features, just as a spreadsheet can also offer advanced operations.

2.1 Limitation of Current No-Code Platforms

No-Code tools have many benefits [32], such as increased productivity and accessibility, and some tools may be better suited for certain types of applications or users. However, none of them provides a GPL for citizens:

Limited functionality No-Code tools typically provide a limited set of features and functionality compared to traditional programming languages. This means that citizens may not be able to build more complex applications or perform advanced customization [30].

Lack of flexibility No-Code tools are sometimes not flexible enough to accommodate unique business requirements or workflows. Users may thus need to adapt their processes to fit within the limitations of the No-Code tool, rather than being able to customize the tool to meet their specific needs [27].

Limited control over code No-Code tools often abstract the underlying code from the user, which means that users have limited control over the code, and may not be able to optimize or troubleshoot it.

Limited scalability No-Code tools may not be able to scale to support large or complex applications. Users may need to switch to traditional programming languages or tools as their application grows in complexity [14].

Vendor lock-in Many No-Code tools are proprietary and may not be easily transferable to other platforms. This can create a dependency on the tool and vendor, which may limit options for scaling or expanding the application in the future [20].

2.2 Desiderata for a Citizens' GPL

In designing a new No-Code language with the citizens' perspective in mind, we assume that citizens would have the following expectations from the GPL:

Expressiveness An expressive GPL provides the necessary features and constructs to let citizen developers express complex ideas and algorithms in a clear and concise way. It typically includes a wide range of abstraction mechanisms for programming and sometimes even for meta-programming.

Readability The generated code should be human readable and suitable for citizen comprehension, somewhat like a natural language. It means that preferably algorithms should be presented in a graphical manner (such as a flowchart, for example) instead of line-oriented text. This also means getting rid of unnecessary symbols and words, such as "`.`", "`;`", and "`this`," and many complex operators, such as "`+=`" and "`!==`," that are common in many GPLs. It means using icons to illustrate things. It means getting rid of parentheses in function calls with no arguments (e.g., use `now` instead of `now()`). It means avoiding weird programmer's abbreviations and refraining from using naming conventions, such as `nightTime` or `night_time`, in favor of simply using `night time` or night·time. It also means code coloring and hierarchical tool-tips.

Intuitive and familiar The language should be based on known concepts, such as spreadsheets, rule-based systems, and directory listings; all are considered abstractions that people grasp easily. Spreadsheets have no learning curve for entry level, but also pack built-in power for professional use. Rules-based systems are used in expert systems by common users and do not require any prior knowledge in programming. They could be found in email filters, parental control applications, routers, and more. They present a very simple logic: if something happens, do this and that. Directory listing is a known concept admitted by common users.

Easy to learn Preferably, the language should be based on a small set of keywords, operators and syntax rules. It takes a lot of time for a professional programmer to assimilate all these constructs and master a new language. Citizens do not have this privilege as they may not be practicing development daily. If the grammar is simple enough, citizens would more likely remember it and use it.

Well documented Citizens should find the necessary information handy. This means lots of tool-tips, well documented libraries, and documented tutorials [1, 2].

Easy to debug Citizens may not even be aware of the notion of *bug* let alone use a debugger with all the controls. This means that the debugger should be simple and intelligent.

Easy versioning Citizens must gain confidence that should something go wrong, they can always go back with ease to a recent working version. It also means that labeling should be simple and handy.

Simple to share and reuse Citizens are not full-time developers and therefore rely on reuse as much as possible. Their involvement in development is minimal, confining to the *last mile*, in order to get the job done. The development environment should enable efficient search and evaluation of existing libraries.

Simple to collaborate Citizens should be able to cooperate on any program with others and especially with professionals through a mentoring partnership. This calls for cloud programming and instant web collaboration without complex installation and configuration [15].

Responsiveness Citizens are not aware of compilation, and therefore, their program should appear to be ready to run at all times, except when fixing inconsistencies (e.g., when removing an instance variable or a method, leaving the logic inconsistent with behavior that depend on them).

Hot plugging In order to reduce downtime, when running a newly created software component, citizens would appreciate an execution model that enables the addition, modification, or replacement of components, while keeping other internal components intact; that is, leaving them running, waving the requirement to reboot them.

Live programming Citizens are more productive in a fluid and interactive programming environment. Reducing the programming feedback loop would enable citizens to instantly see the immediate runtime effects of their code changes as they program.

Language locality Preferably citizens would like to program in their native tongue. Also, there exist an abundant number of potential citizens who are technical but simply do not know English.

3 Language Design Principles

The Too language is multi-paradigm, assimilating different programming approaches to provide a simple programming style for citizens.

3.1 Thing-based

Too refers to objects as things. In Too everything is made of "things" without exception; from a simple register that holds a boolean variable, to a learning machine such as chatGPT that crunches numbers, all are regarded as things. This simplification makes basic types (such as *int*, *float*, etc.) redundant, and altogether makes it easier for developers to assimilate object-oriented principles.

Too supports the duality between composition (*has-a* relationship) and implementation inheritance (*is-a* relationship). As in the Go language, the two have the same syntax, so they both exist at the same time. This way two users may refer to the same Too program, and one will see inheritance, while the other will see composition. For example, in the program `switch { relay }`, one user will say "a switch has a relay," while another will say "a switch is a relay." Both interpretations are legitimate.

A Too program defines a *single* thing, unlike many other object-oriented languages that allow a program to have multiple objects at the root level. This simplification creates very short programs that are concentrated on a single idea.

3.2 Event-driven

An event driven architecture is often perceived more natural and hence would more likely be embraced by citizens. Most GPLs adopt an explicit invocation architecture: one method calls another method and instructs it to perform some action or to retrieve some required information. But often the real world works differently. A company receives a new order; the road segment is congested; the ship is docking; a web server receives a request for a Web page, etc. In neither case did the system schedule or request the action. Instead the event occurred based on an external action or activity. To accommodate this, Too implements an implicit invocation architecture with event processing.

3.3 Third-party Composition

In Too things are reusable software components that are subject to composition by third parties [33]. In contrast to a monolithic system in which all the things are running as a unified entity over a single build, things in Too are deployed independently of one another, and they interact by communication rather than by standard function calls [19]. Hence some parts of the program could be activated while other parts remain intact.

For example, this would allow the number·parameter thing to be updated while the system is running, or to update the conveyor thing while the production·line thing is running. Such decoupling makes the system robust, extensible, replaceable, and live [11]. It also reduces overall system downtime.

3.4 Marketplace for Things

To ensure portability, Too enforces a strict decoupling of universal code from sensitive domain-specific information (that might be confidential). In this way, with a click of a button, the developer may upload a thing to the marketplace, with the desired price tag, desired visibility (grant access to specific developers, to the organization, or to the public),

sanitized from domain-specific information, ready to be used by other developers, like a shared library item.

In addition, Too programs are typically very short, encouraging a much larger audience to get involved, to review the code, to leave a 1–5 star rating, to suggest different interfaces, names, and even icons, creating in this way a democratic marketplace for citizens.

3.5 Simple Programming

Too strives for simplicity in its syntax, editing, data structures, control flow, operators, and multilingual support.

3.5.1 Simplified syntax. Too is simplified to the bare minimum. Too aims to be the world's smallest practical GPL, with no keywords, five operators (Figures 1 and 2), and ten syntax rules (Fig. 3). In comparison, Go, which is a fairly small language, has 25 keywords, 34 operators, and about 100 syntax rules. This facilitates rapid acquisition of the language and rapid development.

3.5.2 Simplified editing. Writing programs in Too is made easy thanks to the use of projectional editing. Projectional editors are editors that modify the AST model of the program through a projection to a view [10]. The projected view can be textual or visual. Projectional editors allow modifications to the AST which are visualized back as changes to the view.

3.5.3 Simplified data-structures. A thing in Too can only own a single collection of minor things (instance variables). The language does not allow multiple data structures, let alone nested data structures and global data structures.

Banning nested structures might be perceived at first as a hit, since it requires to move the nested structure to an external thing. However, this may enforce careful design: "should I have two arrays, one for departure:tick and another for arrival:tick, or should I have a single array of a new thing called flight that contains these two fields?"

3.5.4 Simplified control flow. Expert systems especially, and rule-based systems in general, present a simplified approach to flow control which is human-friendly and easy to master. In such systems the logic is structured in rules with a trigger and a list of sequential actions.

While most GPLs advocate loops that considerably increases citizen incomprehension, Too provides many ways to create an iterative logic but it is *loop-free*. The algorithms in Too may only contain downstream branching. This means that rule actions (or function actions) are executed in a sequential order and there is no way to go back and re-execute an action that was already executed.

Nevertheless, in order to buy-in experienced programmers that are used to loop statements, Too provides a gateway for *advanced developers* with the iterative ("*") operator. Simply add the operator at the beginning of an action, and it becomes iterative. This would convert, for example, an *if-then* conditional action to a classical *while-do* loop. It is unlikely, however, that citizens will need it or use it.

=	Initial value for thing in declaration
*	Wildcard and iteration
?	Decision point
[ ]	Subscript
→	Redirect

Figure 1. Operators

Id	Identifier
Str	Singe or double quoted string
Num	Number
Const	*Num* or *Str*

Figure 2. Tokens

3.5.5 Simplified operators. It is assumed that citizens are familiar with spreadsheets and this reassures the use of functions (e.g., `AND(x,y)`, `IF(x,y,z)`) together with fluent style chaining that is natural for citizens, refraining completely from infix binary operators that might be difficult to remember (e.g., "<<="), solving precedence of operations and simplifying readability. For example, the expression:

```
(a + b) * c
```

would be coded by developers as:

```
a plus(b) times(c)
```

3.5.6 Simplified localization and multilingual support. Localization is the process of adapting the programming language to the citizen's native tongue. This can make it reachable for a larger audience of citizen developers, giving them a smooth entry to the world of software development.

The syntax of Too by itself does not limit the code to the English language since it does not contain any keywords and it supports UTF-8 identifiers. For example, instead of writing:

```
pi times(radius squared)
```

Greek developers would write:[6]

```
π φορές(ακτίνα τετράγωνο)
```

Shifting to a different language only requires translating the development environment once per language and, of course, translating the catalog of things. The translation of the catalog is scalable since it is done by developers. A developer that places a new thing in the marketplace and wishes to make it available in a different language should take care of the translation, or mark it for auto-translate.

4 Language Definition

The grammar of Too has ten syntax rules, displayed in Fig. 3 in the notation of EBNF. An example of a logger program

[6]Pronounced "*pi fores aktina tetragono*."

$Thing ::= Id\ \text{“\{”}\ \{Decl \mid Event \mid When \mid Func\}^*\ \text{“\}”};$	$\mathbf{Id} \times \mathbf{Decl}^* \times \mathbf{Event}^* \times \mathbf{When}^* \times \mathbf{Func}^* = \mathbf{Thing}$
$Event ::= Id\ Params;$	$\mathbf{Id} \times \mathbf{Params} = \mathbf{Event}$
$When ::= \{Id^{\text{src}}\ [\text{“[”}\ Decl\ \text{“]”}]\ \text{“.”}\}^+\ Id^{\text{sig}}\ Params\ \text{“\{”}\ \{Act\}^*\ \text{“\}”};$	$\mathbf{Id}^*_{\text{src}} \times \mathbf{Decl}^* \times \mathbf{Id}_{\text{sig}} \times \mathbf{Params} \times \mathbf{Act}^* = \mathbf{When}$
$Func ::= Id\ Params^{\text{in}}\ \text{“}\rightarrow\text{”}\ Params^{\text{out}}\ [\text{“\{”}\ \{Act\}^*\ \text{“\}”}]^{\text{hasBody}};$	$\mathbf{Id} \times \mathbf{Params}_{\text{in}} \times \mathbf{Params}_{\text{out}} \times \mathbf{Bool}_{\text{hasBody}} \times \mathbf{Act}^* = \mathbf{Func}$
$Act ::= [\text{“}\star\text{”}]^{\text{isIter}}\ Expr\ [\text{“?”}\ \text{“\{”}\ \{Act^{\text{then}}\}^*\ \text{“\}”}\ [\text{“\{”}\ \{Act^{\text{else}}\}^*\ \text{“\}”}]];$	$\mathbf{Bool}_{\text{isIter}} \times \mathbf{Expr} \times \mathbf{Bool}_{\text{isCond}} \times \mathbf{Act}^*_{\text{then}} \times \mathbf{Act}^*_{\text{else}} = \mathbf{Act}$
$Expr ::= (Const \mid Ref)\ \{\ \text{“.”}\ (Ref \mid Call)\}^*\ [\text{“}\rightarrow\text{”}\ (Decl \mid Params)];$	$(\mathbf{Const} + \mathbf{Ref}) \times (\mathbf{Ref} + \mathbf{Call})^* \times (\mathbf{Decl} + \mathbf{Params}) = \mathbf{Expr}$
$Ref ::= Id\ [\text{“[”}\ (Expr \mid \text{“}\star\text{”})\ \text{“]”}]^{\text{hasIdx}};$	$\mathbf{Id} \times \mathbf{Bool}_{\text{hasIdx}} \times (\mathbf{Expr} + \mathbf{Unit}) = \mathbf{Ref}$
$Call ::= Id\ \text{“(”}\ [\{Expr \mid \text{“}\star\text{”}\}^{\text{“,”}}]^{\text{args}}\ \text{“)”};$	$\mathbf{Id} \times (\mathbf{Expr} + \mathbf{Unit})^*_{\text{args}} = \mathbf{Call}$
$Decl ::= [\text{“[ ]”}]^{\text{isArray}}\ [Id^{\text{alias}}\ \text{“:”}]\ \{Id\}^{\text{“.”}}\ [\text{“=”}\ Const]\ [Str^{\text{imp}}];$	$\mathbf{Bool}_{\text{isArray}} \times \mathbf{Id}_{\text{alias}} \times \mathbf{Id}^* \times (\mathbf{Const} + \mathbf{Unit}) \times \mathbf{Str}_{\text{imp}} = \mathbf{Decl}$
$Params ::= \text{“(”}\ [\{Decl\}^{\text{“,”}}\ [\ldots]^{\text{isVariadic}}]\ \text{“)”};$	$\mathbf{Decl}^* \times \mathbf{Bool}_{\text{isVariadic}} = \mathbf{Params}$

Figure 3. Concrete grammar (left) and abstract representation (right)

Listing 1. logger.too

```
logger {//MEMBERS
  []data:string
  []temperature:sensor
  sheet "Alice/sheet"
  statistics normal
  timer = "1s"
  time

  notifies(now:tick, μ:number, σ:number)//EVENTS

  timer notifies·expiry() {//RULES
    timer reset("60s")
    normal clear
    normal append(temperature[*])
    normal n is·not·equal·to(0)? {
      normal mean → μ
      normal variance square·root → σ
      logger work(time now, μ, σ)
    }
  }

  work(tick, μ:number, σ:number) → () {//FUNCTIONS
    data["A"] assign(tick)
    data["B"] assign(μ)
    data["C"] assign(σ)
    sheet append("Normal!A:A", data, "")
    logger notifies(tick, μ, σ)
  }
}
```

in Too projected to a textual view is shown in List. 1. The program can also be projected to a compact (Fig. 4a), expanded (Fig. 4b), or conventional (Fig. 4c) visual view. The logger thing reads sensor values once per minute, calculates their normal distribution, stores it in a Google spreadsheet, and generates an event to signal other things.

4.1 Thing

Too defines four types of things: *major thing*, *minor thing*, *temporary thing*, and *abstract thing*.

4.1.1 Major thing.

Thing in Fig. 3 defines a major thing, i.e., the program. For example, the program in List. 1 is a *Thing*, where logger is the *Id* of the program.

A thing can be instantiated either explicitly via the "*create instance*" option (e.g., bind a Google sheet thing to a Google sheet file ID), or implicitly at runtime when a thing is first accessed, e.g., whenever the program refers to an array element that does not exist, the element is automatically created.

4.1.2 Minor thing.

A *Decl* in Fig. 3 inside a *Thing* defines a minor thing. A thing may have zero or more minors. Collectively, all the minors define a dictionary, which is effectively the structure of the thing.

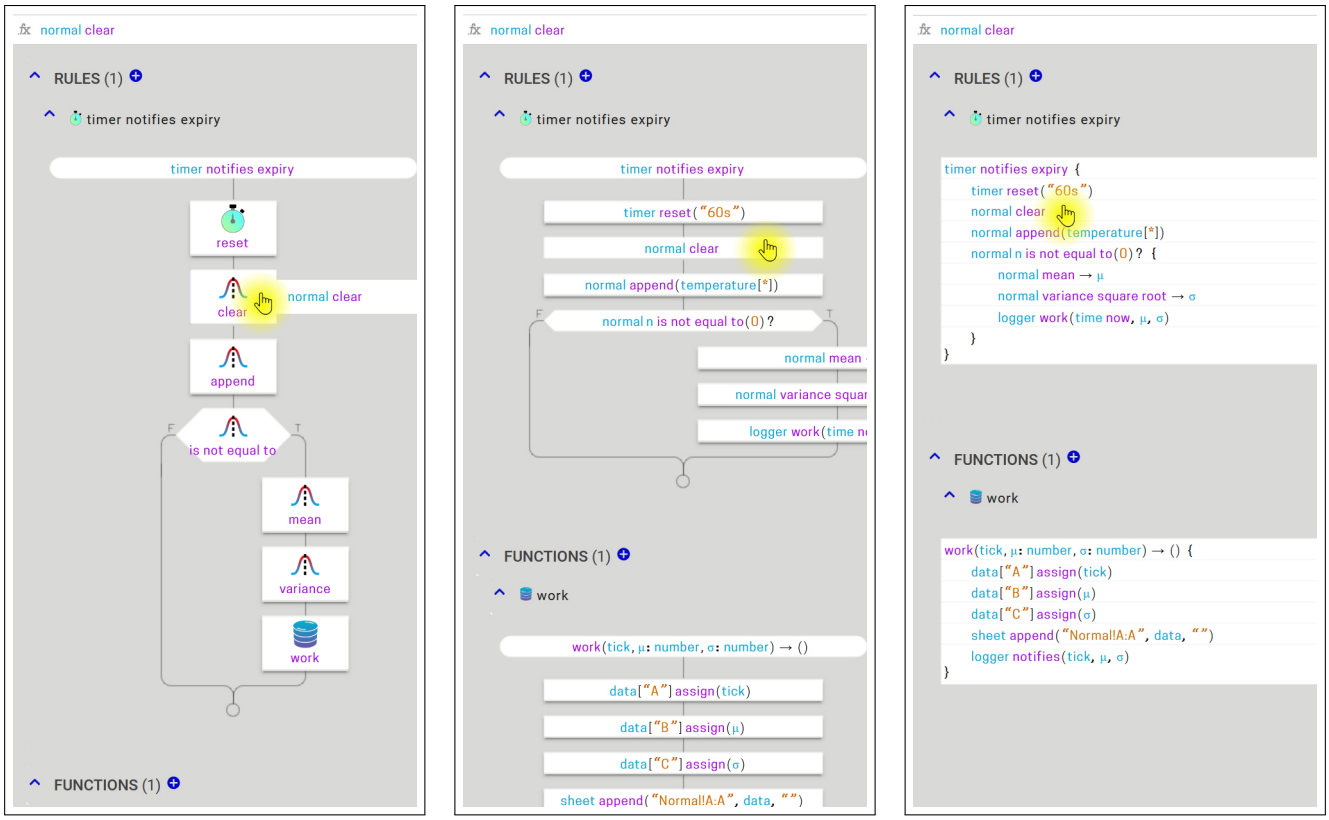
Each minor could be a scalar or an array. A scalar is an instance of a thing, and an array may contain zero or more scalars of the same thing. This is somewhat similar to the JSON concept in which an object is a set of other objects and arrays. The relation between Too and JSON goes further, and a JSON string is used to initialize the thing by default (by unmarshaling the string into the appropriate members).

An array is implemented by a map with named indices, rather than a continuous chunk of memory (with running indices). The keys (named indices) provide extra information, orthogonal to the values. This makes the map a better choice for citizens, as the key-value pairs could be used, for example, to print the names (keys) alongside with the phone numbers (values). A developer who wishes to use a standard array can do so by referring to a library thing such as array·string or array·number.

A *Decl* may contain an array indicator (square brackets prefix), an alias name (Sect. 4.2), an identifier path with one or more things, an initialization value (the = symbol), and an explicit import path (Sect. 4.3).

In List. 1, minor []temperature is an array of temperature sensors, data is an alias for string and also an array, minor timer = "1s" is a timer initialized to expire after one second, and minor sheet "Alice/sheet" is a Google-sheet thing with a specific import path referring to directory *Alice/sheet*.

The last thing on the identifier path is the effective thing that determines the type of the declaration. For example, a statistics thing may include two minors: normal and poisson.



(a) Compact **(b)** Expanded **(c)** Conventional

Figure 4. Projection to a visual view

Another thing may then declare the minor statistics normal, where normal is the effective thing. When referring to normal in the code, there is no need to include the statistics prefix, unless there is another effective thing also named normal.

4.1.3 Temporary thing.

A temporary thing holds the parameter of a function or a rule, or the content of an expression byproduct when redirected (Sect. 4.8). This corresponds to an automatic variable in other GPLs. Unlike minors that are persistent, temporary things disappear when the function, rule, or block ends.

4.1.4 Abstract thing.

An abstract thing is an interface, it may only contain function declarations (without a body), and it cannot contain minor things, events, or rules (Sect. 4.4). When another thing provides an implementation for all the functions in the interface, it is said to implement the interface. A function that accepts an interface as a parameter, may accept any thing that implements the interface.

Scoping is lexical; temporary things have precedence over minor things, and temporary things declared in a block have precedence over temporary things declared in outer blocks.

4.2 Aliases

An alias creates an alternative name for a thing. For example, archimedes:pi declares archimedes to be an alias name for pi. Alias names must be distinctive (unique) within a block and they are transitive; if *a* aliases *b* that aliases *c*, then *a* aliases *c*. Thing *c* in this case is regarded as the basis of *a* and *b*.

Unlike many other GPLs that enforce aliases for every declaration (e.g., *float x*, or *let x:number*), Too only requires aliases to resolve ambiguities. This means that if a function receives a gmail as a parameter, there is no strict requirement to alias it (e.g., g:gmail).

4.3 Imports

By default, things are looked up in the directory tree using a proximity metric, unless an import path is provided with a specific directory. The directory tree may contain multiple variants of the same thing in different directories. In such cases, the program in focus locates the correct variant based on directory distances.

It is important to note that this lookup scheme based on proximity enables us to determine dependencies automatically, unlike in many other GPL where dependencies are explicit (using the `import` or the `include` pragma).

4.4 Signals and Rules

Signals are outbound event messages; rules are inbound event processors.

4.4.1 Signals (outbound events). *Event* in Fig. 3 defines an outbound event. A thing may declare zero or more events that specify how the major thing may signal other things. It defines the name of the signal and the parameters conveyed in the signal message. In List. 1, notifies is an *Event*.

Params in Fig. 3 defines a list of declarations, optionally indicating the last *Decl* as variadic (the ellipsis symbol). For example, in List. 1, the notifies event has the following list of parameters: (now:tick, μ:number, σ:number).

4.4.2 Rules (inbound events). *When* in Fig. 3 defines a rule. A thing may have zero or more rules. A rule comprises a trigger and a block of actions. The rule could be triggered by a self generated signal or by a signal generated by another thing. When triggered, the rule executes the actions sequentially.

The trigger is composed of a *source* and a *signal*. The *source* could be a list of one or more things that are contained in one another. For example, the source of the following rule: thermostat temperature notifies·low is a temperature thing contained in a thermostat thing.

4.5 Actions

An *Act* in Fig. 3 could be *simple* or *conditional*, and executed either once or iteratively, depending on whether or not it is preceded by a '*' prefix.

4.5.1 Simple action. *Expr* in Fig. 3 defines a simple action. It may start with a constant (*String* or *Number*), followed by zero or more *Call* and *Ref*, and optionally end with a redirect operator that assigns the by-product(s) of the last stage into one or more temporary things. In List. 1, normal variance square·root → σ assigns the resulting standard deviation to the temporary sigma.

Ref in Fig. 3 defines a caller thing. In List. 1, the line normal n is·not·equal·to(0) contains a chain of two references: normal followed by n. In case the reference is an array, a subscript may be included. For example, temperature[*] and data["A"].

4.5.2 Conditional action. A conditional action is a simple action with a byproduct constrained to a boolean, plus a *true* block and optionally also a *false* block that follows.

In List. 1, the rule timer notifies·expiry has four actions; the first three are simple actions, and the fourth is a conditional action that has a *true* block of three more actions.

4.6 Functions

A thing may have zero or more functions. For example, work in List. 1 is a function, where argument now:tick is the first *Decl* in *Params*$^{\text{in}}$. A function may take multiple arguments (variadic parameter) and may return multiple values. A function could be defined ad-hoc in a call and could be set to run independently.

Call in Fig. 3 defines a standard function call, comprising a function name and a comma-separated argument list. In List. 1, the line normal variance square·root contains a chain of two calls: variance followed by square·root.

4.6.1 Built-in functions. Too provides a set of predefined functions. The functions are not reserved and the developer may override them with a different implementation. The built-in functions are: id, set·id, as·string, marshal, unmarshal, location, halt, and resume. An *array* thing contains in addition the following functions: length, exist, not·exist, empty, delete, instances, sort, is·ordered, and inverse·order.

4.7 Iterations

In addition to iterations through recursive function calls, there are four possible types for iterations in Too.

4.7.1 Wildcard iteration over an array. The following example iterates through the elements of an array. The '*' operator gets the meaning of wildcard; that is, applies the action to each element of the array. The program goes over the set elements and adds them into sum.

```
bar {
  []set:number
  sum:number
  bar notifies·up {
    sum add·to·it(set[*])
  }
}
```

4.7.2 Wildcard iteration over key-value pairs. The following example iterates through the elements of the set array using key-value pairs, where the '*' operator alone is the key and 'set[*]' is the value, calling the function add·odd to sum up the odd numbers.

```
baz {
  []set:number
  sum:number
  baz notifies·up {
    baz add·odd(*, set[*])
  }
  add·odd(key:string, value:number) →() {
    set[key] is·odd? {
      sum add·to·it(value)
    }
  }
}
```

Note that the expression may contain the wildcard key '*' and the wildcard value 'set[*]' multiple times. It may even contain various arrays, in which case the first wildcard array

that appears in the expression will be the subject for iteration. For example, the print function

```
terminal print(*, ": year=", year[*], ", age=", age[*])
```

iterates through year elements (since it comes before age), and prints also empty strings for age elements that do not exist (when checking for their existence, they are created automatically).

4.7.3 Iterate a simple action. The repetitive action is created by prefixing an action with an '*' operator. In the following example, the iteration affects only the first terminal print action, which is then executed in an infinite loop. It prints *"life is good!"* infinitely, never reaching *"hell"*.

```
goodlife {
  terminal
  goodlife notifies-up {
    * terminal print("life is good!")
    terminal print("hell")
  }
}
```

4.7.4 Iterate a conditional action. Like in the case of iterating a simple action, creating a repetitive conditional action means that the entire *then* block of actions is executed while the boolean expression is true, and the entire *else* block is executed while the boolean expression is false.

The program in the example below repeatedly checks if the sun is in the sky (by checking the time of the next sunrise and next sunset). When true, it executes the *then* block, printing *"The sun is shining!"*. When false, it executes the *else* block, printing *"The stars are blinking!"*. It will go on like this between day and night forever, never reaching *"hell"*.

```
greatlight {
  sun
  terminal
  greatlight notifies-up {
    * sun is-on? {
      terminal print("The sun is shining!")
    }{
      terminal print("The stars are blinking!")
    }
    terminal print("hell")
  }
}
```

If the *else* block is omitted, the loop becomes similar to a *while-do*, and *"hell"* would be reached at sunset.

4.8 Byproducts and Redirection

An expression may have zero or more byproducts, determined as follows. If the expression ends with a thing then this thing is the byproduct. For example, the byproduct of sensor location latitude is a number, since latitude aliases a number. If the expression ends with a number constant (or string constant), then the byproduct is a number (or a string). If the expression ends with a function call, then the byproducts are the function's return values.

An expression may have no byproducts, for example, an event signal (which never returns a value) or a function that does not return a value (e.g., terminal print). Only expressions with a single byproduct could be extended fluently.

If an expression has one or more byproducts, these byproducts could be redirected to temporary thing(s) using the redirect (→) operator. For example, the byproduct of the following action time now →now is a tick, therefore now aliases tick. Similarly, the byproducts of the following action sensor current-reading → (value, error) are the returned values (number, error), therefore value aliases number.

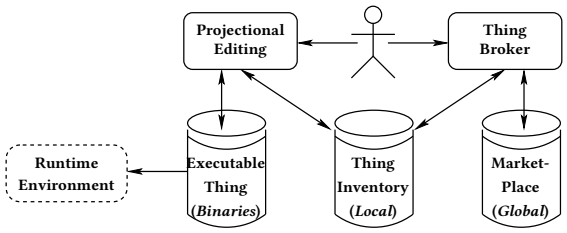


Figure 5. Cloud development environment (CDE)

5 Cloud Development Environment (CDE)

Being simple is a good property for a language but it does not necessarily imply being No-Code. In order to be No-Code the CDE must be designed specifically to enable citizens development by mouse actions alone.

Fig. 5 depicts the user interaction with the CDE. The developer is presented with two distinct applications: *projectional editing* (Sect. 5.1) and *thing broker* (Sect. 5.2). The projectional editing app lets the developer focus on a specific thing; on its data structures and algorithms. The thing broker app lets the developer focus on reviewing things posted by other developers and control the way things sourced by the domain are represented to other developers.

When the developer commands to run the thing in focus, it compiles the Abstract Syntax Tree (AST) together with domain-specific data taken from the binding tree (Sect. 6.2), stores the binary in the *binaries* DB, and runs the program. The program then joins the domain's "orchestra" in receiving and sending messages.

5.1 Projectional Editing

The projectional editing app (Fig. 5) lets the developer manipulate the AST of the thing in focus. The CDE constantly stores the changes in the *local* DB.

5.1.1 Challenges. Current CDEs that are AST aware provide good assistants at the expression level. They list all the functions and objects that are possible at a certain point. However, when the editing of the expression is completed,

Listing 2. carHop.too

```
car·hop {
  []camera
  []last·time:tick
  []last·camera:string
  time

  notifies(plate:string,a:string,b:string,Δ:duration)

  camera[b] notifies·vehicle·identified(plate:string) {
    last·camera[plate] is·not·empty? {
      last·camera[plate] → a
      last·time[plate] → start
      time now difference·from(start) → Δ
      car·hop notifies(plate,a,b,Δ)
    }
    last·time[plate] assign(time now)
    last·camera[plate] assign(b)
  }
}
```

they neither project the next development action, nor do they have any pretension to do so. The developer may choose to end the block, break the loop, return from the function, start a new expression, start a new *if* statement, start a new *for* loop, etc. It seems therefore that listing all these options and waiting for the programmer to make a decision will not promote productivity. On the contrary, the developer would surely outperform such a profusion of suggestions with "hands on the keyboard."

Another challenge with current CDEs relates to the monolithic nature of source files that intermix data structures with algorithms. Current CDEs do not overhaul this, but instead cooperate with this approach. However, this might be too perplexing for citizens that would likely find development easier when the problem is decomposed into simpler problems.

In contrast, the Too CDE decomposes the problem into smaller chunks, using different visual editing tools. In Too the entire development can be accomplished solely by mouse actions alone, using drag-and-drop for developing the *data structure* (Sect. 5.1.2), drag-and-drop for the *flow-control* (Sect. 5.1.3), and drop-down menus for the *actions* (Sect. 5.1.4). These tools are described next in relation to the car·hop program in List. 2. Consider a city with lots of traffic cameras that can identify license plates. The car·hop program notifies of camera *b* that detected the car, together with camera *a* that previously detected the car, and the delta time it took to hop from *a* to *b*. This program could be used later by other programs such as car·speed that notifies of the car speed, car·ticket that notifies of cars crossing the speed limit, and many other programs.

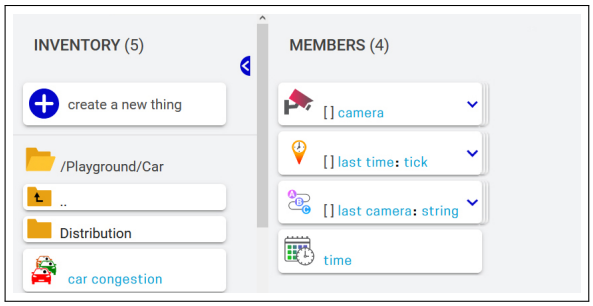


Figure 6. Developing the data structure

5.1.2 Developing the data structure. Since the structure of a thing is simply a dictionary, the CDE enables dragging and dropping things from the inventory into a list of minors, labeled MEMBERS. It then enables removing things from the MEMBERS list, changing their order, changing attributes, such as switching to/from array/scalar, setting an alias name, setting an initial value, and assigning an instance.

Fig. 6 shows a portion of the CDE. The inventory drawer is shown on the left side. This drawer is controlled by the half-circled blue button (◖) and it is normally closed. The car·hop members are shown on the right-hand side of this figure. The cards of the first three members illustrate an array. The small caret symbol is used to expand and contract the stack, listing the array members. An opened array would show the named instances.

If the developer drops a thing and the MEMBERS list already contains a thing by this name, then the CDE automatically assigns a random alias name for the new member, to keep the dictionary sound. Later on, the developer may assign a different alias to the new member.

5.1.3 Developing the control flow. Another portion of the CDE specifies the RULES, FUNCTIONS, and EVENTS. Fig. 7a shows the rule camera notifies·vehicle·identified with its flowchart, built from a trigger in the oval box, and a collection of action boxes connected by edges; simple actions in rectangular boxes, and conditional actions in diamond boxes.

When hovering over a rule or a function, the editing toolbar pops-up, with two small icons outlined in blue: rectangle and diamond. The developer may drag and drop these icons on a flowchart edge to insert an empty action box into the flowchart. Note that the oval box is opened automatically when creating a new rule or function (using the blue plus icon (⊕)).

5.1.4 Developing an action. Fluent-style chaining simplifies editing greatly since it shifts the focus to a single point, namely, the end of the expression. The byproduct determines the alternatives, such as the functions, events, and things that could extend the expression at that point.

The editor may prune some alternatives if they lead to a dead-end. For example, in case of a conditional action,

the editor will not list the terminal since it has no path to a member that results with a boolean.

In addition to the alternatives induced by the byproduct, the syntax dictates a fairly limited number of control alternatives:

- a *delete* alternative that acts as a "backspace" button, erasing the last link in the fluent-style chain,
- a *subscript* alternative ([ ]), only shown if the byproduct is an array,
- a *wildcard* alternative (*), only shown if the expression is empty (new expression) and it is inside a subscript or if it is an argument of a call/signal,
- an *iterative* alternative (*) placed at the beginning of the action, only shown if the expression is the action's main expression,
- a *redirect* alternative (→), only shown if the expression is main and it has one or more byproducts,
- a *string* alternative for typing a constant string, only shown if the expression is empty, and
- a *number* alternative for typing a number constant, only shown if the expression is empty.

This is in contrast to standard GPLs that may contain many more alternatives: for the relevant operators (unary, binary and ternary), and for the relevant opening or closing precedence parentheses. Though, most surely standard GPLs will not have the *wildcard* and *iterative* alternatives.

When the action box is selected (enters edit-mode), the CDE places an *extend* symbol such as a plus or a pencil at the end of every expression. These symbols mark the points where the expressions could grow. Fig. 7b shows two plus symbols at the end of the main expression and at the end of the expression plate located inside the subscript. Then, when the developer selects an *extend* symbol, a drop-down menu opens, listing all the possible alternatives that are relevant at the end of the expression. The list contains all the things (shown in cyan) and all the functions/event signals (shown in magenta) that could be chained at that point, plus a small number of control alternatives at the beginning (shown in gray). In Fig. 7b, for example, only two control alternatives are relevant: *delete* and *iterative*.

The projectional editor colors the *extend* symbols to indicate expression validity. Red indicates that the expression currently does not satisfy a constraint, and blue is used in all other cases. The red symbols remains persistent after the action box is no longer in edit mode. This is to clearly indicate of a problem that prevents the program from running.

5.1.5 Textual view. In addition to the graphical editing by mouse only, the CDE enables conventional typing using a keyboard. This could be used as a gateway for professional developers that require hands on the keyboard. The typing is not entirely free since the structured editor provides smart word completion and keeps the expression structurally sound. While typing, the CDE shows the relevant alternatives in a drop-down menu, like in the graphical editing case, disallowing keys that are not in the initials of the alternatives. The typing is done in the action box or in a special 1-line bar referred to as the *expression bar* (similar to the *formula bar* of a spreadsheet).

The alternatives listed in the drop-down menu could be ordered alphabetically; by their natural order in the things; by frequency of use; or by a smart algorithm that predicts the probability for selecting every alternative. Such an algorithm may use AI capabilities that are built from analyzing many examples in many domains of problems.

In order to enable collaboration, the CDE support *cloud programming*, adopting a concept used in Google docs (or Google sheets), where the cursor (or cell frame) is colored to identify the collaborator. Similarly, when an action is selected by another developer, the action frame changes to a color different than blue, and when hovering, the name of the user is shown (as a tool-tip).

5.1.6 Projecting an action. The projectional editor maintains the AST, and this is used for code coloring; for opening Camel Case (e.g., `weAre` projected as `we are`); for cleaning unnecessary parentheses; for cleaning unnecessary connective symbols such as dots; and for showing the thing's icon when hovering. These measures create a human friendly representation with an improved readability.

For those that prefer the binary infix notation, the projectional editor may convert upon request the fluent style notation to the corresponding binary infix notation. For example,

```
a plus(b) times(c)
```

would be replaced by:

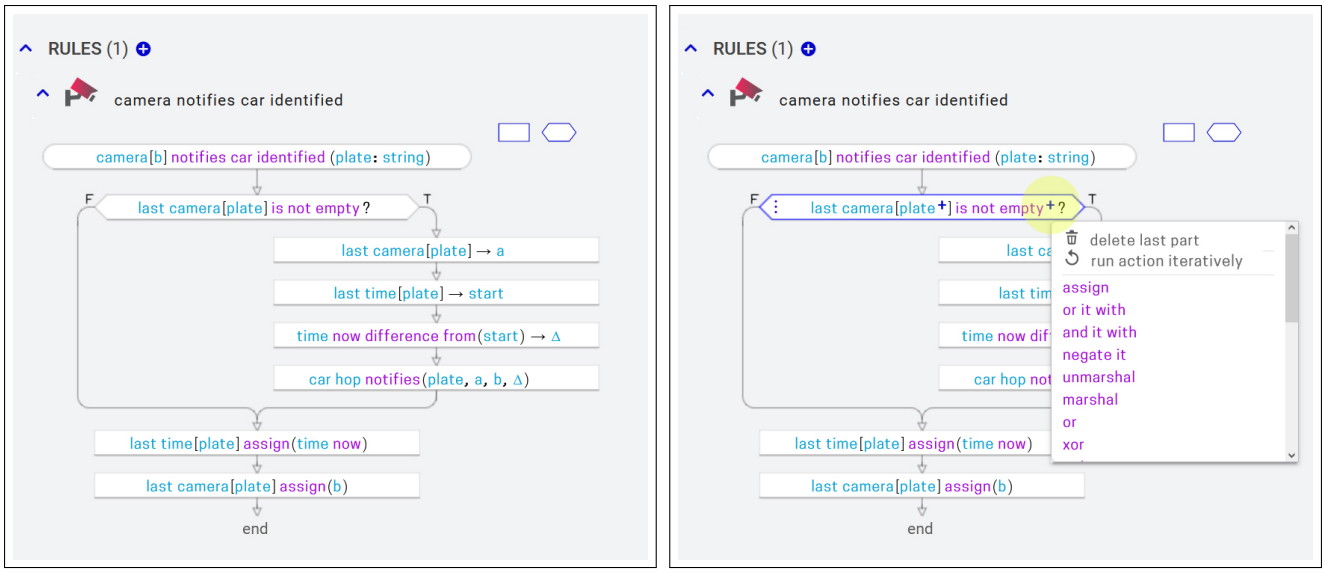
```
(a + b) * c
```

5.2 Thing Broker

The thing broker app (Fig. 5) interacts with the developer and with the two related DBs: the *local* inventory and the *global* marketplace. When the developer instructs the thing broker app to "share" the thing in focus, the thing is copied from the inventory to the marketplace. It also allows the developer to purchase things from the marketplace and copy them to the inventory.

The thing broker app acts like a standard e-commerce arena (like Amazon or AliExpress). It maintains the following records for every thing: name, icon, owner domain, description, price,[7] promotion, search keywords, program brief with function and rule and event headers, documentation, hyperlink for more information (optional), dependency bundle, date posted, number of downloads, rating (such as 1-5

[7] After a period of time the thing may become a public domain and available at no cost, like a patent that expired.



(a) Editing a flowchart

(b) Editing an action

Figure 7. Developing the control flow

stars), developer reviews, related products (things), and number of stock items (may be relevant for physical things, for example).

When purchasing a thing, the thing broker app takes care of the billing; it debits the buyer's domain and credits the seller's domain, leaving some commission to the "house" (to the thing broker). If some things in the dependency bundle have a price tag, then the cost of the downloaded thing would be the accumulated cost, and the buyer get the breakdown of the cost. In such case, the thing broker app takes care of distributing the payment, crediting the various domains. Likewise, if the developer decides to return the thing and cancel the purchase, the thing broker app performs the same actions in the reverse order.

Some things might be located outside the cloud (e.g., an external chat-bot, an external web-server, or a physical sensor), in which case they are represented in the marketplace only with their mock alternative. When the developer purchases a physical thing such as a sensor, it is the responsibility of the seller to ship the real thing to the buyer as a standard merchandise. The thing broker app is open for third parties that wish to post things developed in a different environment. For example, an oracle reports of airplanes take-offs and landing, an AI server, a third party database, etc. Again, in that case the third parties only post the mock alternative in the marketplace, and this mock communicates with the real thing. Third parties are responsible for the availability of the real thing (it is not the responsibility of the Too platform).

5.2.1 Upload (sharing). The *local* inventory contains a subset of the *global* marketplace, with all the things that were purchased, plus some local things that were developed in the domain. When the developer posts a thing for sharing, the thing is copied from the inventory to the same address at the marketplace, along with the *dependency bundle* attached to it.

5.2.2 Download (purchasing). When the thing is purchased via the thing broker, the thing is copied from the *global* marketplace to the same address at the *local* inventory, and then the *dependency bundle* is opened and the things in the bundle are copied to the locations dictated by their paths. At this stage it might be that the domain contains already one or more of the imported things (it contains already the purchased thing or one of the things in its dependency bundle).

In such a case the platform analyzes the discrepancy to verify that the copy can go silently. Otherwise, the platform prompts the developer to resolve the conflict and make the decision whether or not to copy. Silent copy could occur when the imported thing is backward-compatible containing "harmless" additional functions, rules, or events. For example, an imported string thing that contains in addition the replace function.

5.2.3 Versioning. The *dependency bundle* is a set of things that are known to compile and run together, along with their path information and specific version numbers. When the developer runs program foo, the CDE automatically saves the things in the *dependency bundle* of foo and assigns them a version number. At that time, thing foo also receives a version number and is saved. Later on the developer may assign to foo and its *dependency bundle* a meaningful label.

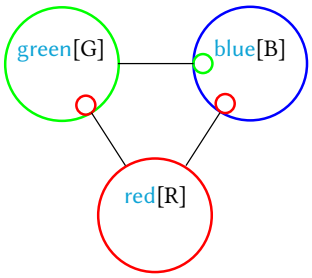


Figure 8. Real vs. mock instances

Note that the CDE also saves things automatically during development. However, these micro-versions are created for undo and redo purposes and do not get version numbers.

5.2.4 Labeling. Labeling is different from versioning since it applies to a group of things rather than a single thing. When the developer assigns a label to a thing, this unique label is attached also to the *dependency bundle*. In this way, at need, the developer may go back to a history label and restore the things in the bundle to recover a sound working version.

6 Language Internals

Too runtime model is based on software components that can be deployed and run independently. Each component runs in a separate process with a dedicated service that can bring it up in case of failure. A component may also run on a different machine, or in a different geographical location (consider a sensor programmed in C++, running inside a Shrimp pool in Guayaquil, Ecuador).

Fig. 8 describes three components (three large circles) that correspond to instances of the following three programs:

```
red { notifies }
green { red }
blue { red green }
```

Instance blue[B] contains instances red[R] and green[G]. However, since instances are unique, blue[B] contains the mocks of these instances (small circles). The mocks provide a thin interface that communicates with the real instance (via RPC).

6.1 Silent vs. Non-silent

A thing in Too can be either silent or non-silent. A thing is defined as *silent* if it does not generate any signal nor receive any signal, recursively. Recursively means that its minors must also be silent. A string and a number are silent and likewise math is silent since it only contains the numbers π and e that are silent (and a collection of functions).

Thing gmail, on the other hand, is non-silent since it generates the notifies·email signal (to notify other things of an incoming email). Likewise red, green, blue are non-silent (since red notifies).

We use this dichotomy to automatically identify the components in the system; components are the non-silent things. The user may choose to activate a non-silent instance (a component) or leave it inactivated, but cannot activate a silent instance. Note that there is no point in activating a silent thing since the result of running it is the same as not running it (no output). For those (rare) cases in which there is a need to reference a silent thing (e.g., boolean) from other things, the library contains a corresponding non-silent version (e.g., boolean·parameter).

When a component is constructed, its silent members are constructed locally as real instances (there is no need to reference them via RPC). This means that if two components instantiate a similar silent thing (e.g., instance "Bob" of thing string), then each maintains a different copy of this instance, and changing one does not affect the other.

A silent thing consumes fairly limited resources resulting in a small footprint in memory. A non-silent thing consumes much more resources, both in memory and network. It generates periodic notifies·alive signals, and it keeps an open connection to the message server in order to send and receive signals. For example, a boolean would consume a single byte in memory, while a boolean·parameter would consume about 10KB (this is the footprint of a TCP client), and would consume in addition network resources.

6.2 Binding Tree

At construction, when resources are allocated, the instances are named and optionally also initialized. The names and initialization strings come from a *binding* JSON file. The JSON is a tree arranged as an interleaved structure of instances and things, and it is generated automatically. List. 3 is an example for a binding JSON that corresponds to the greatlight program in Sect. 4.7.4.

6.3 Initialization

A special initialize function is called right after the thing's resources are allocated and all the minors are constructed. This function enables both the developer and the user to provide initialization values. The developer controls the degree of user involvement in the initialization. It could be that the entire initialization is left for the user; that the developer provides some initialization that the user can override; or that the developer blocks the user completely and provides initialization values alone.

6.4 Run-Shallow vs. Run-Deep

There are two running modes in Too. Run-deep activates the component and its internal members, recursively. Run-shallow only activates the component, leaving its members intact. The run-shallow mode is useful when the user needs to minimize the system's downtime. This may create synchronization problems in case the minor is not yet running,

Listing 3. Binding JSON file

```
{
  "thingName": "greatlight",
  "instances": [{
    "instanceName": "", "initial": "",
    "things": [{
      "thingName": "sun",
      "instances": [{
        "instanceName": "Boston", "initial": "",
        "things": []
      }]
    },{
      "thingName": "terminal",
      "instances": [{
        "instanceName": "Console", "initial": "",
        "things": []
      }]
    }]
  }]
}
```

or the minor is not updated to support the same functionality. However, the system generates appropriate error events that could be processed further. The run-deep mode is useful when the system has gone through a major revision and needs to restart all relevant components to make sure they are all synchronized.

7 Real-World Deployment

An early version of Too in text-based mode was released in 2020 and used by professional developers. The current version with its CDE is available since 2023, and used by both professional and citizen developers. Its library currently contains about 40 things, including: vector, matrix, random, list, gmail, thread·interface, go, channel, hash, io, os, regexp, pdf, unicode, image, error, voice·caller, etc.

7.1 Experience Building Systems

Too is in its infancy but is already being used experimentally in domains such as Precision Agriculture (PA), End-to-End Factory Automation, and Robotic Process Automation (RPA).

7.1.1 System A – Agriculture cultivation farm (a start-up company). In late summer 2020, a large indoor farming startup deployed Too as a control system. It has been in production ever since. To date, tens of millions of dollars have been invested in the 15 indoor spaces controlled by Too.

The system analyses abnormal conditions in the farm, such as the operation of electricity, cooling/heating units, and humidity/temperature/CO2 in the growing rooms. For example, a generator thing reads the number of times the generator has started (via an rs485 thing), and when this number increases, it sends voice calls (via a voice·caller thing) to the persons in charge to indicate that the operation is now running on the generator and not on the power grid.

The system was developed in two months by an expert (a programmer). Today, the system is controlled by two citizens. The system processes about 100,000 events daily, by about 25 components with about 100 instances (plus about 100 physical sensors).

7.1.2 System B – Cosmetics factory. The system controls various processes in the laboratory, such as filling to the threshold, closing the heater when the temperature reaches a certain level, and controlling the process of emulsion by changing the speed of the mixer as the temperature changes, and alerting when to add the next phase.

The system was developed in 2023 by a citizen under the supervision of an expert (programmer). The system is in daily use, and now two citizens constantly improve it. Once the system establishes confidence, the factory plans to use it to control production processes on the shop floor.

7.1.3 System C – RPA. The system converts leads received through a *Contact Us* form on a website, and sends them using an sms thing to the appropriate sales representative based on the *City* field. Each sales representative has a territory defined by a comma-separated list of cities. The system is triggered by email messages containing the contact form in a JSON object. It decomposes the object into a contact thing and then uses the contains function of the string thing to check if the sales rep's territory contains the city.

A citizen created this program unsupervised after watching three training videos: one showing an example of the contains function of the string thing, a second video demonstrating how to convert a JSON object into a thing, and a third video describing the gmail thing. The system processes about 10 leads a day.

7.2 Lessons Learned

Even with No-Code support, software development is generally challenging for citizens. What seems to be trivial for professional developers may be complicated for citizens. For example, OOP was supposed to be very natural but turned out to be intricate especially the invocation of a method (function in Too). Likewise, lacking education in Boolean algebra, creating expressions with multiple operands (such as "`not a or not b`") was confusing for citizens.

However, with sufficient training and mentoring in the early stages, citizens succeeded in creating simple systems. Moreover, in time they modified and extended these systems into complex systems. Generally, after a kick-start, citizens begun to benefit from developing themselves.

The concepts of components, events, and rules were easier to grasp by citizens. Breaking the programming task into smaller tasks was critical. Citizens had no problems with the *flowchart* style in the RULES and FUNCTIONS sections. Citizens

had some difficulties with the MEMBERS section, finding the appropriate thing in the inventory. They requested better documentation and a search function. Creating actions in the RULES section was the most complicated task for citizens. The main difficulty was with the concept of by-products (fluent style function chaining does not exist in spreadsheets and is new to most).

Cloud programming was found to be very useful. It enabled a mentor and a novice to work together on the same program and solve problems in a tight loop. Having a playground was also very helpful. The playground was a good place for citizens to gain confidence because there is no penalty for mistakes there.[8]

7.3 Citizen Feedback

We have interviewed 10 citizens, 5 of whom were asked to solve a similar problem (create a thing with a few rules). There were four levels of difficulty for solving the problem: (i) modifying parameters; (ii) local changes; (iii) creating a new thing; and (iv) creating a complete project with multiple things.

In general, we saw that with no training, citizens succeed with problems of type (i) mainly because Too programs are small and there was no problem locating the relevant parameter. With minimal training, they coped well with simple problems of type (ii) involving adding and modifying actions and conditional actions. Complex problems of type (ii) involving arrays, wildcard operators, and iterations required additional training or mentoring. Similarly, with minimal training citizens succeeded with problems of type (iii), creating a new thing and new rules within that thing.

As for problems of type (iv), these required experienced citizens. Indeed we saw that less experienced citizens could not generalize the RPA solution to *n* reps, which requires iteration and an additional thing (rep). Instead, the outcome was a long chain of *if* sentences (checking if rep1 contains the city, otherwise if rep2 contains the city, etc.)

7.4 Threats to Validity

Answering the question "*could No-Code be Code?*" requires a user study to proper evaluate whether or not citizens could truly develop themselves No-Code solutions in Too. Without a user study, the No-Code narrative should be taken cautiously. Although citizens are not required to write code, they do perform drag-and-drop operations on code blocks. Hence, they still need to read and understand "Code." However, our experience so far suggests that the ability to build a No-Code solution depends more on the complexity of the problem at hand and less on the skills of the developer.

[8]When programming errors might have led to a substantial impact on the business, mentoring was a must.

It is difficult to separate out what we believe would be intuitive to citizens and what would actually prove to be intuitive to citizen developers. Nevertheless, in our experience citizens could be mentored on concepts in Too that were new to them despite their lack of training in programming.

Finally, we note that Too is in active development and has evolved over time. Thus, the user-experience with Too may have varied as a result of this evolution.

8 Discussion and Related Work

No-Code programming is a special case of Low-Code (visual) programming [13]. Low-Code requires minimal but some programming skills, targeting professionals that lack expertise in a specific domain, whereas No-Code requires technical skills but no programming skills, targeting end-users who are citizens.

Both No-Code and Low-Code are special cases of visual programming languages that use visual elements such as blocks, graphs, and flowcharts to represent code. However, not every graphical language is necessarily No-Code.

Block-based coding languages [21] are visual programming languages that let end users create programs by connecting program elements graphically rather than textually [38]. For example, SCRATCH[9] [28] is a popular visual programming language for children that uses blocks to represent commands. More generally, BLOCKLY[10] [25] is a JAVASCRIPT library for building a customized visual programming editor that uses interlocking blocks to represent elements of the code. With KOGI [37], a tool for deriving BLOCKLY code from a simplified context-free grammar of a given language [36], a block-based coding visual environment can be generated for many languages. However, not every block-based coding visual environment necessarily provides a No-Code programming experience for citizen developers, because the graphical abstractions do not necessarily hide the complexity of the underlying language.

Too distinguishes itself from traditional visual languages in its *intent* and *purpose*. The intention of Low-Code platforms is often to reduce coding effort in order to enable rapid development. In comparison, the intention of Too is to empower citizen developers. Most No-Code and Low-Code platforms are special purpose, targeting a specific domain, such as No-Code AI (e.g., OBVIOUSLY.AI)[11] or Low-Code Machine Learning (e.g., AUTOML).[12] They enable "domain citizens" to reap the benefits of the domain without deep knowledge of that domain. In comparison, Too is general purpose.

One of the main goals of Too is to make programming accessible for citizens. Although Too provides an ecosystem

[9]https://www.scratchfoundation.org
[10]https://developers.google.com/blockly
[11]https://www.obviously.ai
[12]https://cloud.google.com/automl

to create complete software solutions, it would be beneficial to have interfaces to other languages. This would enable integration with modules developed by professional programmers in different programming languages.

Too currently provides two options for interfacing with other languages and other development platforms. The first is communication; developers may send and receive event signals to and from an external module. This would allow a Too program to invoke certain actions in another external module, and visa versa; it would allow an external module to invoke certain action in a Too program. The API is available over MQTT protocol and in the future also over HTTP.

The second option is a Go language backdoor; developers may write an entire things in Go language, or some specific functions or some specific rules. This is done by uploading the a Go file to the appropriate folder. File naming convention dictates the compiler that a thing (or function or rule) is written in Go and there is no need to compile it. This backdoor is useful for implementing things that interacts with the operating system, for example.

9 Conclusion

The main contribution of this work is a GPL that is designed to be accessible to non-programmers. The design of Too makes it powerful enough to be general purpose, but at the same time simple enough to program, understand, and evolve. Too is intended to be used by citizens (in their native tongue) to help fill the anticipated gap between the need for programmers and their expected dearth.

Designing a programmable, practical, yet citizen-friendly, GPL is a trail-and-error process. It is analogous to taking out controls and gauges from an aircraft, piece by piece, verifying that it is still airworthy and easy to fly. For example, Too is inspired by Go but omits the `go` keyword for goroutines. Instead, Too uses `go run(thread, "data")`, where `thread` has the thread entry function `start`. Too compiles into Go, but it is not a subset of Go.

Ease of understanding refers to how easily a program written in a language can be understood by a human. The easier it is to understand the code, the simpler the language is considered to be. The minimalism of Too helps keep the abstraction gap between the No-Code graphical projection (Fig. 4) and the underlying textual code (List. 1) small and therefore easy to understand. However, minimizing the language was not a goal in itself; rather, the goal was language simplicity accessible to citizens.

Some of the features that make Too appealing to citizens are: projection of the business logic as rules; projection of the control flow as a two-dimensional flowcharts; projection of action blocks in a way that graphically facilitates flexible edits; and limiting data-structures to a single set of scalars and arrays.

Other features of Too are important mainly for the blend: For example, component based architecture is important for the blend, as citizen developers would greatly benefit from sharing and deploying components selectively, that like in the physical world could be manipulated independently (turned on/off, debugged, etc.). Similarly, projectional editing is important for the blend, as citizens would greatly benefit from an intelligent assistant.

Novel features of Too include: the dichotomy between silent and non-silent things that allows Too to automatically classify components; the ability to automatically resolve dependencies, without explicit `import` or `include` declarations; and the mechanism for developing an action that marks a few cursor points for editing at the end of an expression, which is crucial for No-Code programming in Too.

In designing Too we wanted to create a programming language that would be appealing for both novice and advanced citizens, in the spirit of spreadsheets, but we may have ended up with a language of interest to a broader range of users, including professional developers. The language is small and simple by design to enable end-users easy entry in creating simple programs. But it is also expressive enough to enable advanced users to evolve and create high-end solutions that involve concurrency, synchronization, complex algorithms, and complex data-structures.

Too allows only a single thing per program, which makes programs short, concentrating on a single idea. In addition Too promotes a marketplace where developers can easily share their things. These two features encourage contributions, advocating wisdom of the crowd. Too is a cloud programming language meaning that several developers can work in collaboration on the same piece of code. Add to it flowcharts and icons and you get closer to Knuth's utopia [17] — "*Programming is an Art.*"

Acknowledgments. Many people took part in the design and implementation of Too during its five-year development and construction. We would especially like to thank Shahar Zeira for shaping the language in its early stages (making it stateful, introducing the concept that every thing is made of things including basic types, freeing it from operators by replacing them with function calls, and more). We thank Oz Garinkol for his help in shaping the GUI. Finally, we wish to thank Shahar, Oz, and the anonymous reviewers for their valuable comments on this work.

References

[1] Md Abdullah Al Alamin, Sanjay Malakar, Gias Uddin, Sadia Afroz, Tameem Bin Haider, and Anindya Iqbal. 2021. An Empirical Study of Developer Discussions on Low-Code Software Development Challenges. In *Proceedings of the 18th IEEE/ACM International Conference on Mining Software Repositories (MSR 2021)*. IEEE, Madrid, Spain, 46–57. https://doi.org/10.1109/MSR52588.2021.00018

[2] Md Abdullah Al Alamin, Gias Uddin, Sanjay Malakar, Sadia Afroz, Tameem Haider, and Anindya Iqbal. 2022. Developer Discussion Topics

on the Adoption and Barriers of Low Code Software Development Platforms. *Empirical Software Engineering* 28, 4 (Nov. 2022), 1–59. https://doi.org/10.1007/s10664-022-10244-0

[3] Ken Arnold and James Gosling. 1996. *The Java Programming Language*. Addison-Wesley, Reading, MA.

[4] Corrado Böhm. 1954. *Calculatrices Digitales: du Déchiffrage de Formules Logicomathématiques par la Machine Même dans la Conception du Programme*. Ph. D. Dissertation. L'école Polytechnique fédérale, Zürich, Switzerland. https://doi.org/10.3929/ethz-a-000090226

[5] Travis Breaux and Jennifer Moritz. 2021. The 2021 Software Developer Shortage is Coming. *Commun. ACM* 64, 7 (July 2021), 39–41. https://doi.org/10.1145/3440753

[6] Alan A. A. Donovan and Brian W. Kernighan. 2015. *The Go Programming Language*. Addison-Wesley, Reading, MA, USA.

[7] Brandee Easter. 2020. Fully Human, Fully Machine: Rhetorics of Digital Disembodiment in Programming. *Rhetoric Review* 39, 2 (2020), 202–215. https://doi.org/10.1080/07350198.2020.1727096

[8] Tom Everett. 2012. A collection of formal grammars written for ANTLR v4. https://github.com/antlr/grammars-v4

[9] David Flanagan. 2011. *JavaScript: the definitive guide* (6th ed.). O'Reilly Media, Inc., Sebastopol, CA.

[10] Martin Fowler. 2009. Projectional Editing. Martin Fowler's Bliki. `http://martinfowler.com/bliki/ProjectionalEditing.htmlx`.

[11] Christopher Michael Hancock. 2003. *Real-Time Programming and the Big Ideas of Computational Literacy*. Ph. D. Dissertation. Massachusetts Institute of Technology, Boston, MA, USA.

[12] Anders Hejlsberg, Mads Torgersen, Scott Wiltamuth, and Peter Golde. 2010. *The C# Programming Language (Microsoft .NET Development Series)* (4th ed.). Addison-Wesley, Reading, MA. Annotated Edition.

[13] Martin Hirzel. 2022. Low-Code Programming Models. https://doi.org/10.48550/arXiv.2205.02282 arXiv:2205.02282 [cs.PL]

[14] Sebastian Käss, Susanne Strahringer, and Markus Westner. 2022. Drivers and Inhibitors of Low Code Development Platform Adoption. In *Proceedings of the IEEE 24th Conference on Business Informatics (CBI 2022)*. IEEE, Amsterdam, The Netherlands, 196–205. https://doi.org/10.1109/CBI54897.2022.00028

[15] Lennart C.L. Kats, Richard G. Vogelij, Karl Trygve Kalleberg, and Eelco Visser. 2012. Software Development Environments on the Web: A Research Agenda. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Tucson, Arizona, USA) *(Onward! 2012)*. ACM, New York, NY, USA, 99–116. https://doi.org/10.1145/2384592.2384603

[16] Brian W. Kernighan and Dennis M. Ritchie. 1988. *The C Programming Language* (2nd ed.). Prentice-Hall, Englewood Cliffs, NJ.

[17] Donald ("Don") Ervin Knuth. 1974. A.M. Turing Award lecture. https://amturing.acm.org/award_winners/knuth_1013846.cfm

[18] Donald Ervin Knuth and Luis Trabb Pardo. 1976. *The Early Development of Programming Languages*. Technical Report STAN-CS-76-562. Computer Science Department, School of Humanities and Sciences, Stanford University, Stanford, CA, USA.

[19] David H. Lorenz and John Vlissides. 2001. Designing Components versus Objects: A Transformational Approach. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001)*. IEEE Computer Society, Toronto, Canada, 253–262. https://doi.org/10.1109/ICSE.2001.919099

[20] Yajing Luo, Peng Liang, Chong Wang, Mojtaba Shahin, and Jing Zhan. 2021. Characteristics and Challenges of Low-Code Development: The Practitioners' Perspective. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Bari, Italy) *(ESEM '21)*. ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/3475716.3475782

[21] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *ACM Trans. Comput. Educ.* 10, 4 (Nov. 2010), 16:1–16:15. https://doi.org/10.1145/1868358.1868363

[22] Nicholas D. Matsakis and Felix S. Klock, II. 2014. The Rust Language. *Ada Lett.* 34, 3 (Oct. 2014), 103–104. https://doi.org/10.1145/2692956.2663188

[23] Michael Metcalf, John Reid, and Malcolm Cohen. 2018. *Modern Fortran Explained: Incorporating Fortran 2018* (5th ed.). Oxford University Press, Oxford. https://doi.org/10.1093/oso/9780198811893.001.0001

[24] Jeff Meyerson. 2014. The Go Programming Language. *IEEE Software* 31, 5 (2014), 104–104. https://doi.org/10.1109/MS.2014.127

[25] Erik Pasternak, Rachel Fenichel, and Andrew N. Marshall. 2017. Tips for Creating a Block Language with Blockly. In *Blocks and Beyond Workshop (B&B 2017)*, Franklyn Turbak, Jeff Gray, Caitlin Kelleher, and Mark Sherman (Eds.). IEEE, Raleigh, NC, USA, 21–24. https://doi.org/10.1109/BLOCKS.2017.8120404 Position statement.

[26] Rob Pike. 2015. Simplicity is Complicated. Invited Talk at the European Go Conference (dotGo). Paris, France.

[27] Daniel Pinho, Ademar Aguiar, and Vasco Amaral. 2023. What about the usability in low-code platforms? A systematic literature review. *Journal of Computer Languages* 74 (2023), 101–185. https://doi.org/10.1016/j.cola.2022.101185

[28] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (Nov. 2009), 60–67. https://doi.org/10.1145/1592761.1592779

[29] Benjamin Roussey. 2017. Roll Your Own: What The Citizen Developer Wave Means For Your Enterprise IT Security. TechGenix. https://techgenix.com/citizen-developer-enterprise-it-security

[30] Apurvanand Sahay, Arsene Indamutsa, Davide Di Ruscio, and Alfonso Pierantonio. 2020. Supporting the Understanding and Comparison of Low-code Development Platforms. In *Proceedings of the 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2020)*. IEEE, Portoroz, Slovenia, 171–178. https://doi.org/10.1109/SEAA51224.2020.00036

[31] Bjarne Stroustrup. 1994. *The Design and Evolution of C++*. Addison-Wesley, Reading, MA.

[32] Fahim Sufi. 2023. Algorithms in Low-Code-No-Code for Research Applications: A Practical Review. *Algorithms* 16, 2 (2023), 108. https://doi.org/10.3390/a16020108

[33] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. 2002. *Component Software: Beyond Object-Oriented Programming* (2nd edition ed.). Addison-Wesley, Reading, MA, USA.

[34] David Thomas and Andrew Hunt. 2000. *Programming Ruby: the pragmatic programmer's guide*. Addison-Wesley, Reading, MA.

[35] Guido van Rossum. 1994. *Python Reference Manual*. Technical Report Release 1.0.2. Centrum Wiskunde & Informatica (CWI), Amsterdam, The Netherlands.

[36] Mauricio Verano Merino, Tom Beckmann, Tijs van der Storm, Robert Hirschfeld, and Jurgen J. Vinju. 2021. Getting Grammars into Shape for Block-Based Editors. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering* (Chicago, IL, USA) *(SLE 2021)*. ACM, New York, NY, USA, 83–98. https://doi.org/10.1145/3486608.3486908

[37] Mauricio Verano Merino and Tijs van der Storm. 2020. Block-Based Syntax from Context-Free Grammars. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering* (Virtual, USA) *(SLE 2020)*. ACM, New York, NY, USA, 283–295. https://doi.org/10.1145/3426425.3426948

[38] David Weintrop and Uri Wilensky. 2017. How Block-based Languages Support Novices. *Journal of Visual Languages and Sentient Systems* 3 (July 2017), 92–100.

Received 2023-04-28; accepted 2023-08-11