An Iterative Development Pattern to Learn Notional Machine and Programming Practices

BOISVERT, Charles <http://orcid.org/0000-0002-3069-5726>

An Iterative Development Pattern to Learn Notional Machine and Programming Practices

Charles Boisvert
c.boisvert@shu.ac.uk
Department of Computing
Sheffield Hallam University

Abstract

This paper proposes to use iterative development — the use of incomplete or malfunctioning programs as part of problem resolution. It analyses how this pattern can applies in three Scratch exercises (a simple game, another in three dimensions, and a crank-rod mechanical simulation) that are well suited to present as iterative development problems, and shows how this presentation helps encourage learners to develop better programming practices and improve their notional machine — their mental model of code interpretation.

Keywords: Iterative Development; Scratch; educational resources; Notional Machine

1. Learning to Code

Teaching computer programming as early as primary school is much facilitated today by many tools. In particular, the representation of code through the metaphor of puzzle pieces, which can only combine in syntactically valid ways, has given rise to many systems (Resnick et al., 2009; Harvey & Mönig, 2010; Google, 2019) in use today, as they remove syntactic difficulties from computer programming and support the reification of key concepts through virtual or sometimes tangible representations (Resnick et al., 2009).

But this progress has many limits. The review conducted by Lodi et al. (2019) shows the need to develop programming education in two directions: by extending the perception of the discipline, from code production to associated problem-solving; and by developing the learner's notional machine, the mental model which makes it possible to follow the execution of a program (Du Boulay, 1986). For example, efforts by Sheffield Hallam University to support local schools (Adshead et al., 2015) waned after a few years, with primary and secondary teachers becoming satisfied with a small set of activities they master.

It is clear that the teachers have taken up the challenge of training and quickly offering this new discipline, but the first answers cannot be enough: support must continue over time. This article proposes, to develop learning at school, to rely on some key concepts of software engineering, and offers a way to translate them into teaching.

2. Code is written Step by Step

The practice of developing programs in stages, checking each one before modifying the code, is well known in software engineering. As Lodi et al. (2019) notes, we find these practices in agile methodologies (Beck et al., n.d.) common among development professionals; this use of multiple incremental executable results has become central in the computer approach to problems.

2.1. Refactoring

For example, the French textbook of Daviaud & Revranche (2019) use the technique shown in fig. 1 to explain loops.

By proposing two programs with identical results, first without then with a loop, the authors do not only show the advantage of the second. They use a software engineering technique, refactoring, which consists in rewriting an already functional program to make it easier to handle, without changing the result.

This technique allows developers to rely for most of their project on executable intermediate results,

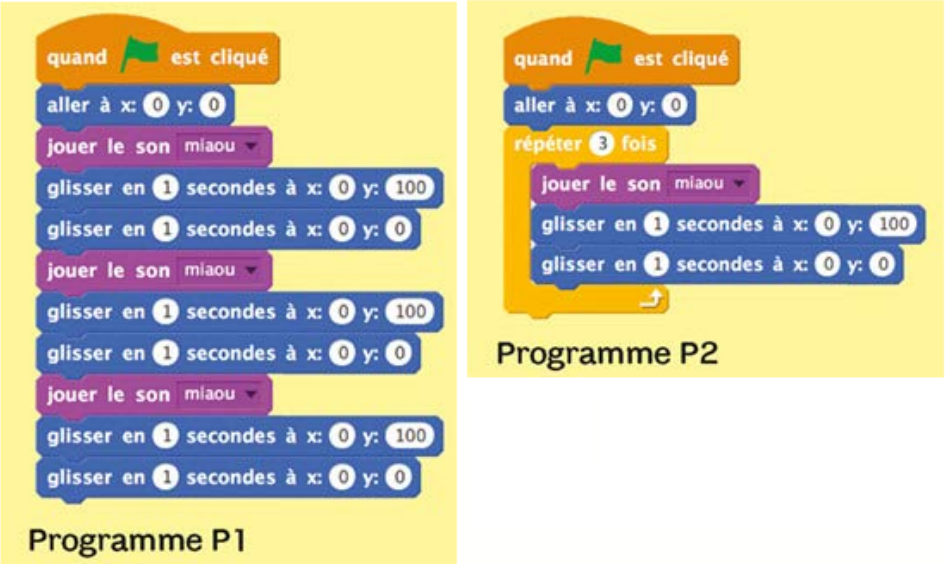

Figure 1 – Rewriting a program: two scripts with identical effects, one long, the other shortened by a loop. Source: Daviaud & Revranche (2019)

which can be evaluated by comparison with an explicit 'definition of done' and which form part of the problem solving process.

2.2. Iterative Development: definition

We could consider that the development of programs by trial and error would have something *natural*, and thus will be a *good* in our teaching methods. But this first idea is not enough: the refactoring example of section 2.1 above, is not motivated by an error — at least not by errors in code or execution.

To better understand how this notion is adapted in a learning situation, a clearer and more restricted definition of what makes iterative development is proposed in Boisvert (2009). These are problems where:

- The solution is not trivial;
- Partial solutions are productive, that is, they support progress towards a more complete solution;
- Mistakes are also productive.

The original article proposes draws examples from well known higher education computer science exercises, such as sorting, displaying a calendar, or string processing. Primary and secondary teachers need examples of interactive and visual creations, more motivating for young students, and more adapted to environments like Scratch.

2.3. Deconstructing our Scratch Scripts

Many Scratch textbooks examples provide under-use this benefit of step-by-step programming. For example, figure 2 shows, a script taken from a game offered by Vorderman (2019). This is a fairly common script, which allows the player to control a sprite's movements to the right or left with the keyboard.

The script simultaneously uses simple visual effects and a number of difficult notions: selections, events, loop, initialization. But above all it is made to use in a single piece. The learner who would like to see the effect of part of the work would have to carefully choose which section to ignore to check a result. A copy error is also more difficult to spot to correct it. Finally, if we wanted to adapt the script, for example to move the sprite up and down, this starting point encourages creating few scripts made of many blocks: a phenomenon of 'spaghetti code' well known to professionals. For being more colorful, Scratch's spaghetti so less unreadable than any other language.

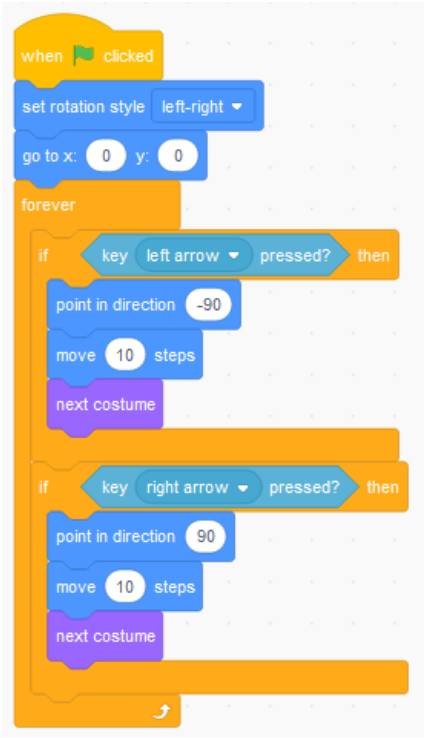

Figure 2 – A complicated script for a simple movement. Reproduced from Vorderman (2019)

An alternative to the example above is instead to refactor the code by separating the intended results into short scripts with each a clear objective. This exploded version is presented figure 3. The refactored version lets you run each script without having to assemble the others. It is also easier to identify a copy or design error. Finally, the series of scripts is easier to extend or modify.

Figure 3 – Le même résultat que figure 2 en traitant chaque objectif (initialisation, déplacement dans chaque direction, animation du mouvement) par un script séparé.

One might worry that this second version calls for complex notions of parallelism and events. The examples in this article indeed use both. However, Scratch makes it easier to understand, by reifying these concepts. These notions are more complex in a larger context — where the programming language, the diversity of systems, the persistence of legacy approaches, the need to guarantee security and stability, impose a detailed and sharp understanding of the concepts, and a demanding practice which often confuses the code.

Mitch Resnick recalls that with his team, while designing Scratch, they were ‘afraid of anything that

might scare eight-year-olds' [1]. The Scratch team looked to represent parallelism or events in such a way that their use did not involve the abstract notions needed to understand them in detail. Therefore, we can propose these notions in practice: in the same way that we do not hesitate to throw a ball at children, even if they are not yet able to grasp the laws of gravitation in full detail, we should not be afraid of giving them two scripts that work at the same time, even if they are not yet able to grasp parallel execution in full detail. Perhaps on the contrary, just as the child who has played ball will one day be able to use this experience to better conceptualize the laws of physics, the one who has seen Scratch execute two scripts simultaneously will be able to rely on this experience to integrate the concepts of parallelism.

To better understand this practice, let us consider more complete examples chosen to associate a problem to resolve and the use of imperfect code as a means of resolution.

3. Three Iterative Problems in Scratch

Many examples in Scratch can be adapted for iterative development. The three cases below illustrate the use that can be made of Scratch functionalities to present a development whose stages are productive and testable.

3.1. Lunar Lander

Landing a rocket is well known as a game and a programming exercise. The version shown here is from a collection of resources created by Quinson (2014). The problem in Scratch, and a solution which makes use of parallelism are illustrated in fig. 4.

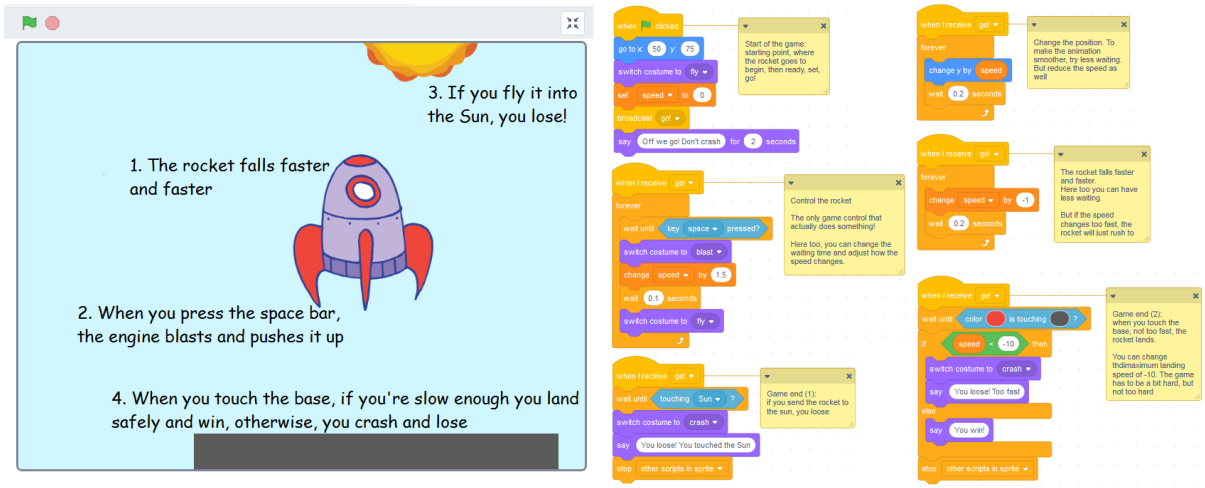

Figure 4 – Lunar lander: the problem (left) and a solution using Scratch parallelism (right)

We notice again several techniques that help in the development and understanding of the program:

- Several short scripts rather than one long one. Each script can be written and tested separately, allowing to check and reflect on partial results: *partial solutions are productive.*
- Separation of concerns. Each script performs only one function in the game (from top to bottom, fig. 4: start of game, rocket movement, acceleration of its fall, player control, and two end conditions).
- A design pattern. The initialization of the game is entrusted to a unique script which establishes initial conditions and then invokes the core of the game.

Too many examples propose, as in the case of figure 2 (Vorderman, 2019), to copy an entire long script before seeing a result. This approach makes it possible instead to introduce the concepts one by one and to check their operation in stages. The full exercise (Quinson, 2014) introduces each stage of

[1] Oral communication: welcome speech, Scratch Conference 2017, Bordeaux

development and emphasizes the ability to run the incomplete program and customize the game as you go. The approach easily extends to many games and animations on Scratch.

Testing a script in Scratch can be an opportunity to practice a software engineering technique. Even for young learners, we can find:

- *A clear definition of 'done'*: for example, the learner should be able to express in advance that 'if it works, the rocket will ... '
- *Multiple tests*: A single test is usually not enough; can the rocket go up and down? What if it goes faster?
- *Considering borderline cases*: what does the rocket do at the start of the animation, at the end?

3.2. A Little 3D

The second example is taken from resources developed for secondary education at the University of Sheffield Hallam (Adshead et al., 2015). One of them is a series of games all of the same type: the player sees objects arrive towards him (enemies to be shot down, obstacles, elements of scenery). Their movement and perspective create the illusion the player is moving through the game, as seen fig. 5.

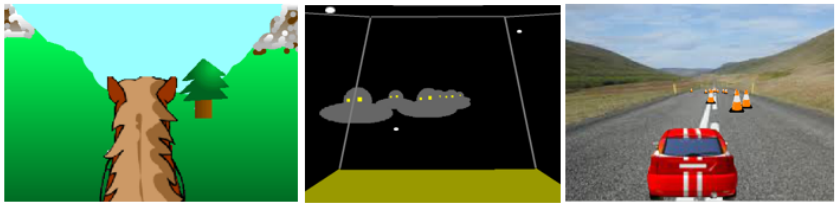

Figure 5 – A series of games using perspective

Young learners cannot master the necessary calculations in this example. The educational material [2] therefore offers a ready-made block, shown figure 6.

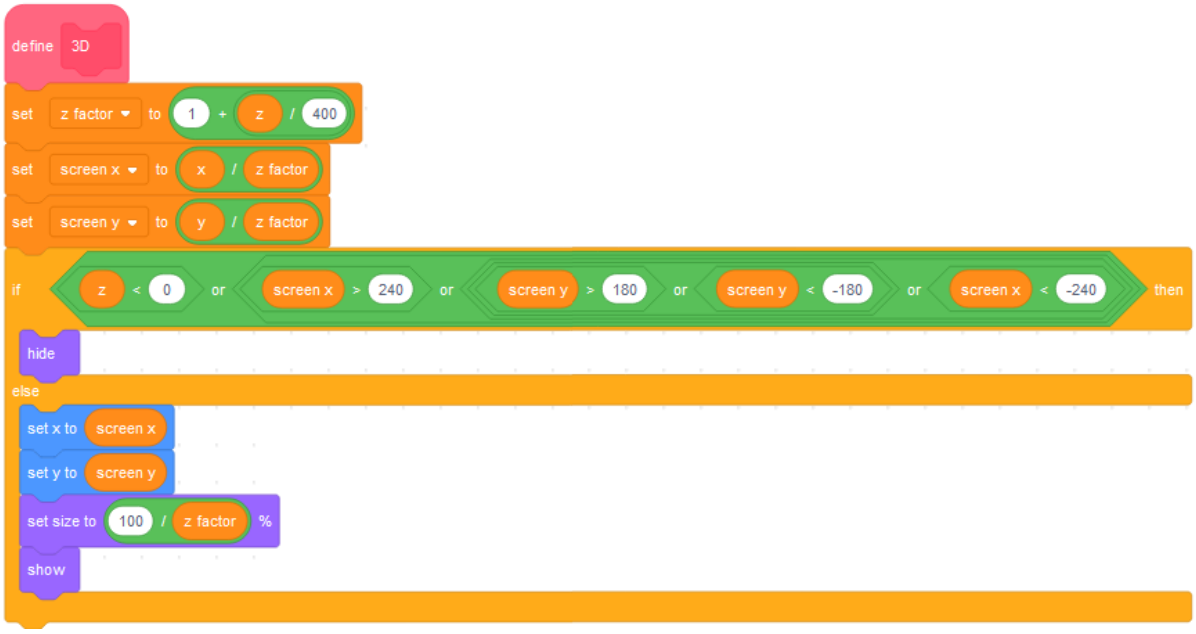

Figure 6 – Calculation of X and Y co-ordinates as well as the apparent size of an object of screen, based on spatial position variables x, y, z

[2] https://tinyURL.com/3DScratch

The block projects the three spatial coordinates into two apparent coordinates, and explains its use without entering into the mathematical details.

Apart from this calculation block, the previous techniques, and in particular the parallelism of Scratch, make it possible to define distinct and separately testable problems: the movement of objects towards the player, the movements of the player to reach or avoid them, the management of collisions, each form separate functions and scripts.

The material also offers learners to take control of their game, in several ways:

- *Costumes in advance*: several sprites and backgrounds are available, giving different themes (car racing, space, horseback riding). The objective is twofold: to prevent learners from spending too long editing the graphics, and to give options from the start. The costumes and the 3D calculation block are proposed in a Scratch project to modify.
- *Offer game options*: alternative solutions during development. Many exercises propose a complete project before inviting the learner to modify it. The progression and the execution tests during development allow us to offer variants while the game is not over.
- *Choosing the right block*: the choice of what a ready-made block does requires a lot of care. This is to facilitate an understanding of the problem by circumscribing an operation and naming it. This choice limits and directs the possibilities of learners, as much as it facilitates certain creations.

3.3. A Crank-Rod Mecanism [3]

This third example is not a game but an animation, showing the movement of a rod-crank mechanism. The result is shown in figure 7.

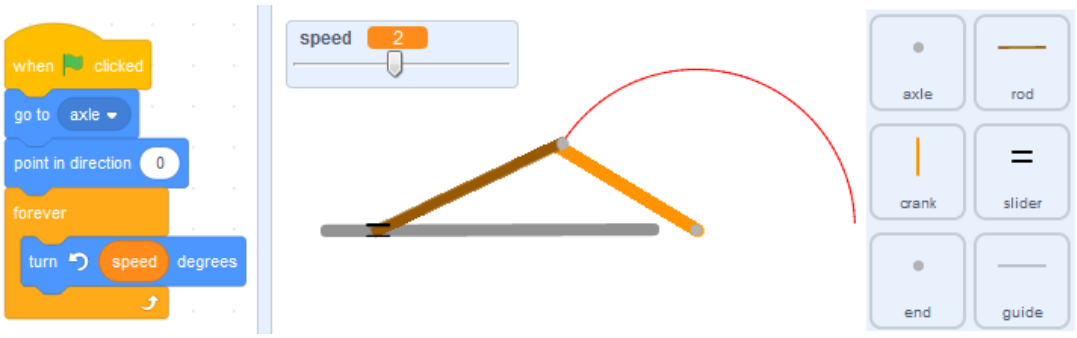

Figure 7 – Rod-crank mechanism. The decomposition lets the movement be processed by coordinating a series of very simple scripts, like the crank (left)

Like the previous two, the problem is based on short scripts that make use of parallelism, and each subproblem is handled by a testable script independently of the others. However, parallel execution is not as visible: the movement of each sprite is controlled by a single script. The peculiarity of the exercise is the very clear match between the parts of the mechanism and the processing of each element of the movement. Six sprites (on the right figure 7) model the mechanism. Two of them (the axle and the horizontal guide) are fixed. The movement of the other four is simple: the crank and its end rotate around the axle; the connecting rod follows the end of the crank and remains turned towards the slider; finally the slider is maintained at an equal distance by the connecting rod.

Positioning an object which revolves around another requires trigonometry, for which a block is provided in advance, indicated in figure 8. This block uses another abstract notion which can worry a teacher but which, in practice, does not bother the learners: the name of the block which serves as a reference point is given in parameter [4].

[3] https://scratch.mit.edu/projects/538088937

[4] Scratch interprets this parameter correctly, but it translates the block incompletely when we change the interface language

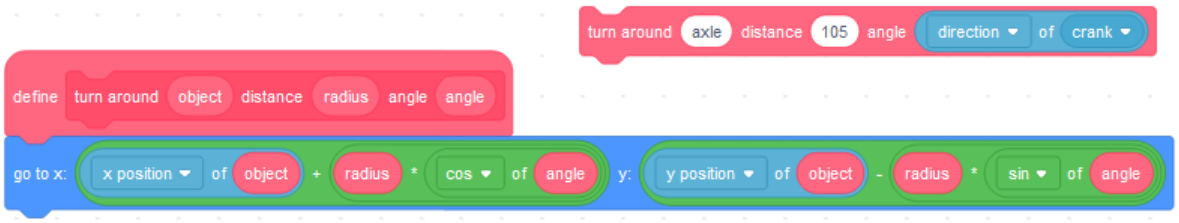

Figure 8 – A block to position a sprite around a reference object which is named in a parameter.

With younger learners, it is best to provide the sprites forming the mechanism in advance. The exercise then consists in finding each movement. As with the previous examples, we can start by defining the expected result, in order to test the scripts, as they are developed, by comparing them to the expected behaviour.

Looking for a solution leads through all kinds of results that are incorrect, but useful for better understanding both the problem and the code, making mistaken code a tool in problem-solving. Seymour Papert illustrates this point beautifully in his work on Logo (Papert, 1980), reproduced in figure 9: the first attempts are wrong, but seeing the result gives food for thought and a child might not find the result without these 'errors' which are also so many indications.

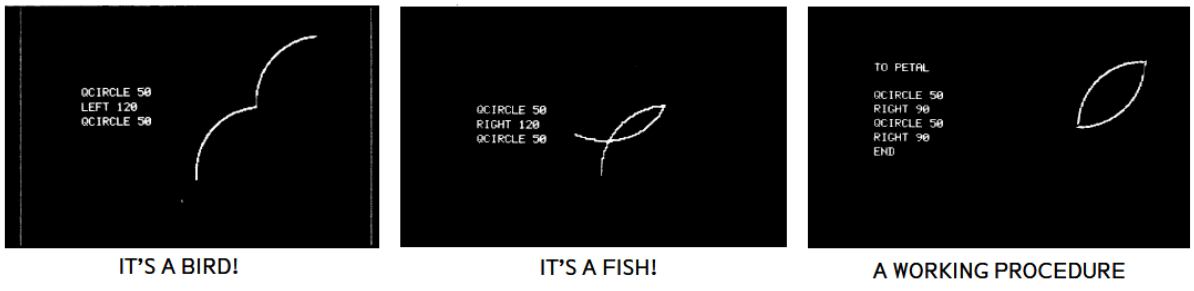

Figure 9 – "It's a bird!", "It's a fish!": debugging according to Papert (1980)

The movement of the slider is the most difficult. One solution is to compare the distance from the slider to the crank end to position it: this option has the advantage of matching a mental model of the mechanism: the connecting rod pushes or pulls the slider.

Figure 10 presents a second option, which can be given ready-made. The solution starts from the same principle, but calculates the error in the distance to better position the slider.

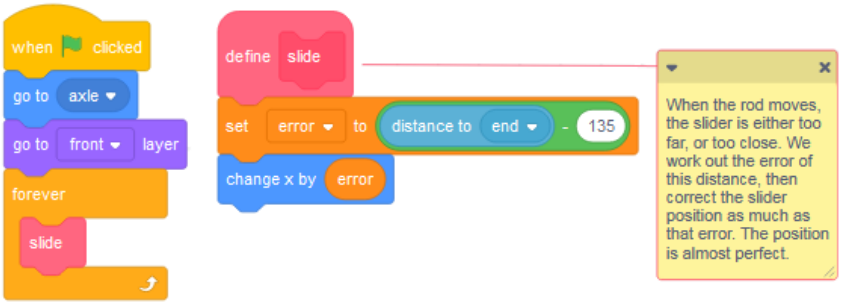

Figure 10 – One way to calculate the approximate position of the end of the rod.

.

For older learners, this exercise can be an opportunity to decompose a problem. The clearly visible mechanism supports the learners in their search for a solution, but the resolution requires a time of

reflection before coding, as recommended by Chevalier et al. (2020). In particular, a good choice of the center of each sprite simplifies the programming of its movement, and the complete solution of the movement of the slide requires mathematical knowledge (the Pythagorean theorem).

4. Conclusion: Notional Machine and Programming Practices

In their review, Lodi et al. (2019) argue that the notional machine formed by many young learners is largely simplified, often relying on on a personification of the interpreter represented by an animated sprite. For the author, educators and learners need support on this point, as well as support in applying software and problem-solving practices. Rather than new technological tools, the examples proposed here attempt to respond with resources that can be used within the existing technological offer. Indeed, the educational use has as much, and sometimes more importance than the tool (Hundhausen et al., 2002).

In the case of Scratch, spaghetti code is a common underuse of the system. The examples proposed here use parallelism to both help build a more complete notional machine, to introduce a simple form of decomposition, and to dissociate the behaviors (modeled by scripts) from the agents (modeled by sprites). At the same time, they let learners experience certain established software development practices: development driven by testing, separation of concerns, and analysis to decompose the problem to address.

References

Adshead, D., Boisvert, C., Love, D., & Spencer, P. (2015). Changing culture: Educating the next computer scientists. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education* (pp. 33–38). ACM.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunnigham, W., & Fowler, M. (n.d.). *Manifesto for Agile Software Development.* Retrieved 2021-04-24, from `https://agilemanifesto.org/`

Boisvert, C. R. (2009, July). A visualisation tool for the programming process. In *Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education* (pp. 328–332). New York, NY, USA: Association for Computing Machinery. Retrieved 2021-04-12, from `https://doi.org/10.1145/1562877.1562976` doi: 10.1145/1562877.1562976

Chevalier, M., Giang, C., Piatti, A., & Mondada, F. (2020, August). Fostering computational thinking through educational robotics: a model for creative computational problem solving. *International Journal of STEM Education*, *7*(1), 39. Retrieved 2021-05-15, from `https://doi.org/10.1186/s40594-020-00238-z` doi: 10.1186/s40594-020-00238-z

Daviaud, D., & Revranche, B. (2019). *Mini Chouette Programmer avec Scratch 5e/4e/3e: cahier de soutien en maths*. Hatier.

Du Boulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research*, *2*(1), 57–73. (Publisher: SAGE Publications Sage CA: Los Angeles, CA)

Google. (2019). *Google blockly.* Retrieved from `https://developers.google.com/blockly`

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

Hundhausen, C. D., Douglas, S. A., & Stasko, J. T. (2002, June). A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages & Computing*, *13*(3), 259–290. Retrieved 2021-02-06, from `https://www.sciencedirect.com/science/article/pii/S1045926X02902375` doi: 10.1006/jvlc.2002.0237

Lodi, M., Malchiodi, D., Monga, M., Morpurgo, A., & Spieler, B. (2019, July). Constructionist Attempts at Supporting the Learning of Computer Programming: A Survey. *Olympiads in Informatics*, *13*, 99–121. Retrieved 2021-04-10, from https://ioinformatics.org/page/ioi-journal-index/44#volume13 doi: 10.15388/ioi.2019.07

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. Basic Books, Inc.

Quinson, M. (2014, May). *quatre jeux · coding4kids.* Retrieved 2021-04-24, from https://github.com/mquinson/coding4kids

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., ... others (2009). Scratch: programming for all. *Communications of the ACM*, *52*(11), 60–67. (Publisher: ACM)

Vorderman, C. (2019). *Computer Coding for Kids: A unique step-by-step visual guide, from binary code to building games*. DK Children.