Informatics in Education, 2024, Vol. 23, No. 4, 823–836

DOI: https://doi.org/10.15388/infedu.2024.29

Principles of Educational Programming Language Design

Michael KÖLLING
Informatics Department, King's College London, UK
e-mail: michael.kolling@kcl.ac.uk

Abstract. The principles of programming language design for learning and teaching have been described and discussed for several decades. Most influential was the work of Niklaus Wirth, describing principles such as simplicity, modularity, orthogonality, and readability. So why is this still an area of fundamental disagreement among educators? Why can teachers still not agree on suitable languages for novice programming? Why do we not have a programming language that is designed for education and in widespread use across the world? This paper enumerates and describes educational language design principles in the context of current systems and technologies and discusses why interpretation of these principles shifts as our discipline progresses. We evaluate what these principles mean in our current world, and why a common agreement has not developed. We discuss the relative benefits of pedagogical languages vs. industry languages and articulate why every generation of learners needs their own language.

Key words: programming education, language design, programming pedagogy.

1. Introduction

The principles which should govern the design of programming languages used in the initial learning of programming have been discussed for almost as long as programming languages have existed. With the design and development of Basic (Kemeny and Kurtz, 1964) in 1964 and Pascal (Wirth, 1971) in 1970, these discussions became mainstream: almost every programmer had an opinion about what made or did not make a "nice" programming language, what a good language should "look like," and which language they would use to teach others.

Quite quickly, discussions included principles of programming language design: While Basic had as its main goal to make programming easier and more accessible to less technically trained users, very quickly arguments emerged making explicit pedagogical points. Logo (Solomon *et al.*, 2020) in 1967 and Pascal in 1970 were designed explicitly as languages for learning. Their creators – Feurzeig, Papert, and Solomon for Logo, and Wirth for Pascal – merged considerations of pedagogy and programming language design in their work, seeking synergies of simplicity that would serve both ease of learning and ease of implementation. Niklaus Wirth especially made repeated and strong points arguing that insights gained from educational considerations could benefit programming language design in general (and vice versa), and that the simplicity and clarity he was striving for

should be a guiding principle for all language design, serving equally pedagogical and development purposes (Wirth, 1971; Jensen and Wirth, 1974). Programming education as a discipline was born.

The design principles that emerged from these discussions over the next decades are reasonably well understood: simplicity, avoidance of redundancy, readability, and a concise set of constructs form the core of the language goals, while availability (in terms of access and cost), motivation (in the form of kinds of outputs a system creates), and a supportive ecosystem (material, community, etc.) are strong secondary aspects.

So where are we today?

Since the primary principles of educational programming language design are reasonably well understood, and rarely disputed in principle – work on educational systems in the last decades is largely concerned with interpreting and instantiating these principles in concrete systems rather than debating the principles themselves – one might expect a situation of widespread adoption of well-designed educational programming languages in a large majority of teaching situations.

Yet the reality is very different.

Teaching experiences in secondary school and upwards (into late school years and early university) are dominated by languages and systems initially designed for professional programmers: Python, Java, JavaScript, and C# are the most commonly used languages, with even C++ – a notoriously tricky language to master – making a regular appearance. Only in primary school education do we find a situation where educational systems dominate: block-based languages are used for initial programming in this age group, and these are pure pedagogical tools. But this may have to do as much with the impossibility to use existing textual languages at this level as with the undisputed merits of these novel systems.

In this paper, we will discuss a number of aspects and influences that contribute to this situation. We present short discussions of various aspects that strongly influence the emergence, success, and adoption of new languages for teaching, relate these to the fundamental principles introduced by Wirth in the formulation of his languages, and we extend the discussion of design principles to include a number of higher-level and external aspects that have strong relevance. We explore how these aspects might influence the design of future educational languages and arrive at some recommendations for the design of future systems.

2. Pedagogical Versus Industry Languages

The selection of a language to use in a teaching context is driven by various considerations. These include pedagogical aspects, such as the design of the language itself, but also external factors, such as the availability and cost of support material and prior experience of the teacher. One of the drivers is the perceived usefulness of the language in a real-world context.

Students (and their parents) often have opinions which language is “better” to learn. In forming these opinions, the definition of “better” can often be vague and driven by limited

insight. One strong aspect commonly cited is the perceived usefulness of a language in the "real world." If a language is widely used in industry, it is more likely to be seen as a useful language to learn.

Pedagogically, this view is spurious. To learn programming well, a student must master more than one language. Only when studying multiple languages does it become clear to learners which aspects of the system they encountered were accidental details, and which illustrate the underlying principles. In transferring experiences and encountering different instantiations of fundamental concepts, a student can form connections and reach an understanding that is not reachable after the encounter of a first language.

While it might be desirable that a student leaving university is fluent in a language in common professional use, the argument above lets us conclude that this does not need to be the first language students encounter. In fact, direct use in industry is a decidedly weak argument for a first language to learn.

This does not mean, however, that this argument is not influential. Use in industry is very often mentioned by lecturers and teachers as a strong motivator for language choice in their teaching. The effects of this are unfortunate.

Pedagogical languages and languages in widespread use by professional programmers have different design goals and different development trajectories over their lifetime. In their initial design, pedagogical languages tend to be smaller, involve less redundancy, and are often able to present simpler, cleaner design because some advanced concepts can be ignored. The differences become even more pronounced as languages age: Both types of languages often start out with a coherent, consistent design and a relatively well-selected set of constructs, but pressures on professional languages are different from those on pedagogical languages. Often, these languages are modified and extended in ways that are detrimental to teaching, for reasons that do not exist in purely educational contexts. (This is further discussed below; see Sections 10 and 11.)

The *use-in-industry argument* is one of the strongest drivers of language choice in schools and universities, and the programming education community should investigate ways and means how this argument can be effectively countered in the ongoing debate.

3. Don't Invent

Successful languages come in two flavours: innovation languages and consolidation languages. Many languages are designed and implemented, and then fail to become successful. The reason is often that they try to be both.

Innovation languages are those that introduce new ideas, new concepts, and new approaches. They advance language design as an academic field and explore new possibilities. These languages are rarely very widely adopted. Simula (Dahl and Nygaard, 1966) is an example. Simula introduced concepts of object orientation, one of the most popular programming paradigms today, but never became widely adopted. Despite its relative lack of users, it is one of the most influential programming languages in history.

Consolidation languages take ideas that were previously introduced in other languages and package them into a consistent, coherent whole. They advance programming languages

in a different way: they change programming practice by creating new languages out of tried and tested elements, and provide a system that can be widely adopted. Java (Arnold and Gosling, 1996) is an example: it brought many concepts to the mainstream, including a garbage-collected virtual machine environment, standardised class libraries, an enforced object model, and exceptions, but none of these were new. They had been implemented in various other languages previously. Here, however, they were merged into a coherent package with a design that learned from prior attempts. Multiple inheritance in Java provides an illustrative example: by separating type and implementation inheritance, its design managed to avoid some of the implementation headaches that plagued earlier attempts, for example in C++ (Stroustrup, 1991).

Success of these two types of languages is measured differently: innovation languages are successful if they introduce concepts that later prove useful in other programming language implementations. The success is academic, their creators are scientists, artists, and engineers combined in a single person.

Consolidation languages, on the other hand, are successful if they are adopted. A large number of programmers writing code in the language, over a large amount of time, is the ultimate metric. They are tools, and the usefulness of the tool is proven by its use. The creators of these languages are engineers through and through.

When designing a new educational programming language, designers should stick to the second type. Select successful, proven concepts and constructs, and simplify, simplify, simplify. Many attempts at popularising new educational languages have ultimately failed because the language creators were tempted to innovate too much. When designing a new language, it is very tempting to be creative, to be novel, to introduce the free creative, artistic element into the process. This temptation should be resisted. More educational languages have failed because they were too ambitiously novel than because of lack of innovation.

4. Keywords Versus Symbols

One prominent aspect of the early debate on educational languages was about the use of keywords versus symbols for syntactic elements of the language constructs. The C-style family of languages (BCPL, C, C++, and later Java and JavaScript) favoured symbols for their conciseness (see Program 1), while the Algol/Pascal family (including Modula-2, Modula-3, Ada, and Eiffel) favoured written-out keywords (see Program 2). Procedures were prefixed with the keyword PROCEDURE, blocks were enclosed in BEGIN/END pairs, IF-statements included a THEN keyword, and so on.

Arguments for using symbols centred around their conciseness: programs were textually shorter, and typing was quicker. Advocates for keyword-based languages emphasised readability: programs were easier to read and understand by human readers, which was seen as a benefit also for learners. Consequently, Wirth designed his languages around keyword-based syntax. Pascal, Modula-2, and Oberon all display this characteristic.

While designers of educational languages typically prioritise readability over conciseness, C-style languages still flourish in actual use in education. C++ and Java saw widespread use for teaching starting from the 1990s until today.

```c
/*
 * Print the output to the terminal depending on val
 */

void printVal(val) {
   if(val > 0) {
      while(val > 0) {
         printf("%d", val--);
      }
   } else if (val == 0) {
      printf("␣");
   } else {
      printf("negative");
   }
}
```

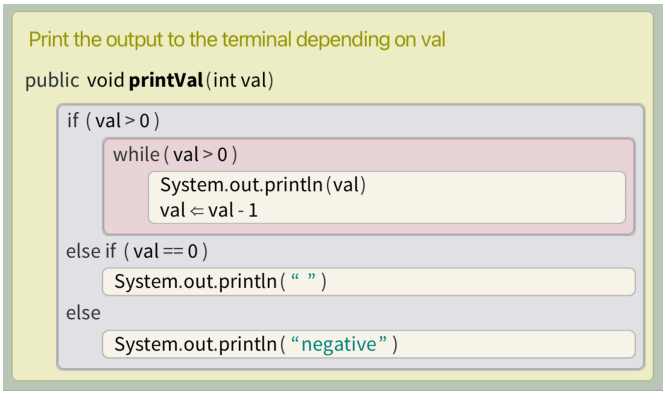
Program 1. Presentation in symbol-based syntax (C)

```
PROCEDURE PrintVal (val: CARDINAL);

   (* Print the output to the terminal depending on val *)

   BEGIN
      IF val > 0 THEN
         WHILE val > 0 DO
            SWholeIO.WriteCard (val);
            val := val - 1;
         END (* WHILE *)
      ELSIF val = 0 THEN
         STextIO.WriteChar ('␣');
      ELSE
         STextIO.WriteString ("negative");
      END (* IF *)
   END PrintVal
```

Program 2. Presentation in keyword-based syntax (Modula-2)

In our time, the argument should be even clearer: tools now exist – sophisticated editors and integrated development environments (IDEs) – that can streamline program input, so that the saving of typing of a few characters can hardly be taken seriously as a decisive argument. Readability, in any objective discussion, should win out over efficiency of typing.

But developments have advanced even further. In modern systems, it is not necessary to rely on characters as the only form of program representation. Block-based and frame-based languages (Kölling *et al.*, 2017) have demonstrated the benefits of replacing some syntactic elements previously represented in text with graphical representations (see Program 3). Instead of arguing whether the extent of a lexical scope should be delineated by parentheses or BEGIN/END keywords, one might choose to use graphical elements – such as frames and colour – instead. This increases readability (especially compared to systems where



Program 3. Presentation in a frame-based editor (Stride)

indentation is under user-control, and thus not guaranteed to be consistent, or when the beginning or end of a scope is out of the current view).

Relying solely on characters for program representation is artificially limiting and lacks justification in a time where all relevant programming platforms support highly sophisticated graphical interfaces. Future educational languages must include these techniques if they want to provide real improvements to the status quo.

5. Separating the Intrinsic from the Accidental

Block-based and frame-based systems provide further lessons beyond the potential to increase readability. Block-based languages, such as Scratch (Maloney *et al.*, 2010) and Snap! (Garcia *et al.*, 2015), have shown that it is possible to create systems that almost entirely eliminate syntax errors. These systems are, however, limited in the size and scope of programs they support to create, and have significant limitations in their flexibility of program manipulation, which make them unsuitable for more experienced and ambitious programmers (Brown *et al.*, 2016). Frame-based systems have shown how some of the advantages of block-based languages can be transferred to more advanced languages (Kölling *et al.*, 2015).

In modern development environments, we should ask ourselves why it is still possible to make errors that the IDE can reliably detect and diagnose. If the development system has all the information about permitted syntax, it makes little sense to let the developer make errors that can easily be prevented by better tools.

At the core of this argument, for educational languages, is the separation of accidental from intrinsic complexity: the challenges intrinsic to software development as a professional discipline should be exposed in an educational system, while accidental complexities should be avoided, ideally by full automation in the development environment. Seeing repeated "Semicolon expected" error messages serves no pedagogical purpose, and modern systems should stop demanding that users expend significant intellectual energy dealing with purely

syntactical issues. Examples include the balancing of parentheses in C-style languages and the manual managing of indentation in Python. Frame-based systems, such as Stride (Kölling *et al.*, 2017) and Strype (Weill-Tessier *et al.*, 2022), have demonstrated how this can be achieved, and future educational systems should not fall back on demanding that users spend effort on tasks that are intellectually meaningless and can easily be automated.

6. Programming for All Versus Programming for Some

Related to the discussion of block-based and frame-based languages is the question of the audience: Who are educational languages for?

For most of the history of the discipline of programming education, the assumption was that the goal is to educate and train programmers. "Programmers" was taken to mean people who were aiming for a career in software development, such as software engineers or other professional developers.

This perspective has changed dramatically.

Programming is now introduced at school age in many countries, often in secondary school and sometimes in primary school (CECE, 2017). The developing consensus among designers of education systems is that every child should have some knowledge of "Informatics," with an experience of programming central to the field (Caspersen *et al.*, 2022).

As a result, the first programming language is encountered by a large section of the population, most of which will not go on to a software development career. Therefore, arguments in language design or language choice that are based on suitability of a language to train professional software engineers have diminished in weight. Future educational languages should be designed with awareness that the majority of the users aim at acquiring a general understanding of programming as a discipline and practice, but will most likely not become software professionals. Thus, languages should concentrate on exposing principles and underlying concepts, and should feel free to ignore detail that is motivated by professional considerations.

7. Piggy-Backing on Existing Infrastructure

Implementing pedagogical programming languages has become much harder than it was in the times of the creation of Logo, Pascal, or Basic. The reason is that expectations of infrastructure and environment have hugely increased.

At the core of this are class libraries: there is now a firm expectation that modern languages come with a significant set of standard libraries, and creating those libraries is essentially impossible for a single developer or a small team.

When Niklaus Wirth designed Pascal, the definitive description of the complete system, the *Pascal User Manual and Report* (Jensen and Wirth, 1974), consisted of just over 40 000 words. And this included the complete language specification, a user manual, and programming examples.

Today, the Java standard class library, for example, consists of over 4 400 classes, defining over 100 000 methods, implemented in millions of lines of code.

While programming systems for primary school age (usually block-based systems) are still relatively small and self-contained, languages chosen in secondary school or later are often expected to support a wide range of functionality, provided in standard libraries.

Implementing the compiler for a language is no longer the main technical challenge; providing the expected extensive environment and libraries is a much larger task. Since educational programming languages are typically designed by academics in small teams with limited resources (while large companies focus on industry-strength languages) this imposes a significant hurdle.

Future educational languages will likely have to find a way to use existing class libraries from established systems (for example, by wrapping a selected set in a syntactic layer for a new language). Implementing sufficient libraries from scratch will be prohibitive for all but very few large, well-funded organisations.

8. Enforcing Good Practice

A principle in the design of Pascal was the idea of enforcing good programming practice. Wirth, in the design of Pascal, aimed at correcting some weaknesses of Basic, the most-used teaching language at the time. In Basic, use of the goto-statement, for example, was common place. Arbitrary transfers of control (including into and out of sub-routines) were possible, and regularly encountered in widely used code.

Wirth sought to improve program structures by eliminating these free-form transfers of control. Flow of control was to be managed by well-defined selection statements, loops, and procedure calls.[1] Similarly, variables had to be explicitly defined and were statically typed, pointers were implicit and could not be arbitrarily manipulated, and Pascal contained various other restrictions in a similar spirit.

For an educational language, this was a very explicit and important choice. Beginners are often not in a position to competently judge which one of a set of alternative constructs provides the best option. Poor programming habits may be formed by accident, and become ingrained over time. A good educational language should nudge users towards good practice, and this includes simply not allowing bad practice where this is feasible.

This fundamental principle is unfortunately often ignored in practical programming education. Java, for example, follows this principle quite strongly: Java is an object-oriented language, and in the object-oriented world, structuring a program in classes, using information hiding and encapsulated data, is seen as good practice. Consequently, Java insists on all code being written in classes and methods, and does not support global data. In common practice, especially at school age, however, programming is often taught using Python.

[1] The banning of goto-statements was not entirely achieved with Pascal. Wirth included the statement in the language – banning it entirely was considered too radical –, but warned in the user manual against its use: "The presence of gotos in a Pascal program is often an indication that the programmer has not yet learned 'to think' in Pascal" (Jensen and Wirth, 1974).

Python too is nominally an object-oriented language, but it enforces no such restrictions. It is much more permissive with the structures it allows.

One of the results of this is that teachers often perceive Python as easier to use than Java. Programs can be written with less syntactic overhead, and students produce their first running program more quickly. In fact, a Python program may consist of a single line, whereas a standard Java program requires several lines of non-trivial boilerplate code.

The perceived simplicity of Python, however, is deceptive. It appears simple not by providing simple constructs for important concepts, but by allowing users to avoid those important concepts (encapsulation, modularisation) altogether. Thus users may fail to appreciate the importance of structure for larger programs and are put in danger of forming bad habits. Java appears more cumbersome, but it is so because it teaches important lessons about program structure, right from the start.

A new teaching language should not give in to the temptation of allowing the writing of shallow, badly structured code for the sake of simplicity in appearance. It should decide on what the important conceptual principles of its paradigm are, and then force or nudge its users into using them. A language designer must be prepared to defend these choices, even in light of the inevitable criticism of those who would like to continue to write quick-and-dirty programs.

9. Programming is Creating Language

In Section 5, we have discussed some of the advantages of block-based languages. These types of language represent the most significant innovation in educational programming languages in the last decades. They combine ease of use with the almost complete avoidance of syntax errors and powerful discoverability. This, combined with the integrated graphical runtime system, has brought programming to a much younger audience than ever before, improving both access and motivation.

Block-based languages are, however, not sufficient to learn about all important aspects and concepts of programming. Most important among their limitations is the difficulty in creating additional blocks.

Creating programmatic solutions to computational problems is often taught as a top-down process. Approaches such as divide-and-conquer and stepwise refinement are used to break large problems into smaller components until their solutions can be implemented in single classes, methods, or functions.

There is, however, a different perspective to program development: a bottom-up approach. The definition of program components (functions, methods, classes) presents the creation of a domain-specific vocabulary, designed to facilitate the effective expression of subject-specific solutions. As such, the program designer is a creator of language, and the act of programming is the extension of the base language provided by the programming system. Modelling of the problem domain is used to understand and express the problem and its solution, and a domain-specific language is created through the implementation of program components, which ultimately allows the elegant expression of the required solution.

Only if both approaches are mastered – top-down and bottom-up – and frequently combined in a single system development process, can a learner become a good programmer. The view of programming as creation of a language is powerful and important, yet not easily represented in traditional block-based systems. As a result, much of school-level teaching of programming focuses on a top-down approach, to the detriment of bottom-up methodologies.

We are aware that more recent block-based systems, in fact, do allow the creation of user-defined blocks, so the criticism here must be moderated. The power to do so is, however, limited in most such systems, and even in systems with comparatively powerful block-creation capability (such as Snap!), use of this feature in actual teaching practice is limited and not typically included as a routine part of early programming education.

For the remainder of this paper, we will be concerned mostly with systems targeted at slightly older age groups, from secondary school upwards into university age. Future educational languages that target these age groups should allow the creation of user-defined abstractions that allow the extension of the base language in ways that blend seamlessly into the syntax of the language.

10. Keeping It Simple

Simplicity has always been a great goal of the design of programming languages, and especially educational languages. It was so central to Niklaus Wirth's languages that a book about his work, written by close collaborators intimately familiar with his thinking, bears the title *The School of Niklaus Wirth: The Art of Simplicity* (Böszörményi *et al.*, 2000).

But simplicity as a goal is not unique to Wirth. When the Java language was designed in the early 1990s, for example, simplicity consistently topped the list of design goals (Sakaiya, 1999). Even today, the team guiding the evolution of Java regularly talks about simplicity as an important goal. Yet, comparing modern Java versions with Pascal reveals huge differences in design decisions.

How can two languages with such a similar goal take such different paths?

One reason is, of course, the passing of time. Requirements of modern languages in the 2020s are vastly different from those of a teaching language in the 1970s. But there is another, more fundamental reason. And the roots of this lie in the meaning of the term "simplicity." While every language designer will readily agree that simplicity is one of the goals of their language design (nobody aims to make their language complex or complicated), and many people debating merits of languages use the term without feeling the need to further define it, there is in fact a clear discrepancy in what different people mean when using this term.

In recent discussions with the Java development team,[2] for instance, its language maintainers quoted simplicity as one of the strong drivers for recent language modifications.

[2]Personal discussion with the author, regarding design of new features in Java, including records, new syntax for switch statements, the introduction of the var keyword and others.

Their view was driven by a perspective of professional programmers: for expert developers, simplicity means having language constructs available that allow them to express an algorithmic solution in elegant terms. A language is simple if, for any given problem, it offers constructs to express its solution in short, concise, and readable terms. Increasing simplicity, in this sense, means offering more language constructs, to increase the expressiveness of the language. It ultimately is the program that is intended to appear simple.

For learners and teachers, on the other hand, simplicity means fewer language constructs. A language is simple if, for any given problem, it offers one (and only one) construct that can be used to address the problem. A simple language avoids redundancy, and its set of constructs is as small as possible.

These two views of simplicity are in fundamental conflict.

The goals of simplicity in educational languages and in languages designed and developed for professional programmers result in very different choices over time.

The development trajectories of most widely used modern languages, including Visual Basic, C++, Java, and Python, illustrate this point. In every case, the languages started out relatively small (compared to their current size and complexity), and constructs were continuously added in the name of improving "simplicity." All of these languages are now much larger than is desirable for a novice teaching language. This development is not a fault of the language maintainers; it is inevitable (see Section 11).

Requirements of simplicity are just fundamentally different in teaching languages versus industry languages. It follows that a good teaching language should be explicitly designed as such, should not be forced to bow to the pressures that come with professional adoption, and should continue, over its lifetime, to be primarily a teaching language.

Our current situation, in any age group after primary school, is very different. The most dominant languages used for teaching are developed and continuously adapted to the requirements of professionals, and using them in teaching introduces unnecessary complexities.

11. Evolution in Established Languages

The most frequently used languages for teaching introductory programming in later school and early university years, at the time of writing, are Java and Python. Both are not primarily designed as teaching languages, but their initial design, even though aimed at professional development, aligned quite closely with design goals one would choose for a teaching language.

James Gosling, the primary designer of the Java language, for example, talked frequently about the goal of simplicity in the language (Gosling, 1997; Arnold and Gosling, 1996), in the sense of a small set of concepts and constructs, the avoidance of redundancy, and the avoidance of special cases. These are aims that fit perfectly with a language for novice learners.

Java, as a result, was widely seen as a clean, concise, and well-designed language, and it was enthusiastically adopted by universities as an introductory teaching language in the late

1990s and early 2000s. In the first decade of this century, it was the most used introductory programming language in universities, and teachers were largely happy with it.

Yet over the nearly 30 years of its life, Java has been consistently extended with new constructs. Examples include generic types, auto-boxing, a new loop construct, lambdas, streams, changes to the syntax of exceptions, records, new syntax for switch statements, default methods in interfaces, sealed classes, and more. This list serves to give a sense of the substance and range of changes made to a popular production language over three decades.

Many teachers who use Java in introductory courses, who are more interested in teaching the principles of programming and computer science than teaching the intricacies of a particular programming language, were unhappy with this development. The size of the language, and redundancy of its constructs, had increased significantly.

Where previously there was one obvious way of solving a task, there are now multiple possibilities, with novices often not well placed to assess trade-offs and make choices. For example, loops over collections were initially arranged with iterators. Later, a for-each loop was added as an alternative. Even later, streams and lambdas were added as yet another alternative. Similar trends are found with various other constructs. Teachers are forced to teach more syntax without necessarily achieving deeper understanding of concepts, and students are forced to make choices that are driven by idiosyncrasies of a particular language, not software development principles.

This development is detrimental to the quality of a language in a novice teaching context. It is tempting to blame language maintainers for continuously "messing with" a perfectly good language, but that view would be naïve. Every language that aims to be seriously used in professional development has to adapt to changes in hardware, operating systems, and programming language research. Some of the changes in Java, for instance, were motivated by improving support for writing code for multi-core architectures, an issue that is important to many professional developers.

In reality, a language in heavy real-world use either consistently adapts, or it dies off to a state of being relegated to the maintenance of legacy projects. For a language to remain attractive for further new development, it must strive to adapt and extend.

In extending, languages become less elegant. Retro-fitting constructs into existing languages is always harder than including them from the start, and frequently requires compromise between new language construct goals and backwards compatibility.

The fact that real-world languages become messier, larger, and more cumbersome (and with this: less suitable as a teaching language) is not a mistake, it is not the fault of their design teams, it is an inevitable force of our discipline. A language grows or dies.

Recognising this, we can clearly see one of the limitations of using industry languages as teaching languages. They may start out well, but they inevitably deteriorate for this purpose.

Using a dedicated teaching language avoids many of these problems. The pressure to adapt is much less strong, and the language can remain stable for a longer time. If it does not change at all, however, it will then be outdated at some point.

What follows from this discussion is that every generation needs their own new language, released at their time of entering the field. Every learner, in an ideal world, starts at a time

when a new small, clean, concise modern language has just been released, and then grows up together with this language.

This ideal requires the design of new pedagogical languages, and the willingness of teachers to adopt them. At the time of writing this paper, it certainly feels that the time is ripe for a new teaching language.

12. Conclusion

The conclusions to be drawn from the points made throughout this paper are reasonably obvious: We need a new language for teaching novices at secondary school and introductory university level.

This language should be designed explicitly for teaching. It should be conservative in its design, selecting and consolidating proven and popular paradigms and constructs, and avoid the temptation of novelty. It should further avoid trying to be everything for everyone. It should emphasise simplicity, conciseness, lack of redundancy, and readability; in doing so, it should move away from restrictions of purely text-based editing and make use of more recent interaction paradigms that incorporate graphical elements and modern code manipulation techniques. This can serve to separate peripheral complexities (such as purely syntactical requirements) from fundamental issues, eliminating the former and emphasising the latter.

The language should make use of existing library and runtime infrastructure to link to an existing ecosystem that would otherwise be hard to maintain for an academic project. It should embody strong views of good practice and not shy away from pushing learners towards those practices. Maintenance and adaptation of this language should be driven by pedagogical considerations, not by industry needs.

Of course, Niklaus Wirth knew all this.

In the design of his own languages, Wirth made practically all of these points. As a community, however, we cannot stay still and rest on our laurels. We have to apply these principles over and over again to keep them relevant. Relying so heavily on industry-oriented languages for our education over the last 30 years has limited our ability to make good pedagogical decisions.

We, as a community, must come back to applying these design principles afresh to create the next great teaching language.

References

Arnold, K., Gosling, J. (1996). *The Java Programming Language*. ACM Press/Addison-Wesley Publishing, New York, NY, USA.

Böszörményi, L., Gutknecht, J., Pomberger, G. (Eds.) (2000). *The School of Niklaus Wirth: The Art of Simplicity*. dpunkt.verlag/Morgan-Kaufmann.

Brown, N., Altadmri, A., Kölling, M. (2016). Frame-Based Editing: Combining the Best of Blocks and Text Programming. In: *Proceedings of the Fourth International Conference on Learning and Teaching in Computing and Engineering (LaTiCE 2016)*, pp. 47–53.

Caspersen, M.E., Diethelm, I., Gal-Ezer, J., McGettrick, A., Nardelli, E., Passey, D., Rovan, B., Webb, M. (2022). Informatics Reference Framework for School, Informatics for All. https://www.informaticsforall.org/the-informatics-reference-framework-for-school-release-february-2022/. Accessed May 2024.

CECE (2017). Informatics Education in Europe: Are We All in the Same Boat? Technical report, Informatics Europe and ACM Europe. Accessed May 2024. https://dl.acm.org/doi/book/10.1145/3106077.

Dahl, O.J., Nygaard, K. (1966). SIMULA: An ALGOL-Based Simulation Language. *Communications of the ACM*, 9(9), 671–678.

Garcia, D., Harvey, B., Barnes, T. (2015). The Beauty and Joy of Computing. *ACM Inroads*, 6(4), 71–79.

Gosling, J. (1997). The Feel of Java. *Computer*, 30(6), 53–57.

Jensen, K., Wirth, N. (1974). *Pascal User Manual and Report*. Springer Verlag, Berlin.

Kemeny, J.G., Kurtz, T.E. (1964). BASIC: A Manual for BASIC, the Elementary Algebraic Language Designed for Use with the Dartmouth Time Sharing System. Technical report, Dartmouth College Computation Center, Hanover, N.H..

Kölling, M., Brown, N., Altadmri, A. (2015). Frame-Based Editing: Easing the Transition from Blocks to Text-Based Programming. In: *Proceedings of the 10th Workshop in Primary and Secondary Computing Education (WiPSCE 2015)*. Association for Computing Machinery, New York, NY, USA, pp. 29–38.

Kölling, M., Brown, N., Altadmri, A. (2017). Frame-based Editing. *Journal of Visual Languages and Sentient Systems*, 3, 40–67. Special Issue on Blocks Programming.

Maloney, J., Resnick, M., Rusk, N., Silverman, B., Eastmond, E. (2010). The Scratch Programming Language and Environment. *ACM Transactions on Computing Education*, 10(4), 16–11615.

Sakaiya, T. (1999). Introduction to Java™ Technology. https://www.oracle.com/java/technologies/introduction-to-java.html. Accessed May 2024.

Solomon, C., Harvey, B., Kahn, K., Lieberman, H., Miller, M., Minsky, M., Papert, A., Silverman, B. (2020). History of Logo. *Proceedings of the ACM on Programming Languages*, 4(HOPL), 1–66.

Stroustrup, B. (1991). *The C++ Programming Language* (2nd ed.). Addison-Wesley.

Weill-Tessier, P., Kyfonidis, C., Brown, N., Kölling, M. (2022). Strype: Bridging from Blocks to Python, with Micro:bit Support. In: *Proceedings of the 27th ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE 2022)*. Association for Computing Machinery, New York, NY, USA.

Wirth, N. (1971). The Programming Language Pascal. *Acta Informatica*, 1(1), 35–63.