International Journal of People-Oriented Programming

January-June 2015, Vol. 4, No. 1

Table of Contents

Special Issue on Kids and Other Novices Learning to Code: Insights, Tools and Lessons from the Visual Programming Frontline

Editorial Preface

Invited Commentary

Research Articles

Editor's Note

The *International Journal of People-Oriented Programming* is indexed or listed in the following: Bacon's Media Directory; Cabell's Directories; DBLP; Google Scholar; INSPEC; JournalTOCs; MediaFinder; ProQuest Advanced Technologies & Aerospace Journals; ProQuest Computer Science Journals; ProQuest Illustrata: Technology; ProQuest SciTech Journals; ProQuest Technology Journals; The Index of Information Systems Journals; The Standard Periodical Directory; Ulrich's Periodicals Directory

Lessons from the Design of Three Educational Programming Environments: Blue, BlueJ and Greenfoot

Michael Kölling, University of Kent, Canterbury, UK

ABSTRACT

Educational programming systems are booming. More systems of this kind have been published in the last few years than ever before, and interest in this area is growing. With the rise of programming as a school subject in ever-younger age groups, the importance of dedicated educational systems for programming education is increasing. In the past, professional environments were often used in programming teaching; with the shift to younger age groups, this is no longer tenable. New educational systems are currently being designed by a diverse group of developing teams, in industry, in academia, and by hobbyists. In this paper, the author describes his experiences with the design of three systems—Blue, BlueJ, and Greenfoot—and extract lessons that he hopes may be useful for designers of future systems. He also discusses current developments, and suggests an area of interest where future work might be profitable for many users: the combination of aspects from block-based and text-based programming. The author briefly presents his work in this area—frame-based editing—and suggest possible future development options.

Keywords: Blue, BlueJ, Design, Educational IDE, Frame-based Editing, Greenfoot

INTRODUCTION

In the last ten years or so, educational programming environments have become very popular for the teaching and learning of introductory programming. This was not always the case: while there have been educational systems for a long time, they were considerably fewer early in this century than today, and older systems were considerably simpler, often consisting of compilers or libraries, rather than complete programming environments. Long and heated debates used to rage among educators about the respective benefits of teaching with dedicated educational versus industry-strength tools. These debates usually remained unresolved.

In the last decade, the situation has shifted, due to a combination of reasons which we discuss below, and educational programming environments have taken a much more prominent role. They are more used, more accepted, and simply many more in number, than ever before. As a result, the design of educational environments has become a topic of considerable interest.

DOI: 10.4018/IJPOP.2015010102

In this paper we describe experiences with the design of a sequence of educational environments dating back more than 20 years. These systems are Blue (Kölling, 1999a), a programming language and development environment for teaching and learning object-oriented programming in a single, integrated system; its successor BlueJ (Kölling, Quig, Patterson, & Rosenberg, 2003), a similar environment using the Java Programming Language; and a third pedagogical system called Greenfoot (Kölling, 2010). Blue was relatively short-lived, but is of interest here because it heavily influenced the design of its successor, BlueJ. BlueJ and Greenfoot are both systems with significant user communities over a number of years (and still very much in use today), and have undergone many changes and adaptations since their first publication.

In this paper we present a short history of these systems and discuss the goals and design rationale for each, their respective target groups and how these influenced design decisions, and their scope and application. Most importantly, we discuss lessons learnt from their use with actual users, and how those lessons shaped the design of the later systems, or later versions. We also discuss their relation to other educational programming systems, similarities, possible sequences of use, and future developments. The emphasis is not on providing a complete description of each system, but to identify the trends and goals at the time of their design, and how these have changed over time. Overall, we present some lessons we learnt along the way that we hope may be of use to designers of future systems.

A SHORT HISTORY OF EDUCATIONAL PROGRAMMING TOOLS

Educational software tools are nearly as old as programming as a discipline. Ever since computer scientists started teaching others about programming, they started thinking about tools to support this challenge. In the early days, there was no difference between the tools used by professionals and the ones taught to newcomers. However, pretty soon systems started to be developed that were designed partly or primarily with beginners as users in mind.

We will not give a complete history of educational software here; instead, we mention just a few influential early systems to arrive quickly at our destination: educational development environments for object-oriented programming. This is where we will slow down and start discussion in more detail.

The first pedagogically oriented software tools were programming languages and their associated compilers. Among the early ones, BASIC (1964), Logo (1967), Pascal (1970), and Smalltalk (1972) stood out as the most used and most influential—all aiming at learners as their primary target group. The goal of these languages was partly *simplification*: taking known concepts and avoiding the complications that could arise in other existing languages at the time. BASIC and Pascal were part of this movement, introducing more rigid structure and creating higher abstraction levels in programming in the process. The other part was the introduction or appropriation of concepts and abstractions that might be more accessible to learners: micro-worlds in the case of Logo (Papert, 1980), and the adaptation of object orientation (a reasonably obscure programming paradigm at the time, introduced a few years earlier in the Simula language (Dahl, Myhrhaug, & Nygaard, 1967)) in the case of Smalltalk.

In parallel, a small number of libraries were being developed for similar purposes, aiming at programming education that was (or later became) language independent. In the 70s and 80s, a few of these dominated the educational space. One of the most successful was *Turtle Graphics*, a library first developed by Seymour Papert and others for the Logo language (Papert, 1980), and later re-implemented for countless other educational languages (Caspersen & Christensen, 2000; Python Software Foundation, 2012; Slack, 1990). Turtle graphics introduced the concept

of a micro-world, together with a single actor, which could produce movement and leave graphical traces as output[1].

In 1981, Pattis expanded on this idea with the widely used *Karel The Robot* system (Pattis, 1981), which again was ported to numerous languages (Bergin, Pattis, Stehlik, & Roberts, 1997; Bergin, Stehlik, Roberts, & Pattis, 2005). With Karel, students programmed a software robot that could move through a grid-based world, collect "beepers", and avoid obstacles. In each of these systems, students could gain experience with fundamental programming concepts and constructs within a carefully controlled and contained problem domain.

In the early 1990s, one of the most relevant developments—even though not aimed primarily at teaching—was the advent and rise of GUI builders. In 1991 Microsoft released Visual Basic, a new system to replace their previous environment, QuickBasic. While QuickBasic was text-based, Visual Basic's main feature was the central role of its integrated GUI builder. Similarly, Borland released Delphi in 1995, a GUI-focused development environment based on Pascal. Professional IDEs with GUI builders were sometimes used in teaching. However, since their professional focus and the dominance of GUI building (usually with automated code generation) did not easily aid the learning of foundational principles, these attempts to start programming teaching with GUI building ultimately led to a widely-held view against IDEs in programing education in general. We will come back to this below.

With the popularization of object orientation in introductory teaching from the late 1990s, the existing educational libraries were adapted to this new paradigm, and new teaching tools started to appear. By the end of the 1990s, however, the choice of educational systems was still fairly limited. Educational software in this time mostly consisted of libraries of this kind, while educational programming languages were being displaced in most teaching institutions by new, industry-strength languages. C++ (Stroustrup, 1986), Visual Basic, and Java (Gosling, 2000) were the languages of choice in introductory courses at this time—all systems developed for professional software engineers. By the turn of the century, full dedicated pedagogical development environments were still very rare[2].

However, this lack of educational environments was about to change.

Within the next 10 years, a substantial number of pedagogical systems were developed and published. Many of these introduced concepts not previously available in educational contexts, such as improved support for interaction and experimentation and the use of rich media.

An early example was Blue (Kölling & Rosenberg, 1996), published in 1995, followed by BlueJ (Kölling, Quig, Patterson, & Rosenberg, 2003) and GameMaker (Overmars, 2004), both published in 1999. This was quickly followed by the publication of Alice 2 in 2000 (Cooper, Dann, & Pausch, 2003), DrJava in 2002 (Allen, Cartwright, & Stoler, 2002), and Jeroo (Sanders & Dorn, 2003) in 2003. 2005 saw the publication of Scratch (Maloney, Resnick, Rusk, Silverman, & Eastmond, 2010), followed by Greenfoot in 2006 (Henriksen & Kölling, 2004), StarLogo TNG in 2007 (Begel & Klopfer, 2007) and Kodu in 2009 (MacLaurin, 2009). After this, development accelerated even more, with numerous systems being published within a small number of years. One aspect that supported this proliferation was the publication of libraries specifically for the development of educational block languages, such as OpenBlocks (Roque, 2007) and Blockly (Fraser, 2013). BYOB/Snap (Harvey & Mönig, 2010, Harvey & Mönig, 2015), Pencil Code (Bau, Bau, Dawson, & Pickens, 2015), Grace (Black, Bruce, Homer, Noble, Ruskin, & Yannow, 2013), and App Inventor (Wolber, Abelson, Spertus, & Looney, 2011) are some of the more recent examples, and more are being developed.

For this paper, we will now concentrate on the development of Blue, BlueJ and Greenfoot, and our experiences with their designs.

BLUE AND BLUEJ: PROGRAM STRUCTURE VISUALIZATION AND INTERACTION

We will start our discussion of the development of the Blue and BlueJ environments with a description of various aspects of their creation, arranged roughly chronological, but selected for their relevance to the ultimate goal: the extraction of a set of generic lessons drawn from our experience, presented at the end of this section. The aspects discussed are those that motivate and explain our design choices and illustrate what we have learned in the process of this work.

The First Goal: An Educational Programming Language

In 1993, we began investigating how object-oriented languages might be taught, and how this teaching might be supported by a programming language. The initial idea was to create something like a "Pascal for object orientation". Pascal was one of the most successful teaching languages ever created, and it had instigated immense progress in structured programming: not only was it used to teach countless beginners, but it had a lasting influence on the design of programming languages, helping to make structured programming the dominant paradigm at the time. With object orientation on the rise, we were looking for a similar language for this paradigm.

Our intended target group was first year university teaching. Object orientation at the time was—if it was taught at all—often seen as an advanced subject taught in higher level courses. However, the idea to start teaching with an object-oriented language in first year was slowly gaining traction among some educators, and we were interested in this approach.

We started by formulating our goals and requirements (Kölling, Koch, & Rosenberg, 1995; Kölling, 1999b), and evaluating existing systems against them. The big contenders at the time were C++, Smalltalk (Goldberg & Robson, 1983) and Eiffel (Meyer, 1988) (with Java following soon after), and a long list of smaller, less popular languages. As Pascal before us, we had goals motivated by pedagogy: we wanted a clear and consistent representation of programming concepts, clear and readable syntax, good error messages, little redundancy, a small language core, and good support for program structure. When evaluating existing candidates, they all fell short. Some came closer than others (Eiffel was the language that came closest to our wish list), while the most popular object-oriented language at the time, C++, scored worst. C++ was popular in many teaching institutions because it was seen as authentic: it was heavily used in industry, and many departments saw this as an advantage. Use in industry was not one of our goals: in our view, using a dedicated teaching language for the first year was perfectly acceptable—Pascal had demonstrated this principle—and might even be preferable. Our goal was that students would be proficient in an industry-strength language when they graduated (after three years of study), but not necessarily after the first year. The goal of the first year was to learn foundational principles, not industry-relevant syntax. This meant necessarily that the education includes a switch of programming language at some stage. While this creates a pedagogical overhead, we considered this necessary anyway: any university level computer science education should include proficiency in more than one language. But it also meant that ease of transfer to other commonly used languages was explicitly included in the criteria for evaluating target languages.

This issue—transferability of learning to potential successor systems that students might use—will surface again in later discussions in this paper. It is one of the core principles of the design of educational systems, and we will come back to it below.

The other issue that is a constant in discussions of educational programming systems is syntax; we will have a lot more to say about this in our discussions below. At this point, we were calling for an "easy, readable syntax" similar to what Pascal or Eiffel were using. We were think-

ing in terms of traditional, text-based syntax forms. Other systems later would make much more radical advances in this area by introducing block-based programming, and with that changing the syntax question fundamentally. We will come back to this below.

For us, probably the most important insight from this evaluation was that one aspect was more important and more influential than any other single issue: the programming environmen

The Importance of the Environment

When we started formulating criteria for assessing suitability of languages for introductory teaching, we were initially thinking about language characteristics. In evaluating a number of systems, it became clear to us very quickly that the development environment in which a language was used had a major impact on the outcome. So much so, that the quality and nature of the environment had—in our view—a stronger influence than any single characteristic of the language (Kölling, 1999c).

As a result, we made the design of the development environment one of the primary goals of our project. This design had to address novel problems in teaching caused by the switch to object orientation: more complex program structure and higher level abstractions

The Problem: Object-Oriented Structures

In the time before object orientation, most departments used structured (procedural) or functional languages in their introductory courses. The source code for typical beginners' programs was usually contained in a single file, program execution could relatively easily be traced on paper, and use of a stand-alone text editor and command shell for execution was the most common mode of work.

With the advent of object orientation, this changed.

Even small programs now consisted of multiple classes—and with this, multiple files—and both abstractions and practicalities became more complex. On the practical side, students had to deal with multiple source files and dependencies (the Java CLASSPATH setting was an example that caused regular problems), and maintaining an overview of a complete program source became more difficult. But more importantly: no support existed to understand and manage the increased complexity of abstractions inherent in these new systems. We now had classes and objects, instantiation, object interaction, and control flow across multiple source files. Both the static and the dynamic aspects of programming had become more complicated, yet the tools had not adapted.

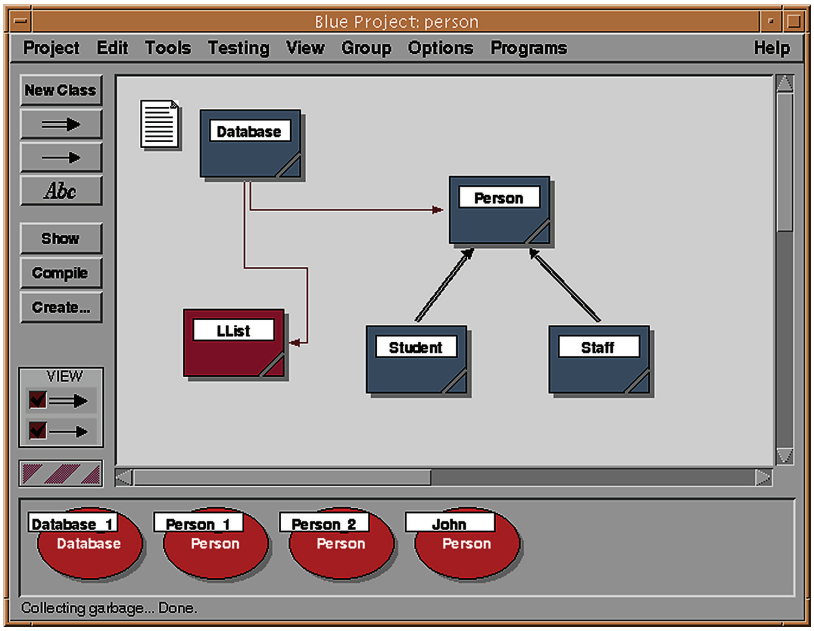
A common complaint of teachers at the time was that students found it very difficult to understand the difference between a class and an object. This was not surprising: since common programming environments concentrated on displaying lines of source code, students were thinking about lines of code. Little support was given in existing environments to understand or interact with class or object structures.

This—the object model, not the syntax—turned out to be the most difficult aspect of the new form of programming.

The Solution: Visualization and Interaction

Our attempt to address this challenge was the design of an integrated language and environment that explicitly supported an object-oriented model, and provided visualization and interaction functionality to investigate and experiment with the underlying abstractions. We started work on such a system, named Blue (Kölling, 1999a) in 1994 (Figure 1).

Figure 1. The original Blue environment. The main part of the window shows the class diagram; along the bottom, objects are displayed on the object bench.



First, we made the decision to concentrate on an integrated environment, rather than separate editing, compiling and runtime tools. This reflected our view about the importance of the development environment in teaching and learning: only if we controlled the environment could we achieve the full pedagogical benefit we were aiming for.

One advantage was that an integrated development environment (IDE) made it possible to overcome many of the practical problems: compilation and execution dependencies could be managed automatically, and various practical issues could be avoided.

More important, however, was the ability to provide tools for visualization and interaction: class structures were visualized in a diagram, objects could be selectively instantiated and methods could be interactively invoked with no need to write test drivers[3].

Putting the class diagram at the center of the system, in front of the eyes of users before they could see lines of source code, reflected our belief that the truly important (and more difficult) aspects of object-oriented programming were not syntax, but program and object structures. In our system, users could not avoid seeing structure, and thus were encouraged to think about it.

By allowing and visualizing interactive instantiation, and by showing objects graphically as separate from classes, we encouraged construction of mental models of program execution that are otherwise difficult to convey.

Overall, the tools for visualization and interaction were the most important contribution of the Blue system. At the base of this was a belief in *active learning*: that the act of experimentation with small parts of code—single methods in our case—and quick turnaround in the edit-compile-run cycle made the most significant contribution to a thorough understanding of programming concepts.

Another Necessity: Simplicity

Integrated environments were not new at the time; many were in use, including environments for object-oriented languages. What was new was a dedicated, object-oriented development environment *for teaching*. And one of the most obvious distinctions to existing environments was the simplicity of its interface.

Existing IDEs were usually designed for professional developers. They offered large amounts of sophisticated functionality, which was useful and important in professional contexts. Most of them had well over 200 buttons and menu items visible in their main interface.

For teaching, this power of functionality becomes a problem.

We wanted to teach about programming concepts, not about the IDE. By far the largest part of the IDEs would never be used by students in an introductory course, and the presence of these functions becomes a hindrance. Students needed to use only a few of these functions, but they often did not know *which* functionality they should know about.

Professional IDEs looked intimidating, and students often did not become comfortable in the environment before the course ended.

One design goal for Blue was to create a simple looking interface. We worked very hard to have very few buttons and menu items initially visible. Simple does not mean simplistic: the environment had some sophisticated functionality, but the interface for the user had to be as simple as we could make it.

This completes the three main principles that survive until today and formed the primary guidelines for all extension and development that was to come later: visualization, interaction, and simplicity.

Other projects attempted to achieve similar simplicity by cutting down existing professional environments for beginners. This would have the advantage of having a simple environment for beginners, which could then be extended into a full professional environment by uncovering more advanced tools, without necessitating a change of environment. Gild for the Eclipse IDE (Storey *et al*, 2003) and Visual Studio Express (Microsoft, 2016) were examples of such projects. Ultimately, though, these efforts failed; the systems were discontinued or did not manage to achieve significant traction in programming education. We believe that the primary reason that they did not succeed is that they viewed a beginners' environment as a subset of a professional environment. Blue, on the other hand, was not a subset of any existing environment. We not only needed *fewer* tools, we needed *different* tools. Pedagogically motivated tools, such as direct interaction and visualization, were just not available in commonly used IDEs at that time.

Integrated Environments and The Acceptability of Black Boxes

In the second half of the 1990s, when Blue was published, a heated debate raged for quite a few years among teachers of introductory programming courses: should students use an IDE, or should they use a stand-alone text editor and a command line environment?

Proponents of the editor/command line argued that only with this tool set would students properly understand how programming works. Using these lower-level tools was somehow "good for the soul". In using an IDE, some teachers argued, too many important steps were hidden, too much code automatically generated, and students would not properly learn all the necessary steps and gain a thorough understanding of important detail. Teaching with IDEs, it was feared, would turn out students who could somehow produce a small working program, but without deep understanding of the principles.

In this argument, "using an IDE" was often conflated with "using a GUI builder", since the most famous IDEs at the time often had a GUI builder as their most prominent tool. This is where the argument about auto-generated code originated, and it is, of course, a fallacy: IDEs do not need to include GUI builders (and Blue quite consciously didn't).

The side arguing for IDEs (which, it should be obvious, included our group) argued that there is no such thing anymore as seeing "how it really works". Using a compiler for a high level language in itself is an abstraction hiding multiple layers of lower level technology. Deciding what the right abstraction level is for introductory object-oriented teaching should be a pedagogical decision, not an accident of history. Do all students need to know the command line for their first programming encounter? Do they all need to know assembler? Machine language? Processor and other hardware details?

Arguing that various layers below could be treated as black boxes, but that using a separate editor and command line was essential, seemed arbitrary, and we argued strongly for the use of integrated environments. With these, we were convinced, if the toolset was right, students would learn more, not less.

Time has since intervened to answer this question: almost all introductory courses (and professional developers) use IDEs today, and this discussion has disappeared. In the late 90s, however, it was a much debated issue.

The Switch to Java: BlueJ

Blue was first published in 1995 and used for introductory teaching at our computer science department at Sydney University (and—to our knowledge—never used with students anywhere else). In the same year, the Java Programing Language was published.

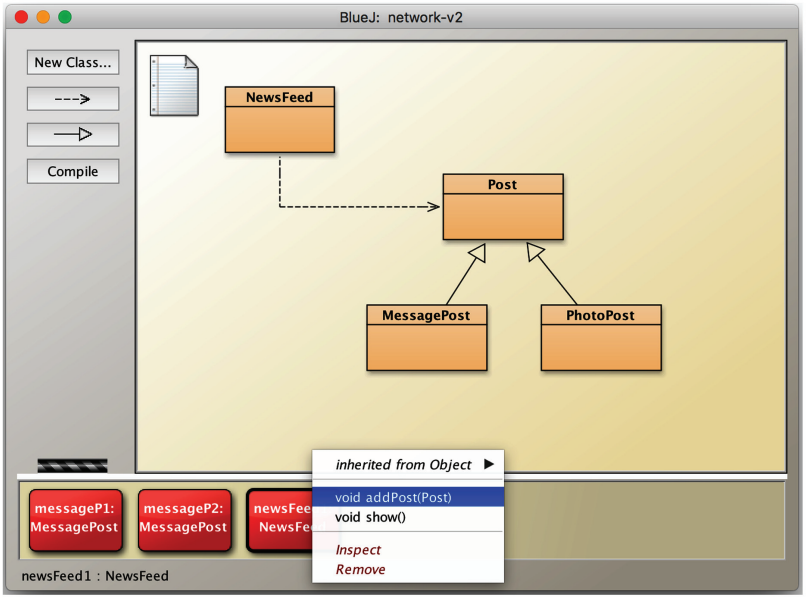
Blue had started as an academic project (a PhD), and by the end of the decade, when the PhD had ended, we had to decide whether and how to continue. While Blue was attracting supportive comments in some academic circles, it was clear that it would be hard to develop and support it to a level where it might be widely adopted. The "team" still consisted of one (former) PhD student and a supervisor.

Java, in the meantime, gained popularity very quickly. It obviously met a need in the market (the need for a simple, free, well supported modern object-oriented language) and was very well supported by a large company, Sun Microsystems. We were still convinced that Blue had advantages over Java in some specific aspects; we preferred its syntax and some language constructs, but most importantly: Java did not have a visual, integrated environment for teaching and learning.

We faced a decision: we could continue what we were doing with Blue, and remain a small research project noticed by a handful of academics, or we could throw away half of our project—the Blue language—, use Java instead, and continue working on the programming environment as our contribution to the state of the art. This way, we would give up some of our work, but may have the chance to have the other part—the environment—potentially adopted by actual users. We chose the latter path. It seemed the more interesting opportunity.

In March 1999, we released the first version of BlueJ, a re-implementation of the Blue environment for the Java language (Figure 2). BlueJ not only supported Java as the user level language, it was also implemented in Java itself (while Blue had been implemented in C++). The promise of cross platform development that Java brought to the table, with its virtual machine architecture and just-in-time compilation, turned out to be vital for us: over the following years, we managed to support BlueJ on a number of different operating systems with a very small team. This would not have been possible without a cross-platform implementation language.

Figure 2. The BlueJ environment. The class diagram is similar to that in the Blue environment. As in Blue, objects are shown on the object bench and methods can be invoked by right-clicking on the object.



Changing to Java as the language was comparatively easy. Blue and Java had a lot in common. The object models were quite similar, and all relevant major abstractions matched very well. This meant that the environment could be recreated for Java without any significant design changes.

The change to Java was successful in creating more interest in actual adoption of our system. Soon after publication, other departments started adopting it for their introductory courses.

The Importance of Material

The single most important aspect of the design of the Blue/BlueJ environments for us was that they allowed a different pedagogical approach to teaching programming. It allowed to focus on the important fundamental principles—objects and classes—first, before getting bogged down in syntax and incidental detail. This aspect was so important to us that we made "Objects First" the title of a series of seminars we offered (Kölling & Rosenberg, 2000)—a term novel at the time that later came to be widely used to represent this general approach to teaching object-oriented programming (as opposed to the *objects-late* approach favored by other educators—a related, but separate, ongoing debate (Astrachan, Bruce, Koffman, Kölling, & Reges, 2005)).

When BlueJ started to be adopted by other teachers, however, we made a surprising (to us) discovery: many educators still started by using and teaching Java's *public static void main* method, using a single class and only static methods (or often: a single static method) for their entire project. In other words: they started exactly with the concepts and constructs we had intended to avoid: syntax, obscure magical incantations of advanced concepts present for incidental reasons, and small scale statements ignoring objects. That BlueJ allowed teachers to circumvent this standard Java trap and discuss (and experiment with) more important constructs first seemed to have little impact. Many teachers ignored these features. The question was why.

When we first published Blue and then BlueJ, we had a somewhat naïve view of how adoption might work. We had designed an environment to allow a different pedagogical approach to teaching (which we thought preferable), and we assumed that teachers would see its value, adopt the environment and follow this teaching approach.

In practice, this did not happen, at least not to the extent to which we hoped it would. Instead, a good number of teachers started to use BlueJ, but did not adapt their teaching approach. They continued to teach structured programming, now in Java, often with examples and projects ported directly from earlier Pascal or C versions. BlueJ was used, but *objects-first* was ignored.

We observed this phase with mixed feelings. It was good to see an increasing number of people adopting BlueJ and starting to teach an object-oriented language, but we could not avoid feeling that *they were doing it wrong*. For us, the whole point of developing BlueJ was to allow an objects-first approach, with interaction, experimentation, and concepts before syntax. Instead, many lecturers adopted BlueJ because it was easy to install, had an easy-to-use editor and good tool integration. They used it as they would use any other environment, but ignored what we thought of as the most important features.

We were not sure about the reasons: did lecturers not know how to use (and teach with) object interaction, or did they disagree with this approach and chose not to? In any case, we realized that to get our message across, making the software available is not enough. We needed to talk more explicitly about pedagogy.

We spent the next few years delivering a series of workshops and seminars, and eventually publishing a textbook (Barnes & Kölling, 2002). The seminars were largely successful—a good share of participants adopted an objects-first approach and BlueJ—but did not achieve significant scale. The book managed to reach a much larger audience. After the publication of the textbook (in 2002), adoption of BlueJ with an objects-first teaching approach increased substantially[4].

Later Extensions

With increasing numbers of users came an increasing number of suggestions and requests for additional features and functionality. While many of the suggestions were sensible, taken together they would have entirely ruined one of the most important aspects of BlueJ: the small size of the system and simplicity of the interface.

We established criteria for selecting areas of functionality that we did consider for inclusion: the feature had to be useful and widely used for introductory teaching at first-year university level (our target user group). We did not aim at supporting later work, more advanced workflows, or program sizes beyond what one might encounter in a first year course. In practice, there was an additional, highly subjective, criterion: we had to be able to envision ourselves using it in our own course.

While we made many and frequent changes and improvements to the system (we typically released two or three updates per year), additions of larger areas of functionality were very conservatively controlled. The main ones were the addition of explicit unit testing support in 2003 (Patterson, Kölling, & Rosenberg, 2003), ad-hoc single-statement evaluation (a *read-eval-print-loop*) in 2004, and support for repository-based team work tools in 2007.

The Lessons Learned

So far in this paper, we presented a lengthy discursive history of our Blue and BlueJ projects. The purpose is to extract some lessons that may be useful in a more general context. Before we go on to describe development from this point forward, we will summarize the main lessons we learned from our experiences so far.

1. **Visualization, interaction, simplicity.** The most important aspects of our system, which were crucial to its success, are these three overarching design goals: visualization, interaction, and simplicity. For any educational programming environment, designers should identify the main concepts to be learned by the users, and make those visible as first class entities in the user interface. They should then design interactions that illustrate the characteristics of these conceptual abstractions and allow users to experiment with them. This has to be balanced with a simple, clear interface. Every other design goal and design decision has to become subordinate to these goals.
2. **Know your target group.** One of the most important aspects to be clear in the minds of the designers are the characteristics of the system's target user group. This sounds obvious when stated explicitly, but is nonetheless easy to lose track of. Many competing ideas will have to be weighed, each competing for attention, interface space, and development time. It happens easily to get excited by an idea, and be tempted to add it, because it is clearly useful for someone. The crucial question to ask is: is it important *for our target group*? Many ideas are good ideas, and still do not belong in your system.
3. **Do one thing well.** The narrower the user group is chosen, the better job the system can do to serve it. Once you have chosen your well defined user group and application area, build the best tool you can *for that group*, to the exclusion of everyone else. It is tempting—but counter-productive—to try to be everything for everyone.
4. **Beware feature creep.** Very closely related to the two previous points is the importance of saying no. As soon as you have a useful and successful system, someone will start using it for application areas at the edge or outside of your envisioned domain. They will ask for additional features and extensions to support their tasks. Often, these are very reasonable and interesting ideas, and would extend the usefulness of your systems to new users. Going down this path, however, always opens the danger of unintended consequences. Feature creep—while very hard to avoid—is the mortal enemy of simplicity. And simplicity, as mentioned earlier, is one of the fundamental, immovable goals. Often, in the years of leading the development of BlueJ, one of our most important tasks for the project has been to say no to people. A good idea might be a good idea, and still not fit well with *this* project. When in doubt, leave it out. Features can always be added later, but it is nearly impossible to ever remove them to regain simplicity.
5. **Do not be afraid to make decisions.** Often, when designing a system, design decisions have to be made. Sometimes, two competing options seem equally good, or might be a matter of preference of different users. A common response is to allow both versions, for example having individual preference settings for system look or behavior, or allowing alternative syntax for a language construct. In education systems, this is a fundamental mistake. Every additional alternative forces users to make a choice. Beginners often do not have the knowledge to make an informed choice, or do not care. We are not doing users a favor by asking them to make a decision about an aspect that they either don't know or don't care about. In addition, every possible variant complicates the development of teaching material. Your job as a designer of an educational system is to make the choice. If some users don't like it—bad luck. The system will be more usable and users more productive if the system is clearer and simpler.
6. **Availability.** It is important that a system is easily available, both to institutions and individuals. This means that it has to run on multiple popular operating systems, and be affordable. (The most affordable price, of course, is free.) Whether the system needs to be open source, however, is much less clear. BlueJ was initially free, but not open source. We received regular complaints from users who demanded an open source system, and stated that they

need to make source amendments. We eventually open-sourced the system, mostly to allow inclusion into Linux repositories and distributions that would not consider closed-source systems. There are, however, no contributors to the BlueJ source code outside of our team, and open-sourcing seems to have made no practical difference other than silencing those people demanding it on principle.

7. **Support material matters.** A software system—no matter how well designed—does not, by itself, make an impact in education. What actually drives adoption is an ecosystem of support material. This can take different forms: a textbook, as in the case of BlueJ, or an active, supportive user community (as we discuss below). In any case, what makes teachers adopt a new system is—by and large—not the software itself, but the opportunity to teach differently. To make teachers see these opportunities, context is required that shows them what is possible. This can be a book, online videos, or a supportive group of peers in a shared community. The software and the pedagogical material are both equally necessary—one without the other will fail.

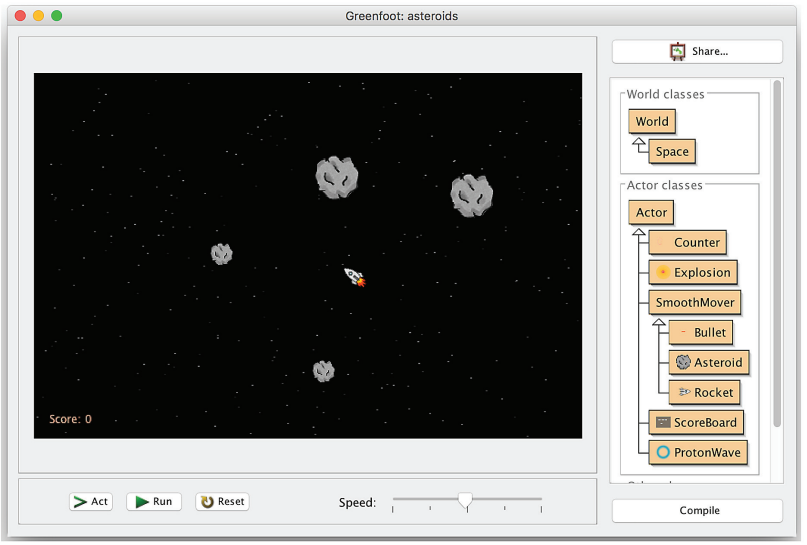
GREENFOOT

In 2004 we started work on a new system, named Greenfoot (Henriksen & Kölling, 2004) (Figure 3), which was first published in 2006. A number of different developments led to the decision to work on a new design. The two main motivations came from two different directions: one was technical, the other was driven by a change in the user group. We discuss both in

Object Look and Behavior

In BlueJ, classes could be instantiated, and the objects were visualized as red rectangles with rounded corners on the object bench. When the objects in question were representing graphical

Figure 3. The main window of the Greenfoot environment. The class diagram is on the right; the main part of the window is used to show the world.



entities, a disconnect developed. We had, for example, one project frequently used in early teaching that showed graphical shapes (Figure 4), which could be displayed in a separate window and be manipulated via method calls. The manipulation (i.e., interactive method calls), however, had to be done on the object bench, while the visible effect occurred in the separate window. Every object was duplicated and possessed two representations. This was irritating.

We started to think: what if the objects on the object bench could change their appearance, or their position? What if they could react directly and visibly to interactive method invocations?

Changes in the Novice Programmer Population

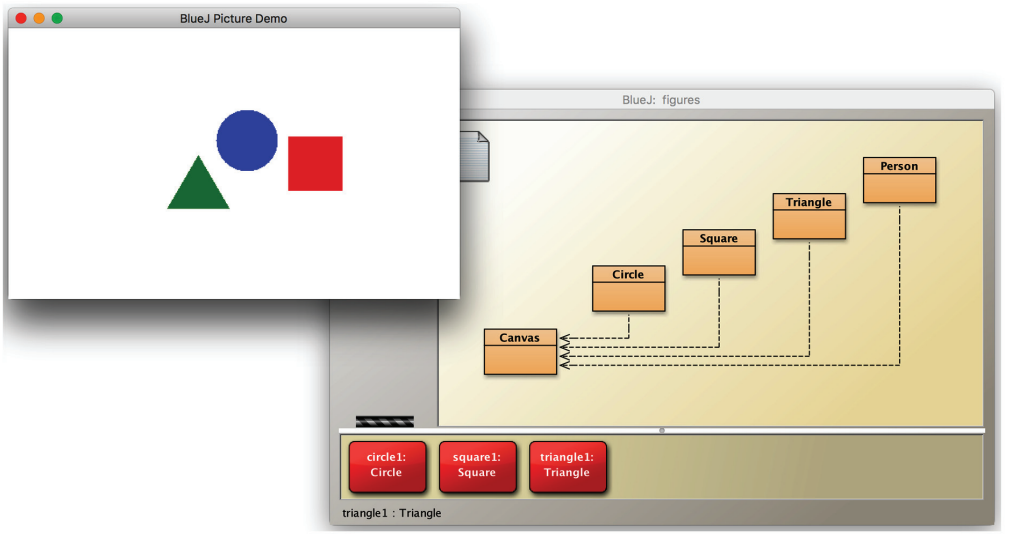
At the same time, over a number of years, a fundamental shift happened in programming education: for more and more students the introductory university course was not the first point of contact with programming anymore. Many beginners were encountering programming at school age, often in formal school instruction and sometimes on their own. Concurrently, in the first half of the 2000s, enrolment numbers in computer science courses were declining in many western countries.

We were always interested in the teaching of initial programming, and supporting that area of instruction as well as we could. When we worked on Blue and BlueJ, the context in our mind had been the introductory university course.

Now we were in a situation where pupils encountered programming—and often decided that they are not interested in studying it—well before they ever came to us in our university departments. If we really wanted to influence early learning of programming, we had to target school age learners.

BlueJ at the time had a significant user base in schools, mostly in the last two or three years of school education: about a quarter of our users were in schools, the rest mostly at universities. However, even though some teachers thought it worked well for them, we felt that this was not the right system for initial programming at school.

Figure 4. The BlueJ environment with the figures example. Objects are represented on the object bench, and then again graphically in a separate window.



New Target Group, New Goals, New Design

The changed target group—school pupils instead of university students—changed our goals, and with it, our design. The most fundamental question we asked ourselves is: what is the most important thing we are trying to achieve? And that question had different answers at the two different age groups. For BlueJ, our goal was to teach the fundamental concepts of object-oriented programming well. We wanted students to develop a clear and consistent understanding of the most important and useful concepts. For Greenfoot, it was different: the main goal was motivation.

The Importance of Motivation

The most important difference between the two target groups—secondary school and first-year university—is that the latter is a selective group. All the students have chosen to be there, so a certain level of motivation can be assumed. Students generally have an interest in the subject.

This is not true in secondary school. Programming there is presented to the complete population, and many of the pupils were—by default—not interested. This does not even have to do with the manner in which programming is presented: strong pre-conceived opinions often existed about programming and programmers—even among those who had never practiced it—and the aversion was already well established *at the start* of programming instruction.

This means that the first and most fundamental problem of a programming tool dedicated to this age group was not to teach them something, but to *get them interested*. Generating motivation—and achieving this quickly—became the main goal.

Plus Ça Change, Plus C'est La Même Chose

These two strands of thinking—the importance of motivation, and aiming for a more visual object bench—merged into the design of Greenfoot, a new environment aimed specifically at school age.

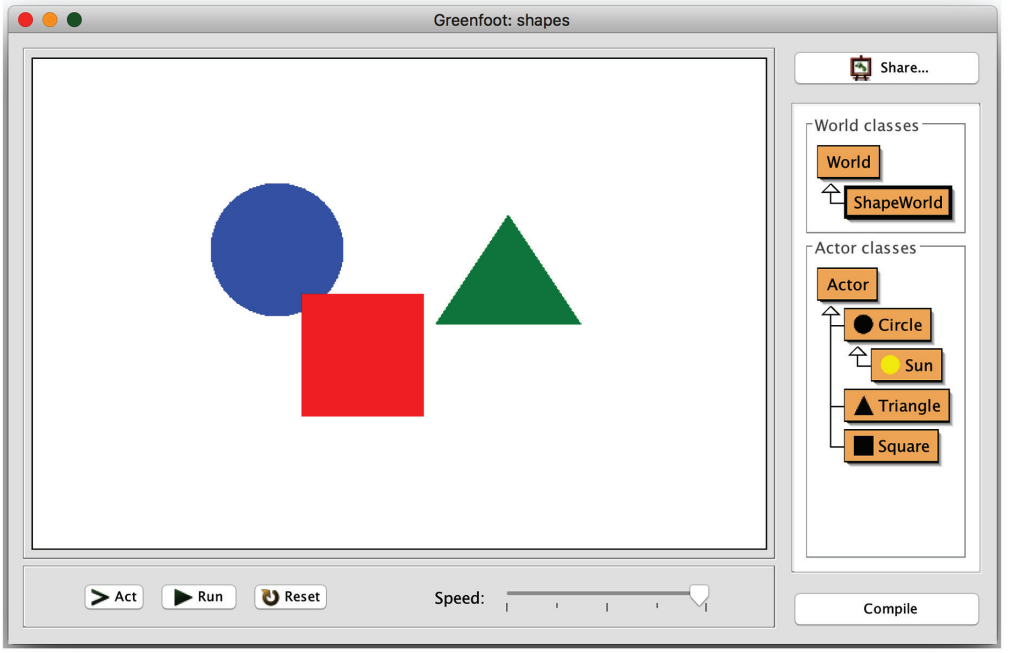
To increase motivation, we decided to concentrate on very visual, graphical, animated examples, where it is quick and easy for beginners to create a first interesting looking program. This had to include the ability to easily make use of graphics, animation, sound, and keyboard control, so that interactive games become achievable as a first example. This approach, called "Game-first programming" by Goschnick and Balbo (2005) and shown to be effective in increasing motivation, had been impossible in BlueJ.

To achieve the goal of a more individual object representation, we decided to push the class diagram, which dominated the interface of BlueJ, to one side, and make the object bench the central part of the user interface (Figure 5). This object bench was now called the *world*, could be styled with a custom background, and objects were given individual appearance, location, and rotation. They could then be programmed to alter their location and appearance, and this change could be automatically visualized. With this, we arrived at a well known destination: micro-worlds.

In a sense, we had come full circle: we were doing things very similar to what Turtle Graphics and Karel the Robot had done decades before. In fact, both Turtle Graphics and Karel the Robot became very easy-to-implement examples in Greenfoot. But Greenfoot was more: Firstly, the micro-world characters could not only be programmed, they could also be interactively manipulated. Characters could be instantiated easily, placed, methods invoked, state inspected, all without writing code. This allowed experimentation to support understanding of the system.

Secondly, more characters could be created. While Turtle Graphics and Karel provided exactly one programmable actor, Greenfoot could produce any number of actors. This is one of the advantages of object orientation.

Figure 5. Greenfoot with the shapes example: Objects have a single representation that is both graphical with custom appearance and interactive.



Thirdly, Greenfoot was not a library, but a complete development environment. It included an editor, runtime system, debugger, graphics library, and more. Programming and execution was more flexible, practicalities easier to handle, and interaction more interactive.

But most importantly, Greenfoot was not a micro-world, but a *micro-world meta framework*. While Turtle Graphics and Karel (and many others over time) presented a single world with a fixed set of possible actors, Greenfoot was not a micro-world; Greenfoot allowed the creation of endless micro-worlds.

The design was carefully created to be generic enough to allow the creation of any program whose output was primarily two-dimensional graphics. Possible projects included micro-worlds and birds-eye-view games, but also other examples such as platform-style games, simulations of ant colonies, a playable on-screen piano, simulations of solar systems, physics simulations, and many more.

In short, while Greenfoot restricted the application domain to two-dimensional graphical applications, it was still a generic programming environment.

The Next Lessons Learned

After more than ten years of continued development, releases, user feedback and maintenance of the Greenfoot system, we can again summarize the major lessons we learned from this experience. Again, most of these are not new insights—others learned and expressed the same lessons before, arrived at the same conclusions from different directions, or might find some of these obvious—but we hope identifying what we consider the most important points might help to

prioritize: there are numerous demands and goals for the design of educational development systems, and our contribution is to suggest that these are the ones that are fundamental to an extent that they should take precedence over competing alternatives.

1. **Motivation.** The most important lesson for us from our experience with Greenfoot is the power of motivation. As university teachers, when we started out in our designs of educational systems, we were very much concerned with *What to teach* and *How to teach*. The Blue design reflected this thinking. With Greenfoot, we concentrated on creating motivation and a sense of ownership. The goal was to get users into a situation where they *want to learn*—if we can achieve that, we can teach them anything. For us, this constituted a significant shift from a teacher perspective to a learner perspective. This also meant that self-directed learning (and use of the system without a teacher present) became a significant design goal. These goals shape the design decisions and functionality of the system. Other systems—most notably Scratch—started with this insight from the beginning. Their designers knew what we took a long time to learn: that in the learning process the learner is more important than the teacher. We will come back to this below. Almost all the other points listed below are in support of this one: they are more concrete aspects helping to increase learner motivation.
2. **Taking control.** To allow users to become self-motivated, it is important to put them in a position where they can take control and ownership of a project quickly. In programming, this is not easy. Since novices at the beginning usually do not know any programming constructs, they can often only do what they are shown to do. In many introductory approaches, this leads to a fairly mechanical observe-and-copy methodology, where students re-create code shown to them by an instructor (or a book, or a video), and then create minor variations (e.g. in parameter values). To create true motivation, this is not enough. The goal of a well-designed system must be to get users into a position where they can set their own goals and then try to achieve them as soon as possible. For this, aspects of the system have to be easily changeable, and discoverable. Most block-based systems achieve this very well by presenting block palettes and a mechanism to try out blocks very quickly and easily. This supports discoverability and recognition of language elements not present in text-based languages, and supports self-directed learning. In text-based systems, this is harder: language constructs are not easily discoverable and typically require explanation; syntax must be learned. In Greenfoot, which initially provided only Java with a traditional text-based editor as a user language, we tried to overcome this by adding other opportunities for taking control. Even before knowing much about programming, users can change images and sounds, and easily create and arrange additional actors. This often leads to widely different background stories for early programming examples, even though the programming constructs used are very similar. When introducing programming to groups of teenagers we often see projects of, for example, rabbits eating pizza, space craft picking up astronauts, or cars racing each other. In a typical case, a teenager saw an image of a wasp in the built-in icon library and was immediately excited to create a project with a "Hunger Games" theme (a popular book and movie at the time, where killer wasps feature in one of the story lines). Even though the programs pupils created are near-identical from a programming point of view (they all concerned one or more keyboard controlled actors moving across the screen and interacting with another kind of actor), each of the users had their own story that they had decided on themselves. Sounds also had a very powerful effect on the sense of ownership: the easy ability to record and add sounds to projects (which allows pupils to add their own voices to their games) led to much excitement and engagement. In short: any environment should

strive to let users take control as early and as far as possible, and the extent to which this can be done will greatly influence motivation, and with that, acceptance.

3. **Visualization and experimentation—again.** Greenfoot again reinforced our belief in the importance of visualization and support for experimentation. Greenfoot is more visual than BlueJ and—especially for younger learners—this was an important characteristic.
4. **Visualization of program execution.** In addition to visualizing the program components (classes and objects), Greenfoot also visualizes the execution of the program. While the program is running, actors can be seen to move on screen. When the program contains an error, this is often immediately obvious in the unexpected behavior of a visible actor. This visualization of the program as it is running is a very fundamental tool that aids greatly in the construction and understanding of programs. This is the fundamental design idea that Turtle Graphics popularized in the 1960s and 70s, and it is inherent in all micro-world related environments. Many later systems, however, do not make use of this, so we feel it is worth reiterating here.
5. **The power of community.** In 2007, one year after the initial publication of Greenfoot, we added an export feature that allowed users to upload their projects to a public website, where others could then execute, download, and comment on these projects. Scratch, independently, added a similar feature in the same year. In the Scratch community, this is taken a step further: users are encouraged to "remix" projects (upload a modified version), and these remixes are tracked and attributed. Remixing, for the Scratch community, turned out to be a powerful motivator for engagement. In each of these communities, members can build a reputation by providing useful content or helping others. These user communities have a high impact. They work in two distinct ways: they serve to provide help and guidance when users have questions or problems, and they greatly increase motivation by giving an opportunity to share and publish projects, and by receiving feedback and encouragement from others. Both aspects are strong drivers of motivation and engagement.
6. **Teacher communities.** In support of teaching with Greenfoot, we created the Greenroom (Fincher, Kölling, Utting, Brown, & Stevens, 2010), a community for teachers. As opposed to the Greenfoot user community, the teacher community is not public and not anonymous—teachers have to apply for access, and we ensure that members actually are who they say they are. The Greenroom provides resources for teaching and learning, and a discussion forum for members. The advantage of excluding public access is two-fold: firstly, tests, examinations and projects for assessment can be published here with solutions, without pupils having access to them. And secondly, teachers can talk much more freely, admit more easily when they have problems, and ask for help in the knowledge that they are talking to their peers without being overheard. When creating this community site, we evaluated various existing platforms, but decided eventually to create our own. Aspects influencing this decisions included questions of resource curation, access control, and encouragement of participation. These aspects are further discussed in (Brown & Kölling, 2013). The Greenroom plays a significant role in the pedagogical support of the system, by providing teaching material and support, much beyond what we could provide directly ourselves. For us, coming from a teaching-oriented perspective, this was a natural development. Other systems, such as Scratch, which developed from a more learner-centric view, also moved in this direction. Scratch was initially designed with self-directed learners in mind, consciously assuming that a teacher might not be present. However, as Scratch became widely used, it was also often used in classrooms, and many teachers started to look for support and material. Providing support for instructors will be crucial for every popular system.

7. **Programming language.** The programming language used within Greenfoot is Java. (A second alternative, Stride, has recently been added, discussed below.) Java is a traditional, text-based language originally designed for professionals. This has a major influence on usability and potential target groups. Using a language such as Java imposes a hard limit on the lower age bound of potential users. For Greenfoot, we aim at users aged from 13 upwards. Below this age, many children do not have the necessary typing and abstraction capabilities to cope with the text-based editor and resulting syntax errors. Languages avoiding this type of syntax, most notably block-based languages, are usable at a much lower age range and can better focus on the initial learning of fundamental concepts. Using a standard text-based language has an advantage only when the learning of syntax and text-based programming is an explicit goal of the learning process. For us, the choice of Java as the user language was pragmatic: as a team, we did not have the capacity to design and implement a language including all necessary libraries at the same time as developing the environment. Blue, for example, never had the scope of libraries that were now routinely expected in modern systems, and we did not want to spend multiple years on implementing a new language. Choosing Java allowed us to reuse significant portions of the implementation of BlueJ and made the project possible, even though it created limitations in its potential use. The choice, however, also has positive aspects: while it imposes a lower age boundary, it removes the limit at the other end. Since the system is full, standard Java, and is executed on a standard Java VM, very complex and sophisticated projects can be created and run efficiently. As a result, Greenfoot is occasionally used, for example, in artificial intelligence courses for the implementation of sophisticated AI algorithms. Overall, the most important aspect of programming language choice is to be aware how the style of syntax and language limits the potential user groups, and to ensure that the language is appropriate for the target group of the system.

RELATION TO OTHER ENVIRONMENTS

Our environments were not developed in a vacuum, and many other learning environments were published at the same time. In this section, we briefly discuss selected aspects of some other systems, and how they relate to our environments.

The Rise of Blocks

Arguably the most significant impact on early learning of programming is made by the availability and popularity of block-based programming systems, a modern incarnation of a Visual Programming Language (VPL). Visual programming languages have a long history, starting with early graphical systems such as Sutherland's Sketchpad in the 1960s (Sutherland, 1963) and going through many iterations and variants (Boshernitsan & Downes 2004). However, only with the development of modern block-based education environments did these systems escape a niche existence where they are not only of interest to researchers and hobbyists, but achieved large scale adoption by practitioners in programming teaching.

Block-based systems, by side-stepping most syntactical problems, have greatly shifted the possible starting age of programming learners downwards. Seven- or eight-year olds can comfortably use some of these systems and learn fundamental programming concepts in the process.

The early block-based languages, such as Alice and Scratch, were embedded in environments that offered a host of other advantages as well: visual micro-worlds, simple execution models,

block palettes that made language constructs discoverable, easy experimentation, engaging context, and user communities. Later systems, such as App Inventor (Wolber, Abelson, Spertus, & Looney, 2011), added development for mobile devices as an additional motivator.

While we cannot fully separate visual from textual programming (Good (2011) quite justifiably stipulates that "'Visual vs. Textual' is no longer a useful distinction for programming languages" since each makes significant use of elements of the other), the terms *block-based* and *text-based* programming are sufficiently clearly understood to form a useful distinction in the discussion of educational systems. The impact of this aspect is, in fact, so fundamental and so lasting that in any categorization of educational programming environments today the division into to block-based and text-based environments might be the first, top-level category choice.

Block-based systems have, over the last ten years, managed to bring programming to large groups of users that were previously considered too young. Text-based systems, at the same time, continue to thrive, with the change to text-based programming currently seen as an important step towards a fuller programming education.

In the remainder of this section, we briefly discuss some aspects of selected block- and text-based systems where they relate to our own environments.

Selected Other Systems

Scratch

Maybe the most interesting system to discuss in comparison with Greenfoot is Scratch. Scratch and Greenfoot were designed and published at about the same time, and thus developed independently. The interesting aspect is that—despite significant differences in many details—many design decisions follow very similar paths and arrive at similar solutions, despite the difference in target group and concrete functionality.

Scratch started with a learner centric view that placed discoverability and experimentation at the center of the design[5]. It uses a custom block-based language and a concurrent, object-based programming model. In these aspects it differs from Greenfoot. However, many underlying design goals align: the use of a micro-world to visualize execution, the goal of supporting easy experimentation, the value of simplicity over extended functionality, the importance of community, and the clear sense of target group. Scratch could have equally been used as an example to illustrate the main conclusions of this paper.

The different age of the target group leads to very different concrete design choices in many cases, and in some aspects—such as the simplification of syntax—Scratch goes much further than Greenfoot. However, it is interesting to observe that the fundamental principles are constant. The two systems illustrate two different implementations of similar design principles for different age groups and different contexts, thus providing two examples of similar abstract ideas.

Because of their similarity of many important aspects, Scratch and Greenfoot form a well-working possible sequence for learners as successive systems.

Alice

Alice is notable as one of the early successful block-based systems that attracted a large user base in many schools and with many individual users. It has some unique characteristics: firstly, it uses a three-dimensional (instead of a 2D) world. The Alice team has argued that the 3D nature of the system adds to its attraction and creates engagement. On balance, evidence for this assertion is thin. Anecdotal evidence points both ways: some teachers report positive comments from users, while others question the benefit in light of increased complexity. Successor systems

developed later do not provide much support for this argument: 3D systems have not become more popular than their 2D competitors. Despite the open question of impact of this particular design decision, it is interesting to note yet again the same underlying goal: increasing motivation and engagement. Another interesting observation has to do with breadth of target group. Alice 2, the popular version early in this century, focused on its own block-based language for implementation. In 2007, the Alice team released a major new version, Alice 3, that added programming in Java as one of the major named goals. Alice 3 was intended to be usable for a very wide age group, starting in primary school and reaching into university level education. However, it failed to gain the same level of traction that Alice 2 had achieved earlier. Today, almost 10 years later, a significant share of the Alice user base still prefer to use Alice 2 with its more limited functionality. This may be an example how targeting a narrower user base may lead to a more successful system than attempting to offer more functionality.

Processing

Processing (Reas & Fry, 2003) is another environment that uses a variant of Java as its user language. It is interesting in our context because it presents another example of a successful educational system that makes use of a pre-existing programming language not originally developed for education. It is also interesting because it represents another example of a different concrete realization of the same design goals: leading to learning by creating engagement. Processing offers the ability to very easily and quickly create graphical programs with very quick visual feedback. In doing this, it combines motivation with learning of a traditional, text-based language and shows an alternative of creating engagement.

FUTURE TRENDS

Transition Issues: From Blocks to Text

The success of these two classes of educational environments—block-based for early learners and text-based for a slightly older age group—leads to a relatively new phenomenon: learners that transition from one to the other. Viewing these systems, and their respective successes, in isolation, is not enough anymore. With the earlier encounter of programming, often in primary school, many learners will now transition through multiple educational systems, and the combination, sequence, and transition between these should be planned in context.

One of the transition points generating most interest is the one from blocks to text (Armoni, Meerbaum-Salant, & Ben-Ari, 2015; Dorling & White, 2015; Hundhausen, Farley, & Brown, 2009; Powers, Ecott, & Hirshfield, 2007; Price & Barnes, 2015; Weintrop & Wilensky, 2015a; Weintrop & Wilensky, 2015b). Recently, teachers have discovered that this transition can create significant problems for learners, and that the added complexity of text-based systems—even when familiar with foundational programming concepts—can present a difficult hurdle (Powers, Ecott, & Hirshfield, 2007; Price & Barnes, 2015).

As a result, designers have started to look at blocks and text in combination.

Combining Blocks and Text

The most common approach to try to support this transition step is by providing a dual system: the programming system is able to present the same program both as blocks or as text, and users are able to switch between these two representations, or to view both of them side-by-side.

Alice 3 introduced a "Java Code On The Side" feature, which displays block based code under construction as Java code (Figure 6). Only the blocks can be edited; the Java code is read-only. *Tiled Grace* (Homer & Noble, 2014) is an environment for the Grace language (Black, Bruce, Homer, Noble, Ruskin, & Yannow, 2013)—originally designed to be text-based—that can show the same program alternatively as blocks or as text, with the ability to edit either representation, and offering an animated transition between them. Pencil Code is another system that allows this, either in its online environment (Pencil Code, 2016) or in the Droplet editor (Bau, 2015). Other systems, such as App Inventor (Wolber, Abelson, Spertus, & Looney, 2011) and a reusable library derived from it, Blockly (Fraser, 2013) offer similar functionality.

All these examples are based on a common assumption: that seeing blocks and text side-by-side (or alternately), and the same program in each, can aid in the learning of textual programming languages.

We believe, however, that this approach—though popular—is not an ideal solution to the problem, and that we can do better. Instead of offering blocks and text as alternatives, and transitions between them, we propose to offer a *single system* that combines aspects from both blocks and text. This then can serve as a stepping stone between the two worlds.

Frame-Based Editing: Merging Blocks and Text

In Greenfoot 3, published in 2015, we introduced a new language called *Stride*, and a new editor to manipulate its programs. The interesting aspects of this work are not in the language, but in its *frame-based editor*: it combines aspects from blocks and text into a single system, aiming to benefit from lessons learned in both (Brown, Altadmri, & Kölling, 2016).

In frame-based editing, some elements (such as scope) are presented graphically, while the overall text-flow is as sequential and aims to be as readable as a text-based program (Figure 7). Statements are represented by *frames*, which are inserted in their entirety with a single keystroke. No half statements can ever exist. Frames, when inserted, may contain slots—areas for nested code to be filled in (Figure 8). Two different kinds of slots exist: *text slots* and *frame slots*. While frame slots can receive nested frames, text slots are filled by typing (structured) text. The whole system can be used entirely keyboard-driven, without the need to use the mouse. However, if desired, frames can be manipulated as first class interface elements, including dragging them

Figure 6. The Alice 3 environment with side-by-side view of blocks on the left and Java code on the right. (Source: Cooper, S., Dann, W. (2015) The Role Of Programming In A Non-Major CS Course, ACM Inroads 6, 1.)

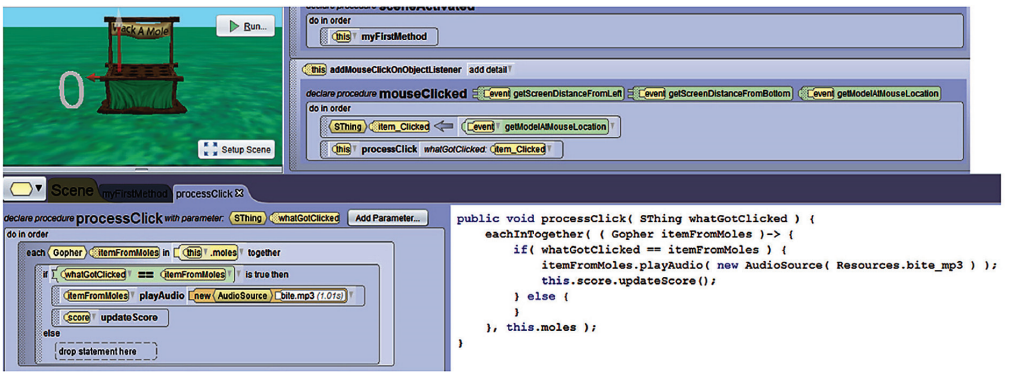


Figure 7. Greenfoot's frame-based editor with the Stride language. The source code is on the left; a clickable cheat sheet is shown on the right. Statements are represented by frames with colored background. Frames are entered with a single keypress.

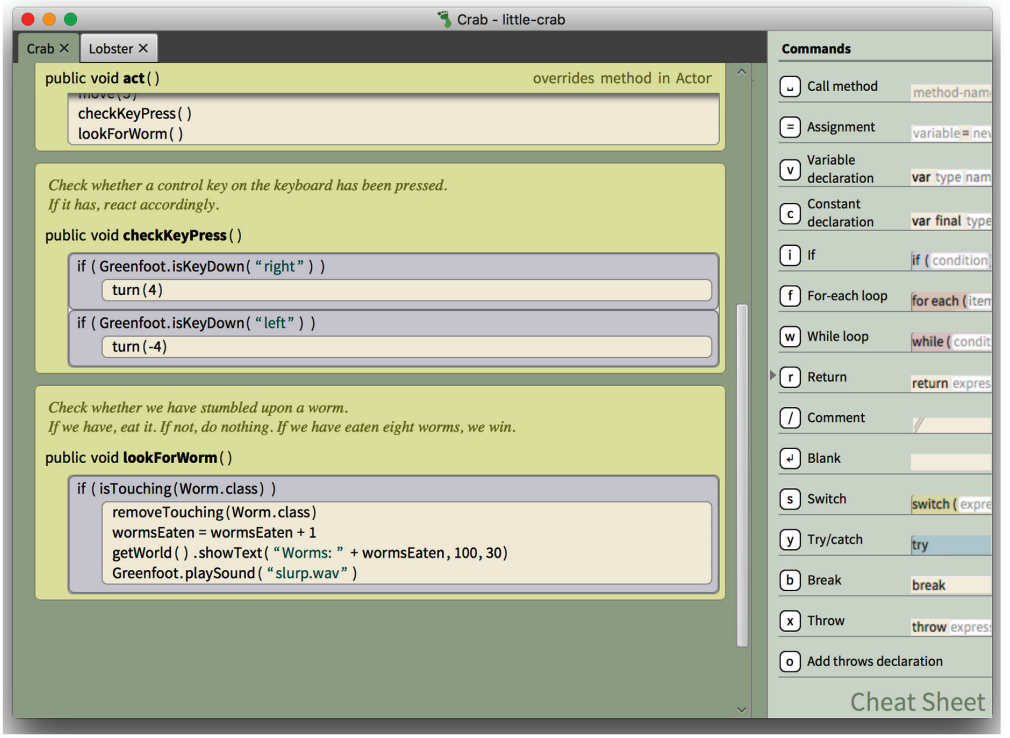
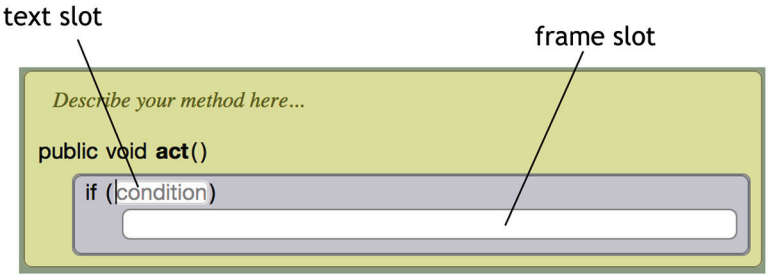


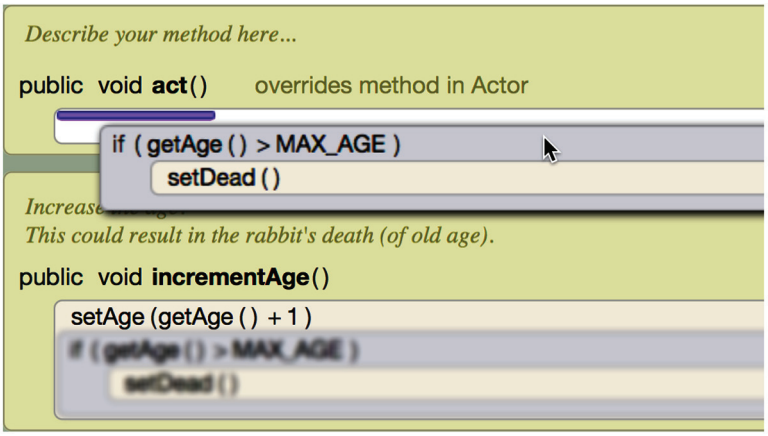
Figure 8. Frames may contain slots. Text slots are filled by entering text with the keyboard, while frame slots receive frames, either through key commands or mouse actions.



to a different (syntactically valid) location (Figure 9). A "cheat sheet" is available to facilitate recognition over recall for program statements, akin to the block palettes in block-based systems (Figure 7).

Overall, the system avoids some of the main drawbacks of both blocks and text, while combining several of their advantages. Many syntax errors common in text-based systems cannot be made anymore, indentation, layout and other program representation is improved and cannot be

Figure 9. Frames in the Stride editor are user interface elements: they can be dragged or selected, and they have a context menu with frame operations.



incorrect, statement syntax does not have to be memorized. At the same time, the edit viscosity of block-based systems is avoided, and programs of professional scale can be developed and remain readable. Entirely keyboard-based editing—faster than either block or text systems—is available, as well as mouse-based alternatives.

This system, with its combination of aspects from block-based and text-based systems, helps to facilitate the transition from one to the other by avoiding incidental overhead and separating the process into two steps: first to frames, and later to text—by then a purely syntactical exercise. Frame based editing, including this aspect of transition from blocks to text, is discussed in more detail in Kölling, Brown, & Altadmri (2015).

We are not alone in the belief that combining aspects of these two types of system can bring benefits. Mönig, Ohshima, & Maloney (2015), for example, are working on a new language called GP, which has a similar goal. This is interesting especially since these authors are designers of existing successful block-based languages (BYOB/Snap and Scratch among them), and are thus approaching the target from the other side: while our experience is with textual languages, and our work adds block-like aspects to those, their background is with blocks, which are now receiving some text-related features. It may be that we meet in the middle with systems that share some commonalities. The most interesting observation, though, is the joint belief that the future of environments supporting the transition lies not in dual systems that offer blocks and code, but in systems offering new functionality combining aspects of the two worlds in a single representation.

On the Obsoleteness of Text

Frame-based editing, should it become successful, will not replace block-based systems.

Block systems have their well deserved place as programming environments for young (i.e. primary school) novice learners. For that target group, they are—and will remain for the foreseeable future—the most appropriate system.

Text-based editing, however, is much less defensible. Frame-based programming is—in principle—superior to text-based programming in just about every aspect. When implemented well, it is more readable, quicker and easier to edit, leads to fewer syntax errors, and is easier to process and build good tools for, than text. In theory, although our current implementation is in an educational system, it could be implemented for standard programming languages in professional IDEs.

The replacement of text by frames—should it ever happen at all—will not be quick. The existing tool chain of software development tools, with its plethora of different software systems, currently relies on plain program text as a common interchange format. Any change must be necessarily slow. However, there is nothing in principle stopping us all from programming in frame-based systems in coming decades.

CONCLUSION

The popularity of educational programming environments has boomed in the last ten years. More systems have been published than ever before, and more are being developed. Programming as a subject is currently being strengthened or introduced (or re-introduced) at school level in many countries (Brown, Kölling, Crick, Peyton Jones, Humphreys, & Sentance, 2013), and this development is again increasing the interest in these types of system. Many environments are currently under development, and more will follow.

In designing these environments, there are many possible paths that can be taken and many design decisions to be made. In this paper, we have described our experiences with the design and support of three systems, Blue, BlueJ and Greenfoot, and attempted to extract some general design principles that are fundamental, system independent, and that can guide the development of other systems in future. We believe that awareness of these principles can improve the design of such systems.

In addition, we have highlighted one area of development that we think will be relevant to many teachers in the near future, and where significant improvements can be made: the combination of block-based and text-based programming modes. Frame-based editing, as implemented in the most recent Greenfoot system, is one possibility of how these two types of system can be combined.

ACKNOWLEDGMENT

Many people have contributed great amounts to the systems discussed in this paper. John Rosenberg was my PhD supervisor during the creation of the Blue project, and also closely involved in the creation of BlueJ. Without him, none of this would have happened. Many people have contributed to the implementation of BlueJ and Greenfoot, to varying degrees. The most significant contributions here are from Bruce Quig, Andrew Patterson, Davin McCall, Poul Henriksen, and Neil Brown, all of which are fantastic programmers. Ian Utting has made various contributions to BlueJ over many years. David Barnes is my co-author for the BlueJ book and with this contributed immensely to BlueJ's success; without him this book may never have been written. Finally, Sun Microsystems, Oracle Inc, and Google have supported the projects over the years; I am very grateful for their continued support for our projects in particular and to the computer science education community in general.

REFERENCES

Allen, E., Cartwright, R., & Stoler, B. (2002). DrJava: A lightweight pedagogic environment for Java. *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, 34(1) 137-141. doi:10.1145/563340.563395

Armoni, M., Meerbaum-Salant, O., & Ben-Ari, M. (2015). From scratch to "real" programming. *ACM Transactions on Computing Education*, *14*(4), 25. doi:10.1145/2677087

Astrachan, O., Bruce, K., Koffman, E., Kölling, M., & Reges, S. (2005). Resolved: Objects early has failed. *ACM SIGCSE Bulletin*, *37*(1), 451–452. doi:10.1145/1047124.1047359

Barnes, D., & Kölling, M. (2002). *Objects First with Java - A Practical Introduction using BlueJ*. Prentice-Hall.

Bau, D. (2015). Droplet, a blocks-based editor for text code. *Journal of Computing Sciences in Colleges*, *30*(6), 138–144.

Bau, D., Bau, D. A., Dawson, M., & Pickens, C. (2015). Pencil code: block code for a text world. *Proceedings of the 14th International Conference on Interaction Design and Children* (pp. 445-448). doi:10.1145/2771839.2771875

Begel, A., & Klopfer, E. (2007). StarLogo TNG: An introduction to game development. *Journal of E-Learning*.

Bergin, J., Pattis, R., Stehlik, M., & Roberts, J. (1997). *Karel*. Wiley.

Bergin, J., Stehlik, M., Roberts, J., & Pattis, R. (2005). *Karel J Robot: A gentle introduction to the art of object-oriented programming in Java*. Dream Songs Press.

Black, A. P., Bruce, K. B., Homer, M., Noble, J., Ruskin, A., & Yannow, R. (2013). Seeking Grace: a new object-oriented language for novices.*Proceeding of the 44th ACM technical symposium on Computer science education* (pp. 129-134). doi:10.1145/2445196.2445240

Boshernitsan, M., & Downes, M. (2004, December). *Visual Programming Languages: A Survey*. Technical report No. UCB/CSD-04-1368. Computer Science Division (EECS), University of California Berkeley.

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

Brown, N., & Kölling, M. (2013). A Tale of Three Sites: Resource and Knowledge Sharing Amongst Computer Science Educators. *Proceedings of theNinth Annual International Computing Education Research Conference (ICER)* (pp. 27-34). doi:10.1145/2493394.2493398

Brown, N. C. C., Kölling, M., Crick, T., Peyton Jones, S., Humphreys, S., & Sentance, S. (2013). Bringing computer science back into schools: lessons from the UK.*Proceeding of the 44th ACM technical symposium on Computer science education* (pp. 269-274).

Caspersen, M. E., & Christensen, H. B. (2000). Here, there and everywhere–on the recurring use of turtle graphics in CS1.*Proceedings of the Fourth Australasian Computing Education Conference* (ACE '00) (pp. 34-40). doi:10.1145/359369.359375

Cooper, S., Dann, W., & Pausch, R. (2003). Teaching objects-first in introductory computer science. *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education* (SIGCSE '03) (pp. 191-195). doi:10.1145/611892.611966

Dahl, O. J., Myhrhaug, B., & Nygaard, K. (1967). Simula 67 common base language.

Dorling, M., & White, D. (2015). Scratch: A way to logo and python.*Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 191-196). doi:10.1145/2676723.2677256

Fincher, S., Kölling, M., Utting, I., Brown, N. C. C., & Stevens, P. (2010). Repositories of Teaching Material and Communities of Use: Nifty Assignments and the Greenroom.*Proceedings of the Sixth international workshop on Computing education research* (pp. 182-196). doi:10.1145/1839594.1839613

Fraser, N. (2013). Blockly: A visual programming editor. *Google developers*. Retrieved from https://developers.google.com/blockly/

Goldberg, A., & Robson, D. (1983). *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc.

Good, J. (2011). Learners at the Wheel: Novice Programming Environments Come of Age. *International Journal of People-Oriented Programming*, *1*(1), 1–24. doi:10.4018/ijpop.2011010101

Goschnick, S., & Balbo, S. (2005). Game-first programming for information systems students. *Proceedings of the second Australasian conference on Interactive entertainment* (IE '05) (pp. 71-74). Creativity & Cognition Studios Press.

Gosling, J. (2000). *The Java language specification*. Addison-Wesley Professional.

Harvey, B., & Mönig, J. (2010). Bringing "no ceiling" to Scratch: Can one language serve kids and computer scientists. *Proc. of Constructionism* (pp. 1-10).

Harvey, B., & Mönig, J. (2015). Lambda in blocks languages: Lessons learned. *Proceedings of the IEEE-Blocks and Beyond Workshop (Blocks and Beyond)* (pp. 35-38). doi:10.1109/BLOCKS.2015.7368997

Henriksen, P., & Kölling, M. (2004). Greenfoot: combining object visualisation with interaction. *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (pp. 73-82).

Homer, M., & Noble, J. (2014). Combining tiled and textual views of code. *Proceedings of the2014 Second IEEE Working Conference on Software Visualization* (pp. 1-10). doi:10.1109/VISSOFT.2014.11

Hundhausen, C. D., Farley, S. F., & Brown, J. L. (2009). Can direct manipulation lower the barriers to computer programming and promote transfer of training?: An experimental study. *ACM Transactions on Computer-Human Interaction*, *16*(3), 13. doi:10.1145/1592440.1592442

Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., & Kay, A. (1997). Back to the future: The story of Squeak, a practical Smalltalk written in itself. *ACM SIGPLAN Notices*, *32*(10), 318–326. doi:10.1145/263700.263754

Kay, A. (2005). *Squeak Etoys Authoring & Media*. Viewpoints Research Institute.

Kölling, M. (1999a). *The Design of an Object-Oriented Environment and Language for Teaching* [Ph.D. thesis]. Basser Department of Computer Science, University of Sydney.

Kölling, M. (1999b). The Problem of Teaching Object-Oriented Programming, Part 1: Languages. *Journal of Object-Oriented Programming*, *11*(8), 8–15.

Kölling, M. (1999c). The Problem of Teaching Object-Oriented Programming, Part 2: Environments. *Journal of Object-Oriented Programming*, *11*(9), 6–12.

Kölling, M. (2010). The Greenfoot Programming Environment. *ACM Transactions on Computing Education (TOCE), 10(4)*,182-196.

Kölling, M., Brown, N., & Altadmri, A. (2015). Frame-Based Editing: Easing the Transition from Blocks to Text-Based Programming.*Proceedings of the 10th Workshop in Primary and Secondary Computing Education* (pp. 29-38). doi:10.1145/2818314.2818331

Kölling, M., Koch, B., & Rosenberg, J. (1995). Requirements for a First Year Object-Oriented Teaching Language.*Proceedings of the 26th SIGCSE Technical Symposium on Computer Science Education* (pp. 173-177). doi:10.1145/199688.199770

Kölling, M., Quig, B., Patterson, A., & Rosenberg, J. (2003). The BlueJ system and its pedagogy. *Computer Science Education*, *13*, 249–268. doi:10.1076/csed.13.4.249.17496

Kölling, M., & Rosenberg, J. (1996). An object-oriented program development environment for the first programming course. *ACM SIGCSE Bulletin*, *28*(1), 83–87. doi:10.1145/236462.236514

Kölling, M., & Rosenberg, J. (2000). Objects first with Java and BlueJ (seminar session). *ACM SIGCSE Bulletin*, *32*(1), 429. doi:10.1145/331795.331912

MacLaurin, M. (2009). Kodu: End-user programming and design for games.*Proceedings of the 4th International Conference on Foundations of Digital Games*. doi:10.1145/1536513.1536516

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. doi:10.1145/1868358.1868363

Meyer, B. (1988). Eiffel: A language and environment for software engineering. *Journal of Systems and Software*, *8*(3), 199–246. doi:10.1016/0164-1212(88)90022-2

Microsoft. (2016, February 14). Visual Studio Express. Retrieved from https://www.visualstudio.com/en-us/products/visual-studio-express-vs.aspx

Mönig, J., Ohshima, Y., & Maloney, J. (2015). Blocks at your fingertips: Blurring the line between blocks and text in GP. *Proceedings of theBlocks and Beyond Workshop (Blocks and Beyond)* (pp. 51-53). doi:10.1109/BLOCKS.2015.7369001

Overmars, M. (2004). Learning object-oriented design by creating games. *Potentials, IEEE*, *23*(5), 11–13. doi:10.1109/MP.2005.1368910

Papert, S. (1980). *Mindstorms: Children, Computers, and Powerful Ideas*. New York, NY, USA: Basic Books, Inc.

Patterson, A., Kölling, M., & Rosenberg, J. (2003). Introducing Unit Testing With BlueJ.*Proceedings of the 8th conference on Information Technology in Computer Science Education (ITiCSE 2003)* (pp. 11-15).

Pattis, R. E. (1981). *Karel the Robot: A gentle introduction to the art of programming*. New York, USA: John Wiley & Sons, Inc.

Pencil Code. (2016, February 14). Pencil Code. Retrieved from https://pencilcode.net

Powers, K., Ecott, S., & Hirshfield, L. M. (2007). Through the looking glass: Teaching CS0 with Alice. *ACM SIGCSE Bulletin*, *39*(1), 213–217. doi:10.1145/1227504.1227386

Price, T. W., & Barnes, T. (2015). Comparing Textual and Block Interfaces in a Novice Programming Environment.*Proceedings of the eleventh annual International Conference on International Computing Education Research* (pp. 91-99). doi:10.1145/2787622.2787712

Python Software Foundation. (2012). Python standard library: Turtle graphics for tk. Retrieved from http://docs.python.org/library/turtle.html

Reas, C., & Fry, B. (2003). Processing: a learning environment for creating interactive Web graphics. *ACM SIGGRAPH 2003 Web Graphics*.

Roque, R. V. (2007). *OpenBlocks: an extendable framework for graphical block programming systems* [Doctoral dissertation]. Massachusetts Institute of Technology.

Sanders, D., & Dorn, B. (2003). Jeroo: A tool for introducing object-oriented programming.*Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education* (pp. 201-204). doi:10.1145/611892.611968

Slack, J. M. (1990). *Turbo Pascal with turtle graphics*. St. Paul: West Publishing Co.

Storey, M. A., Damian, D., Michaud, J., Myers, D., Mindel, M., German, D., & Hargreaves, E. et al. (2003). Improving the usability of Eclipse for novice programmers.*Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange* (pp. 35-39). doi:10.1145/965660.965668

Stroustrup, B. (1986). *The C++ programming language*. Pearson Education India.

Sutherland, I. B. (1963) Sutherland, I. B. SKETCHPAD, a man-machine graphical communication system. *Proceedings of the Spring Joint Computer Conference* (pp. 329–346).

Weintrop, D., & Wilensky, U. (2015a). To block or not to block, that is the question: students' perceptions of blocks-based programming.*Proceedings of the 14th International Conference on Interaction Design and Children* (pp. 199-208). doi:10.1145/2771839.2771860

Weintrop, D., & Wilensky, U. (2015b). Using commutative assessments to compare conceptual understanding in blocks-based and text-based programs.*Proceedings of the Eleventh Annual International Conference on International Computing Education Research, ICER 15* (pp. 101-110). doi:10.1145/2787622.2787721

Wolber, D., Abelson, H., Spertus, E., & Looney, L. (2011). *App Inventor*. O'Reilly Media, Inc.

ENDNOTES

1 In Logo's case, this also involved physical "turtles"—robots in the real world, while later microworlds were often software simulation only.

2 Smalltalk was an exception: it provided a full integrated development environment. It had, however, failed to get traction in programming education and was rarely used for introductory teaching.

3 This functionality has been described in some detail elsewhere (Kölling, 1999a; Kölling, Quig, Patterson, & Rosenberg, 2003).

4 In 2001, BlueJ was downloaded just over 100,000 times; in the following years, growth was exponential for some time and this number doubled every two years, exceeding 2.5 million downloads per year by 2010.

5 Some earlier systems, such as Squeak (Ingalls, Kaehler, Maloney, Wallace, & Kay, 1997) and Etoys (Kay, 2005) were based on a similar approach, but failed to achieve the same impact as Scratch.

Michael Kölling is a Professor at the School of Computing, University of Kent, in Canterbury, UK. He holds a PhD in computer science from Sydney University, and has worked in Australia, Denmark and the UK. Michael's research interests are in the areas of object-oriented systems, programming languages, software tools, computing education and HCI. He has published numerous papers on object-orientation and computing education topics and is the author and co-author of two Java textbooks. Michael is the lead developer of BlueJ and Greenfoot, two educational programming environments. He is a UK National Teaching Fellow, Fellow of the UK Higher Education Academy, Oracle Java Champion, and a Distinguished Educator of the ACM. In 2013, he received the ACM SIGCSE Award for Outstanding Contribution to Computer Science Education. Michael is a founding member of 'Computing At School', a UK organisation furthering computing teaching at school level.

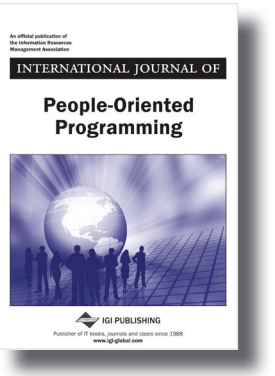
CALL FOR ARTICLES

International Journal of People-Oriented Programming

An official publication of the Information Resources Management Association

MISSION:

The primary mission of the **International Journal of People-Oriented Programming (IJPOP)** is to be instrumental in the improvement and development of the people-oriented programming, appealing to both academics and practitioners. It also educates a wider audience discussing the conceptualization, design, programming, configuration and orchestration of self-fashioned tools and products that ultimately suit the user's own unique needs and aspirations. The journal publishes original material of high quality concerned with the theory, concepts, techniques, methodologies and the tools that service a market-of-one—the empowered user.



ISSN 2156-1796
eISSN2156-1788
Published semi-annual

COVERAGE/MAJOR TOPICS:

- Activity theory and modeling
- Agent meta-models, mental models
- Alert filter and notification software, automated task assistance
- Augmented reality, augmented interaction
- Automating personal ontologies, personalised content generation
- Client-side conceptual modeling
- Computational models from psychology
- Context-aware systems, location-aware computing, ubiquitous computing
- Cultural probes, self-ethnography
- End-user composition, end-user multi-agent systems
- Game development support tools
- Game mods, game engines, open game engines
- Home network applications
- Human-centered software development
- Interface generators, XML-based UI notation generators
- Interface metaphors
- Life logs, life blogs, feed aggregators
- Mashups, mashup tools, cloud mashups
- Model-driven design, didactic models, model-based design and implementation
- New generation visual programming
- People-Oriented Programming (POP)
- People-Oriented Programming case studies
- Personal interaction styles, touch and gestures
- Personal ontologies and taxonomies
- Personalisation, individualisation, market of one
- Personalized learning
- Personas and actors
- Real-time narrative generation engines
- Role-based modeling
- Service science for individuals
- Situated computation, social proximity applications
- Smart-phone mashups, home network mashups, home media mashups
- Software analysis & design, software process modeling
- Software component selection
- Speech and natural language interfaces
- Storyboarding, scenarios, picture scenarios
- Task flow diagrams, Task-based design
- Task models, task analysis, cognitive task models, concurrent task modeling
- Use case models, user interface XML notations
- User interface tools, XML-based UI notations
- User modelling, end user programming, end user development
- User-centered design, usage-centered design
- Wearable Computing
- Wearable computing, bodyware
- Web-service orchestration, web-service co-ordination

All inquiries regarding IJPOP should be directed to the attention of:
Steve Goschnick, Editor-in-Chief • IJPOP@igi-global.com
All manuscript submissions to IJPOP should be sent through the online submission system:
http://www.igi-global.com/authorseditors/titlesubmission/newproject.aspx

Ideas for Special Theme Issues may be submitted to the Editor-in-Chief.

Please recommend this publication to your librarian. For a convenient easy-to-use library recommendation form, please visit:
http://www.igi-global.com/IJPOP