Learning to program in a constructionist way

Mattia Monga, Michael Lodi, Dario Malchiodi, Anna Morpurgo, Bernadette Spieler
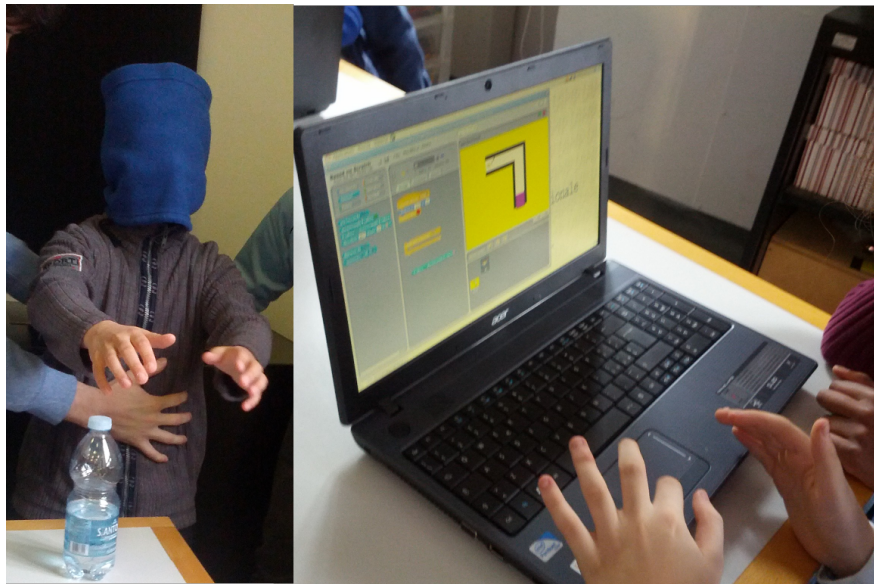
Learning to program in a constructionist way

Constructionism Working Group 6

Michael Lodi*
Alma Mater Studiorum -
Università di Bologna
Italy
michael.lodi@unibo.it

Dario Malchiodi
Mattia Monga
Anna Morpurgo
dario.malchiodi@unimi.it
mattia.monga@unimi.it
anna.morpurgo@unimi.it
Università degli Studi di Milano
Italy

Bernadette Spieler
Technische Universität Graz
Austria
bernadette.spieler@ist.tugraz.at

ABSTRACT

Although programming is often seen as a key element of constructionist approaches, the research on learning to program through a constructionist strategy is somewhat limited, mostly focusing on how to bring the abstract and formal nature of programming languages into "concrete" or even tangible objects, graspable even by children with limited abstraction power. However, in order to enable constructionism in programming several challenges must be addressed. One of the crucial difficulties for novice programmers is to understand the complex relationship between the program itself (the text of the code) and the actions that take place when the program is run by the interpreter. A good command of the notional machine is a necessary condition to build programming skills, as is recognizing how a relatively low number of abstract patterns can be applied to a potentially infinite spectrum of specific situations.Programming languages and environments can either help or distract novices, thus the choice is not neutral and their characteristics should be analyzed carefully to foster a good learning context. The mastery of the notional machine, however, is just the beginning of the game: to develop a real competence one must be able to think about problems in a way suitable to automatic elaboration; to devise, analyse, and compare solutions, being able to adapt them to unexpected hurdles and needs. Moreover, it is important to learn to work productively in a team, in an "organized" way: agile methods seem based on common philosophical grounds with constructionism.

*Also with INRIA Focus, Italy.

1 INTRODUCTION

Constructionism [Papert and Harel 1991] is a strategy of education which has its roots in Piaget's constructivist theory of learning as an active process, in which people actively *construct* knowledge from their personal experience of the world. In general, students

do not just receive pre-built ideas from teachers: they have to make them up by engaging themselves with problems, projects, and other people (instructors, but also peers). Papert's constructionism indeed emphasizes the importance of having personally-meaningful goals and *"public artifacts"* (not necessarily concrete ones: either "a sand castle on the beach or a theory of the universe" [Papert and Harel 1991]) that can be shared and discussed with others interested in the same (learning) enterprise [Resnick 1996]. This is sometimes summarized with the four P-words: Projects, Peers, Passion, Play and this motto indeed inspired successful educational initiatives such as the Scratch programming language [Resnick 2014].

1.1 Constructionism and programming

However, while programming is often seen as a key element of constructionist approaches (starting from Papert's LOGO, a programming language designed to enable the learning of geometry), the research on learning to program through a constructionist strategy is somewhat limited, mostly focusing on how to bring the abstract and formal nature of programming languages into "concrete" or even tangible objects, graspable even by children with limited abstraction power [Hauswirth et al. 2017; Horn and Jacob 2007; Resnick et al. 2009]. Notwithstanding this, constructionist ideas are floating around mainstream programming practice and they are even codified in some software engineering approaches: agile methods like eXtreme Programming [Beck and Andres 2004], for example, suggest several techniques that can be easily connected to the constructionist word of advice about discussing, sharing, and productively collaborating to successfully build knowledge together [Resnick 1996]; moreover the incremental and iterative process of creative thinking and learning [Resnick 2007] fits well with the agile preference to "responding to change over following a plan" [Beck et al. 2001].

The iterative process described by Resnick in [Resnick 2007] originated by observing how kindergarteners learn, and is now called "creative learning spiral" (Figure 1), that describes MIT's view on how to learn creatively [Resnick 2017]. When you learn by creating something (e.g. a computer program) you **imagine** what you want to do, **create** a project based on this idea, **play** with your creation, **share** your idea and your creation with others, **reflect** on the experience and feedback received from others, and all this leads you to **imagine** new ideas, new functionalities, new improvements for your project, or new projects. The process is iterated many times.

This spiral describes an iterative process, highly overlapping with iterative software development cycle (see 5.1).

2 WHAT DOES IT MEAN TO LEARN PROGRAMMING?

The basic assumption/premise behind programming — *i.e.,* producing a precise description of how to carry out a task or to solve a problem — is that an *interpreter*, different from the producer of the description, can understand it and effectively carry out the task as described. There are thus two distinct but tightly tied aspects in programming:
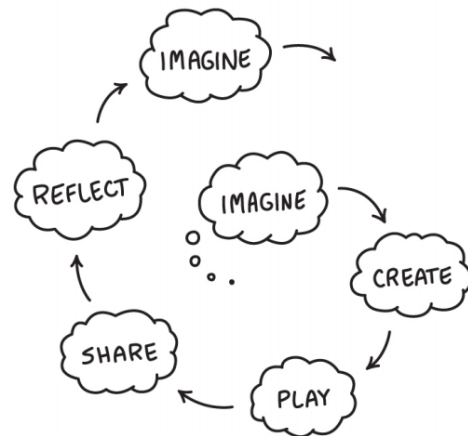
(1) the program itself (the text of the code),

Figure 1: Creative learning spiral [Resnick 2017]

(2) the actions that take place when the program is run by the interpreter.

We thus need to know the interpreter in order to program, in particular we need to know:

- the set of basic actions it is able to perform,
- a language it is able to understand, with rules on how to compose basic actions,
- the relation between *syntax* and *semantics,* that is what actions it will perform given a description, and, conversely, how to describe a given sequence of actions so that it will perform them.

The first aspect, that is the program *source code,* is explicit, visible. The second one instead, that is the actions that take place when the program is run, is somewhat implicit, hidden in the execution time world, and not so immediate to grasp for novices. Moreover, this aspect is sometimes underestimated by both teachers and learners: teachers, as experts, give it for granted; learners tend to construct personal intuitive, not necessarily coherent, ideas of what will happen.

This dichotomy of programming — its static visible code and its implicit dynamics — emerges as a critical issue when learning to program, as shown by studies from different perspectives. To cite a few [Sorva 2013]:

- Phenomenography studies show how novice programmers tend to perceive programming as no more than the production of code, missing to relating instructions in the program to what happens when the program is executed.
- Studies on programming misconceptions point out how most of programming misconceptions have to do with aspects that are not readily visible in the code but are related to the execution time, both in term of what will happen and of what will not unless explicitly specified in the code.
- Threshold concept theory identifies program dynamics as a candidate threshold concept in programming as it has many of the features that characterise threshold concepts; among others: it is a troublesome barrier to student understanding,

it transforms how the student perceives the subject, it marks a boundary between programmers and end users.

To help novice programmers take into account also the dynamic side of programming, the concept of *notional machine* has been proposed. A notional machine is a characterisation of the computer in its role as executor of programs in a particular language (or set of languages, or even a subset of a language) for didactic purposes. It thus gives a convenient description of the association syntax-semantics. The following learning outcomes should therefore be considered when teaching to program:

- the development by students of a perception of programming that does not reduce to production of code, but includes relating instructions to what will happen when the program is executed, and eventually comes to include producing applications for use and seeing it as a way to solve problems;
- the development of a mental model of a notional machine that allows them to make the association (static) syntax - (dynamic) semantics and to trace program execution correctly and coherently.

In particular this latter outcome goal will include the development of the following skills:

- given a program (typically one's own) and an observed behaviour:
  - identify when debugging is needed,
  - identify where a bug has occurred,
  - be able to correct the code;
- given a program and its specifications, be able to test it;
- understand that there can be multiple correct ways to program a solution.

If this is a crucial point in learning to write executable descriptions, however, programming is indeed a multifaceted competence, and the knowledge to construct and the skills to develop span over several dimensions, besides predicting concrete semantics of abstract descriptions given via programming languages:

- understanding general properties of automatic interpreters able to manipulate digital information;
- thinking about problems in a way suitable to automatic elaboration;
- devising, analysing, comparing solutions;
- adapting solutions to emerging hurdles and needs;
- organizing team work and productively eliciting, organizing, and sharing the abstract knowledge related to a software project.

2.1 Constructionism and learning to program

In some sense programming is intrinsically constructionist as it always involves the production of an artifact that can be shown and shared. Of course when teaching programming, this aspect can be stressed or attenuated.

Failure rates and dropout percentages in traditional programming courses and the urge to introduce programming early in school curricula have fostered new approaches to teaching programming, where this aspect has gained importance. Indeed the following points are given particular consideration:

- *motivation:* programming tasks should be engaging to keep pupils' motivation high;

- *syntax:* novices should be introduced first to the logical aspects of programming and only at a later stage to the syntax;
- *a constructivist approach:* the construction of knowledge is to be fostered, for example through *unplugged* activities that are more suitable to group work and shared meta-cognition;
- *constructionism:* the production of personal projects and artifacts must be encouraged.

In this perspective, for educational purposes visual programming languages (Section 3), have been developed and unplugged activities have been designed. In particular visual programming languages allow novices to concentrate on the logical aspects of programming without having to strive with unnatural textual syntactic rules. Moreover, they make it possible to realise small but meaningful projects, keeping students motivated, and support a constructionist approach where students are encouraged to develop and share their projects — video games, animated stories, or simulations of simple real world phenomena.

2.2 Computers Unplugged

Offline or unplugged programming activities were often used to explain important concepts or vocabulary to students without actually using a PC, laptop, or smartphone, e.g., x/y coordinates, the need for precise instructions for computers/robots, or variables and lists. Examples are to program a classmate like a robot, paint instructions, pack a rucksack, or send "broadcast messages" to colleagues.

Unplugged activities in small groups have become popular over the years to introduce basic computer science concepts in non vocational contexts. They offer a number of advantages:

inexpensive set up: they usually require very basic and inexpensive materials, so they can be easily proposed in different contexts;

no technological hurdles: they do not involve the use of technology, with which not all teachers are at ease;

a constructivist environment: indeed
- by manipulating real objects or dramatising processes, pupils can observe what happens, formulate hypotheses, validate them through experiments, i.e. develop a scientific approach to the construction of their knowledge;
- by working in group, pupils are encouraged to participate, share ideas, verbalise and uphold their deductions.

As the results of the activities, be they the execution of a procedure, the design of a solution or the construction of an object, are always shared in the class, unplugged activities also have a constructionist flavour and can be the first phase of a more structured constructionist proposal.

The following two examples, taken from CSUnplugged[1] and ALaDDIn[2], illustrate typical unplugged approaches to introduce children to programming.

In CSUnplugged "Rescue Mission", pupils are given by the teacher a very simple language with only three commands: 1 step forward, 90 degrees left, 90 degrees right. The task is to compose a sequence of instructions to move a robot from one given cell on a grid to a given other cell. Pupils are divided into groups of three where each one has a role: either programmer, bot, or tester. This division

---

[1] https://csunplugged.org/
[2] http://aladdin.di.unimi.it/

of roles is done to emphasise the fact that programs cannot be adjusted on the fly; they must be first planned, then implemented, then tested and debugged until they work correctly. So the programmer must write down the instructions for the task, then pass them to the tester, who will pass them on to the bot and will observe what happens; its role is to underline what doesn't work and hand them back to the programmer, who can then find the bug and fix it.

ALaDDIn "Algomotricity and Mazes" is an activity designed according to a strategy called algomotricity [Bellettini et al. 2012, 2013, 2014; Lonati et al. 2011], where pupils are exposed to an informatic concept/process by playful activities which involve a mix of tangible and abstract object manipulations; they can investigate it firsthand, make hypotheses that can then be tested in a guided context during the activity, and eventually construct viable mental models. Algomotricity starts "unplugged" [Bell et al. 2012] and ends with a computer-based phase to close the loop with pupils' previous acquaintance with applications [Taub et al. 2012].

"Algomotricity and Mazes" focuses on primitives and control structures. The task is that of verbally guiding a "human robot"(a blindfolded person) through a simple path. Working in groups, initially pupils are allowed to freely interact with the "robot", then they are requested to propose a very limited set of primitives to be written each on a sticky note, and to compose them into a program to be executed by the "robot". Also, they have the possibility of exploiting three basic control structures besides sequence (if, repeat-until, repeat-n-times). Groups may try their solutions as they wish and, when they are ready, each group is asked to execute its own program. Then the conductor may decide to swap some programs, so that a program is executed by the "robot" of another group. This allows the instructor to emphasise the ambiguity of some instructions or the dependency of programs on special features of the "robot" (e.g., step/foot size). In the last phase, students are given computers and a slightly modified version of Scratch. They are requested to write programs that guide a sprite through mazes of increasing complexity and foster the use of loop control structures[3].

2.3 Notional machines

Alan Perlis, in his foreword to "Structure and Interpretation of Computer Programs" (SICP) [Abelson et al. 1996], sets the stage for learning to program: *"Our traffic with the subject matter of this book involves us with three foci of phenomena: the human mind, collections of computer programs, and the computer. Every computer program is a model, hatched in the mind, of a real or mental process."* And then: *"The source of the exhilaration associated with computer programming is the continual unfolding within the mind and on the computer of mechanisms expressed as programs and the explosion of perception they generate. If art interprets our dreams, the computer executes them in the guise of programs!"*.

This seems an important intuition for approaching programming from a constructionist perspective: programs are a join point between our mind and the computer, the interpreter of the formal description of what we have in mind. Thus, programs appeal to (or even exhilarate) our curiosity and ingenuity and are wonderful

artifacts to share and discuss with other active minds. Such a sharing, however, assumes that the interpreter is a "common ground" among peers. When a group of people program the same 'machine', a shared *semantics* is in fact given, but unfortunately people, especially novices, do not necessarily write their programs for the formal interpreter they use, rather for the *notional machine* [Berry and Kölling 2014; Sorva 2013] they actually have in their minds.

A notional machine is an abstract computer responsible for executing programs of a particular kind [Sorva 2013] and its grasping refers to all the general properties of the machine that one is learning to control [Boulay 1986]. The purpose of a notional machine is to explain, to give intuitive meaning to the code a programmer writes. It normally encompasses an idealized version of the interpreter and other aspects of the development and run-time environment; moreover it should bring also a complementary intuition of what the notional machine cannot do, at least without specific directions of the programmer.

To introduce a notional machine to the students is often the initial role of the instructors. Ideally this should be somewhat incremental in complexity, but not all programming languages are suitable for incremental models: in fact most of the success for introductory courses of visual languages (see 3.5) or lisp dialects (see 3.4 and 3.1) is that they allow shallow presentations of syntax, thus focusing the learners on the more relevant parts of their notional machines [4].

An explicit reference to the notional machine can foster metacognition and during team work it can help in identifying misconceptions (see 2.5). But how can the notional machine be made explicit? The discussion with novice programmers should be guided by an effort of tracing the computational process and visualizing the execution, in order to make as clear as possible what one expects the notional machine will do and what it actually does.

2.4 Abstract programming patterns

One of the most relevant competencies to be mastered when learning computer programming is that of recognizing how a relatively low number of abstract patterns can be applied to a potentially infinite spectrum of specific conditions. This is often a challenge for novices, given that most of the times the discipline is taught using a two-step procedure based on: (i) introducing one or more primitive tools (say functions, variables, or control flow statements), and (ii) showing some (small number of) examples highlighting how these tools can be combined together in order to solve specific problems. This might lead to the rise of *misconceptions* of pupils w.r.t. the above mentioned tools (see 2.5 for more details).

The concept of *role of variables* [Proulx 2000; Sajaniemi 2002] has been proposed in order to suggest a more constructionist-like learning path in which knowledge about variables is built exploiting some concepts at an intermediate level between those of the operational definition of a variable as the holder of a mutable value

---

[3]The pictures shown in the first page show the first and last phase of the "Algomotricity and Mazes" activity, respectively.

[4]On the other hand, several languages of widespread use by experienced programmers are in this sense more complex to handle in the context of introductory courses on programming. This is due to the fact that they force the novice to perform true leaps of faith in accepting an intricate syntax required even to write the simplest programs, and definitely obfuscating the underlying notional machine. Just to state an example, implementing in Java a canonical "hello, world!" requires the programmer to (i) define a class, (ii) add to the latter a public, static, void method, (iii) provide a contract for this method, written in terms of an array of strings, and (iv) actually write the only relevant line of code, yet invoking a static method on a class variable of a system class.
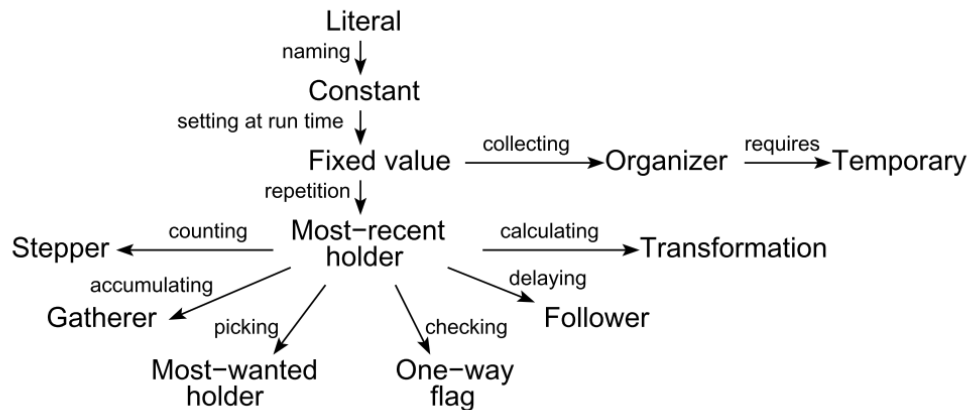
Figure 2: Roles of variables, organized in a constructionist-like hierarchy where the predecessor of an arrow is a prerequisite for learning the corresponding successor (source: [Sajaniemi 2002])

of a given type (say, of a float variable storing numeric quantities) and its specific use cases in order to solve given problems (for instance, that of computing the maximal value within a sequence of numeric quantities). Here, the key concept is related to an *abstract* use of variables — regardless of their type — following a small number of *roles* (namely: *fixed value*, *stepper*, *follower*, *most-recent holder*, *most-wanted holder*, *gatherer*, *transformation*, *one-way flag*, *temporary*, and *organizer*). Just to state a couple of examples,

- *most-wanted holder* identifies the role of a variable storing the most appropriate value found at any intermediate execution time in order to solve a problem (e.g., the variable typically containing the maximal encountered value while scanning a sequence), and
- the *follower* role applies to all variables whose values is always copied from other variables when the latter are about to be changed (like in the naive algorithm used in order to teach how to generate the Fibonacci sequence without introducing recursion or the golden ratio).

Keeping up with the example of finding a maximal value, instead of jumping right away to the algorithmic solution, there is a great opportunity in letting pupils reason about how this problem is in fact a special case of the more general quest for an optimal value which can be found exploiting a *greedy* strategy. This strategy consists in using a *most-wanted holder* to be compared each time with a new element of the sequence, possibly updating the former if the new element allows us to find a more precise solution. This general-purpose method easily fits the search of the maximal value, as well as the minimal one, but it allows us to efficiently handle less obvious cases such as that of finding the distinct vowels occurring in a sentence.

The roles themselves fit the constructionist approach also because they can be gradually introduced following the hierarchy illustrated in Figure 2, starting from the concept of *literal* (e.g., an integer value or a string) and building knowledge about one role on the top of the knowledge of already understood roles.

```
while True:
    value = input('insert a positive, odd value')
    if value > 0 and value % 2 == 1:
        break
    print('the value is not valid')
```

Figure 3: A typical *loop and a half* pattern applied to the repeated validation of external input to a procedure

Loop patterns [Astrachan and Wallingford 1998] are an analogous interesting teaching subject when the concept to be learned identifies with iterations. For instance, the *loop and a half* pattern is introduced as an efficient way of implementing a processing strategy to be applied to a sequence of elements whose end can be detected only *after* having accessed at least one of the elements themselves. The pattern here resorts to rebuilding the almost extinguished repeat/until iterative structure [Roberts 1995] using an infinite loop whose body accesses the next sequence element and subsequently checks whether or not it is the last one. If it is, the loop is escaped through a controlled jump, otherwise some special actions (such as printing a warning) are executed before proceeding to the next iteration. In this way, there is no need of duplicating code lines (typically accessing the first element in the sequence outside the loop), and the check concerns the end of sequence rather than its logical negation (which could be harder to grasp if it involves non-trivial logical connectives). Figure 3 shows one of the canonical incarnations of this pattern, namely the possibly repeated check of a value given as input via keyboard, detecting and ignoring invalid entries; other incarnations concern in general the handling of external inputs (for instance when reading from a file), or the serialization of sequences which have been generated computationally. It is worth noting that a loop might encompass more than one pattern simultaneously: for instance the code in Figure 3 is also an example of *polling loop*.

Loop patterns fit well within a constructionist-based learning path also because they naturally arise when critically analyzing a less efficient implementation of a loop: for instance, the previous polling loop could be the point of arrival of a reasoning scheme which started from the detection of a duplicated line of code in a quick-and-dirty implementation proposed by pupils.

In general, abstract programming patterns are provided in a short number, so as to be easy to fully cover this subject within a standard introductory course in computer programming; moreover, the related concepts are easily and rapidly grasped by experienced computer science teachers [Ben-Ari and Sajaniemi 2004], thus they can be embedded in already existing curricula with low effort.

2.5 Misconceptions

Sorva defines a **misconceptions** as *understandings that are deficient or inadequate for many practical programming contexts* [Sorva 2013].

Some authors [Ben-Ari 2001] believe that computer science has an exceptional position in constructivist's view of knowledge constructed by individuals or groups rather than a copy of an ontological reality: in fact, the computer forms an "accessible ontological reality" and programming *features many concepts that are precisely defined and implemented within technical systems [...] sometimes a novice programmer "doesn't get" a concept or "gets it wrong" in a way that is not a harmless (or desirable) alternative interpretation. Incorrect and incomplete understandings of programming concepts result in unproductive programming behavior and dysfunctional programs* [Sorva 2013].

According to Clancy [Clancy 2004] there are two macro-causes of misconceptions: *over- or under-generalizing* and *a confused computational model.* High-level languages provide an abstraction on control and data, making programming simpler and more powerful, but, by contrast, hiding details of the executor to the user, who can consequently find mysterious some constructs and behaviors.

Much literature about misconceptions in CSEd can be found: we list some of the most important causes of misconceptions, experienced especially by novices, divided into different areas, found mainly in [Clancy 2004; Sirkiä 2012; Sorva 2013] and in works they reference. For a complete review see for example [Qian and Lehman 2017].

English Keywords of a language do not have the same meaning in English and programming. For example, the word *while* in English indicates a constantly active test, while the construct `while` can test the condition again only at the beginning of the next iteration. Some students believe that the loop ends at the precise moment the condition is falsified. Similarly, some of them think of the `if` construct as a test continuously active and awaiting the occurrence of a condition, others believed that the `then` branch is executed as soon as the condition becomes true.

Syntax Although one may think the syntax is one of the biggest sources of misconception, studies show that it is a problem only in the very early stages. In particular, some students were able to write syntactically valid programs, which, however, were not useful for solving the given problem, or were semantically incorrect.

Mathematical notation Reported by many authors, classical is the confusion that generates the assignment with the = symbol (for example, seen as an equation or as a swap of values between variables) or the increment (`a = a + 1`) thought of as an impossible equation.

Examples of over-generalization Some authors found a series of non-existent constraints (e.g., methods in different classes that must have different names, arguments that can only be numbers, "dot" operator usable just in methods) dictated by the fact that the students had not seen any counterexample for such situations.

Similarities The analogy "a variable is like a box" can foster the idea that - like a box - it can contain more elements at the same time. The analogy "programming with the computer is like conversing with it" can bring to attribute *intentionality* to the computer and therefore to think that it:
- has a hidden intelligence that understands the intentions of the programmer and helps him achieve his goal (the so-called "superbug");
- has a general vision, knowing also what will happen in lines of code that it is not currently running.

Some aspects of programming are particular carriers of misconceptions.

Sequence Many misconceptions are due to lack of understanding of the program flow: all lines active at the same time "magic" parallelism, order of instructions not important, difficulty in understanding the branches.

Passing parameters Students present difficulties in this area, for example by confusing the types of passing (by value, by reference ...), making confusion with the return value or with parameters scope.

Input Input statements are particularly problematic. Students do not understand where the input data come from, how they are stored and made available to the program. Some of them believe that a program remembers all the values associated with a variable (its "history").

Memory allocation There are considerable difficulties in understanding the memory model of languages where allocation happens implicitly.

3 PROGRAMMING LANGUAGES FOR LEARNING TO PROGRAM

Ptydepe, as you know, is a synthetic language, built on a strictly scientific basis. [...] There are many months of intensive study ahead of you, which can be crowned by success only if it is accompanied by diligence, perseverance, discipline, talent and a good memory. And, of course, by faith. [Václav Havel]

From a constructionist viewpoint of learning, programming languages have a major role: they are a key means for sharing artifacts and expressing one's theories of world. The crucial part is that artifacts can *be executed* independently from the creator: someone's (coded) mental process can become part of the experience of others, and thus criticized, improved, or adapted to a new project. In fact, the origin of the notion itself of constructionism goes back to Papert's experiments with a programming environment (LOGO, see 3.1) designed exactly to let pupils tinker with math and geometry (a similar approach applied to physics had a smaller impact) [Papert 1980]. Does this strategy work even when the learning objective

```
TO CIRCLE
  REPEAT FOREVER
    [
     FORWARD 1
     RIGHT 1
    ]
```

Figure 4: A procedure to draw a circle in LOGO

is the programming activity itself? Can a generic programming language be used to give a concrete reification of the computational thinking of a novice programmer? Or do we need something specifically designed for this activity? Alan Kay says that programming languages can be categorized in two classes: "agglutination of features" or "crystalization of style" [Kay 1993]. What is more important for learning effectively in a constructivist way? Features or style?

In the last decade, a number of block-based visual programming tools have been introduced which should help students to have an easier time when first practicing programming. These tools, often based on web-based technologies like Adobe Flash and later JavaScript, CSS, and HTML5, as well as an increase in the number of modern smartphones and tablets, opened up new ways for innovative coding concepts [Kahn 2017]. In general, they focus on younger learners, support novices in their first programming steps, can be used in informal learning situations, and provide a visual/block-based programming language which allows students to recognize blocks instead of recalling syntax [Tumlin 2017]. Many popular efforts for spreading computer science in schools, like [Goode et al. 2012] or the teaching material from Code.org [Code.org 2018] rely on the use of such block based programming environments. In addition, such tools are broadly integrated in primary through secondary schools, and even at universities, thus they have been adopted into many computing classes all over the world [Meerbaum-Salant et al. 2010].

3.1 LOGO

LOGO was designed (since 1967) for (constructionst) educational purposes by Wally Feurzeig, Seymour Papert and Cynthia Solomon [Papert 1980]. Its syntax was heavily influenced by Lisp (at the time the standard language for Artificial Intelligence research) and LOGO featured a graphical (at least in principle) environment: the instructions the programmer writes are directed to a "turtle" (a small isosceles triangle in which the acutest angle marks the head) who moves around the screen, possibly leaving a colored trace. The turtle should help learners (typically 6th-8th graders) with a sort of self-identification: its movements have a clear correspondence with their movements in the real world, (although the turtle moves in a 2D space). The patterns drawn by the turtle can be the way the learners build their understanding of 2D geometry, discovering in the process even deep mathematical truths as the fact that a circle can be approximated by a high number of straight segments [Abelson and DiSessa 1986] (see Figure 4).

LOGO was conceived to empower learners of geometry and kinematics, not programming. Programming is *just* a means of expression, but one with a great epistemic potential. According to

Papert: "in teaching the computer how to think, children embark on an exploration about how they themselves think. The experience can be heady: Thinking about thinking turns every child into an epistemologist, an experience not even shared by most adults" [Papert 1980]. Also, by expressing something in a way even the LOGO turtle can "understand" can be fruitful even for real world activities. Juggling, for example, a complex motoric activity which requires dedicated training, can be analyzed with LOGO: the identification of *proper* sub-activities (*i.e.,* sub-routines like TOP-RIGHT to recognize when one juggling ball is at the top of its trajectory going to the right, or TOSS-LEFT to throw the ball with the left hand) may shorten significantly the time for acquiring juggling skills (from days to hours, according to [Papert 1980]). And here 'proper' should be understood as appropriate to the task, but also as "fitting properly with the programming language idiomatic way of describing computational processes". LOGO had many independent implementations and its approach is still very popular, even Python has a `turtle` package in its standard library.

3.2 Smalltalk

Smalltalk [Goldberg and Kay 1976] also has its roots in constructionist learning. Back in the early seventies, at the Learning Research Group within the Xerox Parc Research Center, people were envisioning a world of personal computing devices: they should have intuitive user interfaces and an explicit "programmability". Smalltalk, with whose lineage traces clearly to LOGO and Lisp, was designed with a general audience in mind, since everyone should be comfortable with programming and computing devices should become ubiquitous in learning environments "along the lines of Montessori and Bruner" [Kay 1993]. Thus, although clearly designed to foster a personal learning experience, Smalltalk was not directed specifically to children and it has conquered a wide professional audience. In Smalltalk everything is an 'object' able to react to 'messages'. RiThere follows a highly consistent object-oriented approach and code can be factored out by inheritance and dynamic binding. Smalltalk introduces also the idea that everything in the system is programmable: in fact, the tool-chain itself and the application a programmer is writing are indistinguishable and available for modifications, even at run-time. By design, such a dynamic environment encourages a trial-and-error approach. A common practice for Smalltalk developers is programming with the constant support of the debugger: instead of creating a method before its call, one sends a message that an object "does not understand", then they use the debugger to catch the exception and write the code that is needed. A specific Smalltalk system for children was designed later as an evolution of Squeak Smalltalk: E-toys [Kay et al. 1997] provided a world of "sprites", funny characters that can be moved (concurrently) around the screen by programming them in Smalltalk. E-toys then evolved in Scratch (see 3.5), where the programming part was replaced by visual blocks.

3.3 BASIC, Pascal

The search for programming languages suitable for students in fields other than hard sciences and mathematics was in fact started in the sixties. In 1964 at Dartmouth College rose BASIC (Beginner's All-purpose Symbolic Instruction Code) [Kurtz 1978]. It seems legit

to mention BASIC in a paper on constructionism and programming: for years BASIC has been the elective language for personal projects and even before widespread Internet connectivity, several communities shared BASIC programs in Bulletin Board Systems and magazines. Its popularity among self-taught programmers, however, was due mainly to its availability on personal and home computing devices. Moreover, the language was typically implemented using an interpreter, thus naturally fostering the trial-and-error and incremental learning styles typical of a constructionist setting. A generation grown with BASIC still thinks it is a wonderful approach to get children hooked on programming (see for example [Brin 2016]). However, many believe BASIC is not able to foster good abstractions and fear that BASIC programmers will bring bad habits to all their future computational activities[5].

In 1970 Niklaus Wirth published Pascal [Wirth 1993], a small, efficient ALGOL-like language intended to encourage good programming practices using structured programming and data structuring. For about 25 years, Pascal (and its successors like TurboPascal or Modula-2) was the most popular choice for undergraduate courses and a whole generation of computer scientist learned to program through its discipline of well-structured programs popularized by Wirth in his book "Algorithms + Data Structures = Programs". Only Java had a similar success in undergraduate courses. However, while Java popularity was (and is) influenced by trends in software industry, Pascal was appealing mainly for its intrinsic discipline, which matched the academic sentiment of the time. Alan Perlis, in his foreword to "Structure and Interpretation of Computer Programs" (see 3.4), says: "Pascal is for building pyramids—imposing, breathtaking, static structures built by armies pushing heavy blocks into place. Lisp is for building organisms—imposing, breathtaking, dynamic structures built by squads fitting fluctuating myriads of simpler organisms into place. The organizing principles used are the same in both cases, except for one extraordinarily important difference: The discretionary exportable functionality entrusted to the individual Lisp programmer is more than an order of magnitude greater than that to be found within Pascal enterprises. Lisp programs inflate libraries with functions whose utility transcends the application that produced them. The list, Lisp's native data structure, is largely responsible for such growth of utility. The simple structure and natural applicability of lists are reflected in functions that are amazingly nonidiosyncratic. In Pascal the plethora of declarable data structures induces a specialization within functions that inhibits and penalizes casual cooperation. It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures. As a result the pyramid must stand unchanged for a millennium; the organism must evolve or perish."

Structured programming had also some LOGO-like descendants, the most famous one is probably Karel [Pattis 1981], in which the programmer controls a simple robot that moves in a grid of streets (left-right) and avenues (up-down). A programmer can create additional instructions by defining them in terms of the five basic instructions, and by using conditional control flow statements with environment queries.

3.4 Scheme, Racket

Scheme [Abelson et al. 1998] is a language originally aimed at bringing structured programming in the lands of Lisp (mainly by adding lexical scoping). The language has been standardized by IEEE in 1999 and nowadays it has a wide and energetic community of users. Its importance in education, however, is chiefly related to a book, "Structure and Interpretation of Computer Programs" (SICP) [Abelson et al. 1996], which had a tremendous impact on the practice of programming education. The book derived from a semester course taught at MIT. It has the peculiarity to present programming as a way of organizing thinking and problem solving. Every detail of the Scheme (which, being a Lisp dialect, has lightweight syntax) notional machine is worked out in the book: in fact at the end, the reader should be able to understand the mechanics of a Scheme interpreter and to program one by her/himself (in Scheme). The book, which enjoyed widespread adoption, was originally directed to MIT undergraduates and it is certainly not suitable either for children or even adults without a scientific background: examples are often taken from college-level mathematics and physics. Its emphasis on "organized abstraction" and "procedural epistemology" makes it a fundamental reading for anyone reflecting on teaching and learning how to build complex systems.

A spin-off of SICP explicitly directed to learning is Racket. Born as 'PLT Scheme', one of its strength is the programming environment DrScheme [Findler et al. 2002] (now DrRacket): it supports educational scaffolding, it suggests proper documentation, and it can use different *flavours* of the language, starting from a very basic one (Beginning Student Language, it includes only notation for function definitions, function applications, and conditional expressions) to multi-paradigm dialects; this flexibility is relatively easy in a Lisp-like world, since most of the "syntax" is in fact provided by macros, that can be active or not[6]. The DrRacket approach is supported by an online book "How to design programs" (HTDP) [7] and it has been adapted to other mainstream languages, like Java [Allen et al. 2002] and Python. The availability of different languages directed to the progression of learning should help in overcoming what the DrRacket proponents identify as "the crucial problem" in the interaction between the learner and the programming environment: beginners make mistakes before they know much of the language, but development tools yet diagnose these errors as if the programmer already knew the whole notional machine. Moreover, DrRacket has a minimal interface aimed at not confusing novices, with just two simple interactive panes: a definitions area, and an interactions area, which allows a programmer to ask for the evaluation of expressions that may refer to the definitions. Similarly to what happens in visual languages, Racket allows for direct manipulation of sprites, see an example in Figure 5.

The authors of HTDP claim that "program design — but not programming — deserves the same role in a liberal-arts education as mathematics and language skills." They aim at systematically designed programs thanks to systematic thought, planning, and understanding from the very beginning, at every stage, and for every step. To this end the HTDP approach is to present "design

---

[5]A recent anecdote: Brian Kernighan — one of the designers of C — called a book written by a BASIC programmer "the worst C programming textbook ever written"! See https://wozniak.ca/blog/2018/06/25/Massacring-C-Pointers/index.html for the full story.

[6]A small syntactic improvement of Racket over Lisp, very useful for beginners, is that nested parentheses can be matched easily by using different bracket families: for example, (- {/ [* (+ 2 3) 4] 2 } 1)

[7]Current version: http://www.htdp.org/2018-01-06/Book/index.html

```
(define (picture-of-rocket.v3 height)
  (cond

    [(<= height (- 60 (/ (image-height 🚀) 2)))

     (place-image 🚀 50 height
                  (empty-scene 100 60))]

    [(> height (- 60 (/ (image-height 🚀) 2)))

     (place-image 🚀 50 (- 60 (/ (image-height 🚀) 2))
                  (empty-scene 100 60))]))
```

Figure 5: Racket code for "landing a rocket"

recipes", supported by predefined scaffolding that should be iteratively refined to match the problem at hand. This is indeed very close to the idea of micropatterns discussed in 2.4.

3.5 Scratch, Snap!, Alice, and others

EToys worlds (see 3.2) with pre-defined — although programmable — objects, evolved in a generic environment in which everything can be defined in terms of 'statement' blocks. Scratch [Resnick et al. 2009], originally written in Smalltalk (but this is hidden to most users: only a "secret" key combination can bring the Smalltalk environment alive), is the most popular and successful visual block based programming environment. Launched in 2007 by the MIT Media Lab, the Scratch site has grown to more than 25 million registered members with over 29 million Scratch projects shared programs.

Unlike traditional programming languages, which require code statements and complex syntax rules, here graphical programming blocks are used that automatically snap together like Lego bricks when they make syntactical sense [Ford 2009]. In visual programming languages, a block represents a command or action and they are arranged in scripts. The composition of individual scripts equals the construction of an algorithm. The building blocks offer the possibility, e.g., to animate different objects on the stage, thus defining the behavior of the objects. In addition to the basic control structures, there are event-triggering building blocks/conditions for event-driven programming [Georgios and Kiriaki 2009]. Familiar concepts such as variables, variable lists, Boolean logic, user interface design, etc. are provided as well. Furthermore, most visual programming environments offer the possibility to integrate graphics, animations, music, and sound to create video games, movies, and interactive stories. In that way, creative and artistic talents of the students are displayed in their games, stories, and applications. Thereby, these visual languages offer the same programming logic and concepts as other (text-based) programming languages.

Some characteristics of the Scratch environment [Maloney et al. 2010] are particularly relevant in the constructionist approach.

Liveness The code is constantly executed and can be changed on the fly, immediately seeing the runtime effects of the change; this encourages users to tinker with the code.

No error messages When you play with Lego bricks, they stack together or they don't - the same happens in Scratch; program always run: syntax errors are prevented from the block shapes and connections, and also runtime errors are avoided by doing something "reasonable" (e.g., in the case of an out-of-range value); this is particularly important not to frustrate kids and to keep them iterating and developing:

"*A program that runs, even if it is not correct, feels closer to working than a program that does not run (or compile) at all*" [Maloney et al. 2010].

Other characteristics are useful to help novices avoiding misconceptions that often arise when starting to learn to program.

Execution made visible A glowing yellow border surrounds running scripts; moreover version 1.4 (and Snap!, for example) provides a "single-stepping" mode, where each block is highlighted when it is executed; this is very helpful in program reading and debugging, and helps students form a correct mental model of the notional machine underlying the program execution.

Making data concrete You can see in a variable box, automatically shown, its current value: again, this is helpful for making the underlying machine model visible.

Finally, other characteristics introduce important software engineering and development concepts.

Open source Each shared project has a "see inside" button that brings you to the project source; you can read and edit the blocks to see what happens.

Remixing If you edit someone else's project, you create a remix: you are the author, but the system automatically gives credits to the original author (at any depth, keeping track of multiple remixes in a tree) and suggests you to explicitly declare what changes you made.

The main limitation of Scratch programs is that they do not scale well from the abstraction point of view: only since version 2 you can "make a new block" that is, a procedure with optional parameters. These blocks have no possibility to return a value (like a number or a boolean) and so can't be nested inside other blocks, forcing you to modify global variables if needed.

Snap![8] (originally BYOB, Build Your Own Blocks) is an extended reimplementation of Scratch with functions and continuations. These added capabilities make it suitable for a serious introduction to computer science for high school or college students: in fact, Snap! is used as the basis for an Advanced Placement CS course at Berkeley[9].

The Scratch approach was also ported to mainstream programming languages: in Alice [Dann et al. 2008] visual blocks are in fact Java instructions. Alice worlds are 3D: this choice makes it very attractive and appealing to pupils, who can program amazing 3D animations. It also adds many complexities, since moving objects in a 3D space is not trivial.

Recently, several block-based development environments for web-browsers were published (even the last version of Scratch is web based, thus abandoning its Smalltalk inner soul). The most popular is probably Google's Blockly[10], which allows for programming both with blocks and textual programming languages (Javascript and Python): the programmer can see the source code in different interchangeable ways. MIT has developed a version of Blockly that can even be used to create Android applications, to be executed on mobile phones and that can take advantage of sensors and Google

---

[8] https://snap.berkeley.edu/
[9] https://bjc.berkeley.edu/
[10] https://developers.google.com/blockly/

services like maps (App Inventor[11]). Even Apple recently proposed a block-based interface for its Swift programming language[12] and other Android-based visual programming language environments exist. For example, Pocket Code allows the creation of games, stories, animations, and many types of other apps directly on phones or tablets, thereby teaching fundamental programming skills [Slany 2014]. The free and open source project Catrobat [13] was initiated 2010 in Austria at TU Graz. This team develops free educational apps for teenagers with the aim to introduce young people to programming. With a playful approach, young people can be engaged and game development can be promoted with a focus on design and creativity. The drag and drop interface provides a variety of bricks that can be joined together to develop fully fledged programs. The app is freely available for Android at the Google's Play Store [14] and soon it will be available on the Apple App Store for iOS. The target audience for Pocket Code are teenagers between the age of 11 and 17 years old who have access to or own an Android smartphone.

Since its first versions, Scratch had blocks able to connect and program external (physical) robots. In fact most of the mentioned environments can connect to physical devices and sensors, with the goal of increasing the constructionist appeal of block programming, and opening to the world of "tinkering" with electronics. Resources like the 'Makey Makey' [15] tool became popular for activities during coding workshops with innovative forms of production and do-it-yourself work [Schön et al. 2014; Sheth et al. 2012].

All in all, visual programming languages seem to provide an easier start and a more engaging experience for learners. The ease of use, simplicity, and desirability of new visual programming environments enables young people to imagine complex goals. A study which compared three classes that used either block-based (Scratch), text-based (Java), or hybrid blocks/text (Snap!/JavaScript) programming languages showed that students generally found block-based programming to be easier than the text-based environments [Weintrop and Wilensky 2015]. Some researchers, however, argue that students are not fully convinced that a visual language can help them learn other programming languages [Lewis et al. 2014].

3.6 Common features

The above short survey of programming languages for education shows they have some recurrent traits that link them to the themes discussed in Sect. 2.

Personification The interpreter becomes a "persona", computation is then carried out through anthropomorphic (or, better, zoomorphic, since animals are very common) actions. This seems to contradict a famous piece of advice coming from no less than E. W. Dijkstra [Dijkstra 1985]. Speaking of anthropomorphism in computer science, he noted: "The trouble with the metaphor is, firstly, that it invites you to identify yourself with the computational processes going on in system components and, secondly, that we see ourselves as existing in time. Consequently the use of the metaphor forces one to what we call 'operational reasoning', that is reasoning

in terms of the computational processes that could take place. From a methodological point of view this is a well-identified and well-documented mistake: it induces a combinatorial explosion of the number of cases to consider and designs thus conceived are as a result full of bugs." The *reasoning in terms of the computational processes*, however, is what is probably needed for a novice in order to familiarize with the notional machine. Some sort of operational reasoning seems also important to foster the basic intuition needed by a constructivist approach, in which concrete, public actions are key.

Visualization and tracking Computational processes that evolve in time are described by static texts: the mapping between the two is not trivial and it requires an understanding of the notional machine. Educational programming environments often try to make the mapping more explicit with some visualization of the ongoing process: the trace left by the LOGO turtle, or some other exposition of the changing state of the interpreter.

Appeal Engagement of learners is crucial: to this end it is important to give learners powerful libraries and building blocks. It is not clear, however, how to properly balance amazing effects in order to avoid they become a major distraction: sometimes children may spend their (limited) time in changing the colors of the sprites, instead of trying to solve problems. While they surely learn something even in these trivial activities, they also risk to lose their opportunity of recognizing the thrill which comes when solving computational problems by themselves.

4 LEARNING TO THINK COMPUTATIONALLY

The expression "Computational Thinking" (CT) has a very special relationship with Papert's constructionisim: the first ascertained attestation is in Papert's seminal book Mindstorms [Papert 1980]. As already stated in 3.1, Papert was not focused on teaching computer science concepts, but on the use of computation as a means of self expression and as a tool for learning ideas and concepts of other disciplines.

The expression almost completely disappeared (except for a Papert's work about math [Papert 1996]) till 2006.

In 2006 Jeannette Wing brought the expression "Computational Thinking" back to the discussion [Wing 2006], gaining a massive attention[16]. In that seminal article, Wing didn't give a definition, but tightened the concept to the computer science discipline, stating "*Computational thinking involves solving problems, designing systems, and understanding human behavior, by drawing on the concepts fundamental to computer science*" or that "*thinking like computer scientists*" is a fundamental skill for everyone.

In the following years she proposed (along with Cuny, Snyder and Aho) a formal definition: CT is "*the thought processes involved in formulating problems and their solutions so that the solutions are represented in a form that can be effectively carried out by an*

---

information-processing agent" [Wing 2010]. Wing argues that children who are introduced to CS learn more than just programming and that this opens a new way of thinking [Wing 2008]. Moreover, students should think first about possible solutions to a given problem (problem solving skills), and second implement their ideas by using a computing device (programming skills [Lye and Koh 2014]). In order to successfully implement their own solutions, students have to apply different programming concepts, such as loops and conditions, as well as practices, such as abstraction and debugging [Kafai and Burke 2013]. For teachers, however, developing computational thinking skills in students is actually a challenging task [Sentance and Csizmadia 2015]. After Wing's articles, the discussion about the nature of CT and how to teach it in fact exploded.

In [Corradini et al. 2017] authors analysed different CT definitions and frameworks, finding out that all definitions agreed *on the fact that CT is a* **way of thinking** *(thought process) for* **problem solving** but specifying that *it is not just problem solving: the formulation and the solution of the problem must be expressed in a way that allows a* **processing agent** *(a human or a machine) to carry it out.*

Moreover, authors in [Corradini et al. 2017] noticed that each definition listed some characteristics of CT, from thinking habits to specific programming concepts. They classified them in four categories. We quote here their findings.

Mental processes: Mental strategies useful to solve problems.

- *Algorithmic thinking*: use algorithmic thinking to design a sequence of ordered steps (instructions) to solve a problem, achieve a goal or perform a task.
- *Logical thinking*: use logical thinking and reasoning to make sense of things, establish and check facts.
- *Problem decomposition*: split a complex problem in simpler subproblems to solve it more easily; modularize; use compositional reasoning.
- *Abstraction*: get rid of useless details to focus on relevant information / ideas.
- *Pattern recognition*: discover and use regularities in data, problems.
- *Generalization*: use discovered similarities to make predictions or to solve more general problems.

Methods: Operational approaches widely used by computer scientists.

- *Automation*: automate the solutions; use a computer or a machine to do repetitive tasks.
- *Data collection, analysis and representation*: gather information/data, make sense of them by finding patterns, represent them properly; store, retrieve and update values.
- *Parallelization*: carry out tasks simultaneously to reach a common goal, use parallel thinking.
- *Simulation*: represent data and (real world) processes through models, run experiments on models.
- *Evaluation*: implement and analyze solutions to judge them, in particular for what concerns effectiveness, efficiency in terms of time and resources.

- *Programming*: use some common concepts in programming (eg. loops, events, conditionals, mathematical and logical operators).

Practices: Typical practices used in the implementation of computing machinery based solutions.

- *Experimenting, iterating, tinkering*: in iterative and incremental software development, one develops a project with repeated iterations of a design-build-test cycle, incrementally building the final result; tinkering means trying things out using a trial and error process, learning by playing, exploring, and experimenting.
- *Test and debug*: verify that solutions work by trying them out; find and solve problems (bugs) in a solution/ program.
- *Reuse and remix*: build your solution on existing code, projects, ideas.

Transversal skills: General ways of seeing and operating in the world; useful life skills enhanced by thinking like a computer scientist.

- *Create*: design and build things, use computation to be creative and express yourself.
- *Communicate and collaborate*: connect with others and work together to create something with a common goal and to ensure a better solution.
- *Reflect, learn, meta-reflect*: use computation to reflect and understand computational aspects of the world.
- *Be tolerant for ambiguity*: deal with non-well specified and open-ended real-world problems.
- *Be persistent when dealing with complex problems*: be confident in working with difficult or complex problems, persevering, being determined, resilient and tenacious.

As you may notice, a lot of concepts are involved. Sometimes this led to critiques [Hemmendinger 2010]: some of these concepts are not exclusively associated with CS, but taught in other disciplines (e.g., math) or are general skills that children have been learning for a long time before the birth of CS. Anyway CS brings to the discussion some characteristic problem solving methods (e.g., the possibility to effectively execute a solution/a model/an abstraction by running an implementation of its algorithm [Martini 2012]). Constructionist's spirit emerges mainly in practices and transversal skills, but we should pay attention not to forget mental processes and CS methods: loosing this connection may lead to misconceptions about the nature of computational thinking, de-empowering Wing's idea of the importance of "thinking like computer scientists" in many human activities.

The most used technique to teach CT is teaching to program (with languages suitable for the learner's age and capacity). Programming is in fact the main way computer scientists express their solutions and so the main way they learn to think computationally. Teaching to program leads to problems and opportunities discussed in the previous sections. Nonetheless, unplugged activities (see 2.2) have been proposed as well, as they help kids to act like the

computer, becoming aware of the underlying notional machine (see 2.3).

5 LEARNING TO PROGRAM IN TEAMS

Working in teams requires new skills, especially because software products (even the ones in the reach of novices) are often tangled with many dependencies and division of labour is hard: it inevitably requires appropriate communication and coordination. Therefore, it is important that novice programmers learn to program in an "organized" way, thus discovering that as a group they are able to solve more challenging and open-ended problems, maybe with interdisciplinary contributions.

To this end, agile methodologies seem to fit well with the needs of a constructionist team of learners and they are increasingly exploited in educational settings (see for example [Kastl et al. 2016; Missiroli et al. 2016]):

- agile teams are typically small groups of 4–8 co-workers;
- agile values [Beck et al. 2001] (individuals and interactions over processes and tools; customer collaboration over contract negotiation; responding to change over following a plan; working software over comprehensive documentation) relate well with constructivist philosophies;
- agile teams are self-organizing and emphasize the need of reflecting regularly on how to become more effective, and tune and adjust their behavior accordingly;
- typical agile techniques like pair programming, test driven development, iterative software development, continuous integration are very attractive for a learning context.

5.1 Iterative software development

For program development cycles, concepts of agile and iterative software development can be used to leverage this process and to see first results very quickly [Davies and Sedley 2009; DeMarco-Brown 2013]. Figure 6 visualizes the process of game design in reference to agile methods. As the scope of this paper is limited, only a simplified life cycle is visualized to describe the process of how a team works in iterations to deliver or release software.

The first step, *Research*, includes the development of the core idea by producing a simple game concept or a storyboard. In this phase the story, title, genre, and theme of the game should be selected, as well as a rough concept about the structure and gameplay.

In the second step, during the *Design*, the artwork, game content and other elements (characters, assets, avatars, etc.), and the whole gaming word is produced.

The *Development* phase, includes all of the actual programming, followed by *Testing* the code and the software (playtesting). Several iterations between testing and bug fixing are possible.

As a final step, the *Release* phase is planned. This could include several beta releases or a final release for end users. The agile model required to get started with the project works to bring customer satisfaction by rapid, continuous delivery of useful software.

> "In an iterative methodology, a rough version of the game is rapidly prototyped as early in the design process as possible. This prototype has none of the aesthetic trappings of the final game, but begins to define its fundamental rules
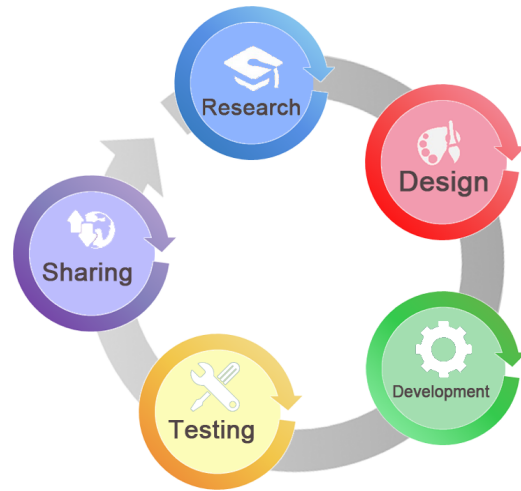
Figure 6: Agile Development Cycle [Davies and Sedley 2009; DeMarco-Brown 2013]

and core mechanics." [Salen and Zimmerman 2003, p. 11]

To phrase it differently, before focusing on every detail of the project, focus on the smallest step the games needs for playing it, e.g., limited interface control but basic game functionality.

Collaborative work and the connection with a real world problem makes their learning valuable [Jaime and Ruby 2011]. Project work is always student-centered and task oriented. The final artefacts of this project work can be shared with a community, thus fostering ownership. The benefits of learning through projects include enhanced students' participation within active learning and self-learning, enhanced communication skills, addressing of a wider set of learning styles, and improved critical thinking skills. This collaboration is needed for developing the game idea, communicating, sharing, and managing assets and codes, playtesting and documentation.

Teachers have to consider how to support collaboration and communication during the whole game production process [Ferreira et al. 2008]. They must therefore stick to certain design patterns and iterative cycles (e.g., agile) and explain game elements and rules. The project in general should foster the teamwork in producing game assets and software. Teachers have to take into account the different preferences of students, e.g., if they feel more confident in the role of developers or artists, or that students do not shy away from switching roles. Finally, the teacher has to ensure that the ideas for the project are simple and clear, as well as reduce the size and complexity of the game projects.

5.2 Test-driven development and debugging

Within the agile framework, a special constructivist emphasis is provided by test-driven development, or TDD for short [Beck 2003]. This technique reverses the classical temporal order between the implementation of code and the test of its correctness. Namely, the specification of the programming task at hand is actually provided

with a test the *defines* correct behavior. The development cycle is then based on the iteration of the following three-step procedure:

(1) write a test known to fail according to the current stage of the implementation,

(2) perform the smallest code update which satisfies all considered tests, including the one introduced in previous point, and

(3) optionally refactor the produced code.

TDD makes testing the engine driving the overall development process, and suggesting a good *next test* is one of the hardest-to-find contributions that a facilitator might give in an active programming learning context. Such suggestion has indeed the role of letting pupils aware that their belief at a broad level ("the program works") is false, thus an analogous belief at a smaller scale (for instance, "this function always returns the correct result") should be false, too. This amounts to the destruction of knowledge necessary to build new knowledge (aka a working program) in a constructivist setting. Even the refactoring step corresponds well to the constructivist re-organization of knowledge that follows the discovery of more viable solutions. In fact, most of the developing activities consist in *realizing* that a more or less complex system which was thought to correctly work actually it is not able to cope with a new arising test case. This applies of course also to the simplest coding tasks faced by students engaged in learning the basis of computer programming.

Once pupils are convinced that their implementation is flawed, the localization of the code lines to be reconsidered is the other pillar of an active learning setting. Again, a paramount contribution for a successful learning process should be provided by a facilitator suggesting suitable debugging techniques (e.g., proposing critical input values to the wrong program, hinting specific points in the execution flow to be verified, or giving advice about variables to be tracked during the next run).

6 CONCLUSIONS

Literature on learning to program through a constructionist strategy mostly focuses on how to bring the abstract and formal nature of programming languages into manipulation of more concrete (or even tangible) "objects". Many proposals aim at overcoming the (initial) hurdles which textual rules of syntax may pose to children. Also, several environments have been designed in order to increase the appeal of programming by connecting this activity to real world devices or providing fancy libraries. Instead, more work is probably needed to make educators and learners more aware of the so-called notional machine behind the programming language. Programming environments could be more explicit about the complex relationship between the code one writes and the actions that take place when the program is executed. Moreover, micro-patterns should be exploited in order to enhance problem solving skills of novice programmers, such that they become able to think about the solution of problems in the typical way that make the former suitable to automatic elaboration. Agile methodologies, now also common in professional settings, seem to fit well with constructionist learning. Besides the stress on team working, particularly useful seems the agile emphasis on having running artifacts through all the development cycle and the common practice of driving development with explicit or even executable "definitions of done".

REFERENCES

H. Abelson and A.A. DiSessa. 1986. *Turtle Geometry: The Computer as a Medium for Exploring Mathematics.* AAAI Press.

H. Abelson, R.K. Dybvig, C.T. Haynes, G.J. Rozas, N.I. Adams, D.P. Friedman, E. Kohlbecker, G.L. Steele, D.H. Bartley, R. Halstead, D. Oxley, G.J. Sussman, G. Brooks, C. Hanson, K.M. Pitman, and M. Wand. 1998. Revised5 Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation* 11, 1 (01 Aug 1998), 7–105.   https://doi.org/10.1023/A:1010051815785

H. Abelson, G.J. Sussman, and J. Sussman. 1996. *Structure and Interpretation of Computer Programs* (second ed.). MIT press.

Eric Allen, Robert Cartwright, and Brian Stoler. 2002. DrJava: A Lightweight Pedagogic Environment for Java. *SIGCSE Bull.* 34, 1 (Feb. 2002), 137–141.   https://doi.org/10.1145/563517.563395

Owen Astrachan and Eugene Wallingford. 1998. Loop patterns. In *Proceedings of the Fifth Pattern Languages of Programs Conference.*

Kent Beck. 2003. *Test-driven development: by example.* Addison-Wesley Professional.

Kent Beck and Cynthia Andres. 2004. *Extreme Programming Explained: Embrace Change* (second ed.). Addison-Wesley Professional.

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. 2001. Manifesto for Agile Software Development.   http://agilemanifesto.org/iso/en/manifesto.html.

T. Bell, F. Rosamond, and N. Casey. 2012. Computer Science Unplugged and Related Projects in Math and Computer Science Popularization. In *The Multivariate Algorithmic Revolution and Beyond*, Hans L. Bodlaender, Rod Downey, Fedor V. Fomin, and Dániel Marx (Eds.). Springer-Verlag, Berlin, Heidelberg, 398–456.   http://dl.acm.org/citation.cfm?id=2344236.2344256

Carlo Bellettini, Violetta Lonati, Dario Malchiodi, Mattia Monga, Anna Morpurgo, and Mauro Torelli. 2012. Exploring the processing of formatted texts by a kynesthetic approach. In *Proc. of the 7th WiPSCE (WiPSCE '12).* ACM, New York, NY, USA, 143–144.   https://doi.org/10.1145/2481449.2481484

Carlo Bellettini, Violetta Lonati, Dario Malchiodi, Mattia Monga, Anna Morpurgo, and Mauro Torelli. 2013. What you see is what you have in mind: constructing mental models for formatted text processing. In *Proceedings of ISSEP2013 (Commentarii informaticae didacticae).* Universitätsverlag Potsdam, 139–147.   http://opus.kobv.de/ubp/volltexte/2013/6368/pdf/cid06.pdf

Carlo Bellettini, Violetta Lonati, Dario Malchiodi, Mattia Monga, Anna Morpurgo, Mauro Torelli, and Luisa Zecca. 2014. Extracurricular activities for improving the perception of Informatics in Secondary schools. In *Proceedings of ISSEP2014 (Lecture Notes in Computer Science)*, Yasemin Gülbahar and Erinç Karataş (Eds.), Vol. 8730. Springer, 161–172.   https://doi.org/10.1007/978-3-319-09958-3_15

Mordechai Ben-Ari. 2001. Constructivism in computer science education. *Journal of Computers in Mathematics and Science Teaching* 20, 1 (2001), 45–73.

Mordechai Ben-Ari and Jorma Sajaniemi. 2004. Roles of variables as seen by CS educators. In *ACM SIGCSE Bulletin*, Vol. 36. ACM, 52–56.

Michael Berry and Michael Kölling. 2014. The State of Play: A Notional Machine for Learning Programming. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education (ITiCSE '14).* ACM, New York, NY, USA, 21–26.   https://doi.org/10.1145/2591708.2591721

Benedict Du Boulay. 1986. Some Difficulties of Learning to Program. *Journal of Educational Computing Research* 2, 1 (1986), 57–73.   https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9

David Brin. 2016. Why Johnny can't code. https://www.salon.com/2006/09/14/basic_2/.

Michael Clancy. 2004. Misconceptions and attitudes that interfere with learning to program. In *Computer science education research*, Sally Fincher and Marian Petre (Eds.). Routledge, 85–100.

Code.org. 2018. HOour of Code. Retrieved July 17, 2018 from https://hourofcode.com/de.

Isabella Corradini, Michael Lodi, and Enrico Nardelli. 2017. Conceptions and Misconceptions About Computational Thinking Among Italian Primary School Teachers. In *Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17).* ACM, New York, NY, USA, 136–144.   https://doi.org/10.1145/3105726.3106194

Wanda P Dann, Stephen Cooper, and Randy Pausch. 2008. *Learning to program with Alice.* Prentice Hall Press.

Rachel Davies and Liz Sedley. 2009. *Agile Coaching.* The Pragmatic Bookshelf.

Diana DeMarco-Brown. 2013. *Agile User Experience Design - A Practitioner's Guide to making it work.* In Elsevier Inc.

Edsger W. Dijkstra. 1985. On anthropomorphism in science. EWD936.   https://www.cs.utexas.edu/users/EWD/ewd09xx/EWD936.PDF.

Alexandre Ferreira, Eliane Pereira, Junia Anacleto, Aparecido Carvalho, and Izaura Carelli. 2008. The common sense-based educational quiz game framework "What is it?". *In ACM International Conference Proceeding Series* (2008).

Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. 2002. DrScheme: A programming environment for Scheme. *Journal of functional programming* 12, 2 (2002), 159–182.

Jerry Lee Ford. 2009. *Scratch programming for Teens.* In Computer Science Books.

Fesakis Georgios and Serafeim Kiriaki. 2009. Influence of the Familiarization with "Scratch" on Future Teachers' Opinions and Attitudes about Programming and ICT in Education. In *In Annual Joint Conference Integrating Technology into Computer Science Education.*

Adele Goldberg and Alan Kay. 1976. *Smalltalk-72 Instruction Manual.* Xerox.

Joanna Goode, Gail Chapman, and Jane Margolis. 2012. Beyond curriculum: the exploring computer science program. *In Magazine ACM Inroads* (2012).

Matthias Hauswirth, Andrea Adamoli, and Mohammad Reza Azadmanesh. 2017. The Program is the System: Introduction to Programming Without Abstraction. In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research (Koli Calling '17).* ACM, New York, NY, USA, 138–142. https://doi.org/10.1145/3141880.3141894

David Hemmendinger. 2010. A Plea for Modesty. *ACM Inroads* 1, 2 (June 2010), 4–7. https://doi.org/10.1145/1805724.1805725

Michael S. Horn and Robert J. K. Jacob. 2007. Designing Tangible Programming Languages for Classroom Use. In *Proceedings of the 1st International Conference on Tangible and Embedded Interaction (TEI '07).* ACM, New York, NY, USA, 159–162. https://doi.org/10.1145/1226969.1227003

Sanchez Jaime and Olivares Ruby. 2011. Problem solving and collaboration using mobile serious games. *In Elsevier Ltd* (2011).

Yasmin Kafai and Q. Burke. 2013. Computer programming goes back to school. *In Phi Delta Kappan* (2013).

Ken Kahn. 2017. A half-century perspective on Computational Thinking. *In technologias, sociedade e conhecimento* (2017).

Petra Kastl, Ulrich Kiesmüller, and Ralf Romeike. 2016. Starting out with Projects: Experiences with Agile Software Development in High Schools. In *Proceedings of the 11th Workshop in Primary and Secondary Computing Education (WiPSCE '16).* ACM, New York, NY, USA, 60–65. https://doi.org/10.1145/2978249.2978257

A. Kay, K. Rose, D. Ingalls, T. Kaehle, J. Maloney, and S. Wallace. 1997. Etoys & SimStories. *Walt Disney Imagineering* (1997).

Alan C. Kay. 1993. The Early History of Smalltalk. *SIGPLAN Not.* 28, 3 (March 1993), 69–95. https://doi.org/10.1145/155360.155364

Thomas E. Kurtz. 1978. BASIC. *SIGPLAN Not.* 13, 8 (Aug. 1978), 103–118. https://doi.org/10.1145/960118.808376

Colleen Lewis, Sarah Esper, Victor Bhattacharyya, Noelle Fa-Kaji, Neftali Dominguez, and Arielle Schlesinger. 2014. Children's perceptions of what counts as a programming language. *In Journal of Computing Sciences in Colleges* (2014).

Violetta Lonati, Mattia Monga, Anna Morpurgo, and Mauro Torelli. 2011. What's the Fun in Informatics? Working to Capture Children and Teachers into the Pleasure of Computing. In *Proceedings of ISSEP2011 (Lecture Notes in Computer Science),* I. Kalaš and R.T. Mittermeir (Eds.), Vol. 7013. Springer-Verlag, 213–224. https://doi.org/10.1007/978-3-642-24722-4_19

S.Y. Lye and J.H.L. Koh. 2014. Review on teaching and learning of computational thinking through programming: What is next for K-12? *In Computers in Human Behavior* (2014).

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

Simone Martini. 2012. Lingua Universalis. *Annali della Pubblica Istruzione* 4-5 (2012), 65–70. https://hal.inria.fr/hal-00909609

O. Meerbaum-Salant, M. Armoni, and M. Ben-Ari. 2010. Learning computer science concepts with scratch. *In Proceedings of the Sixth international workshop on Computing education research* (2010), 69–76.

Marcello Missiroli, Daniel Russo, and Paolo Ciancarini. 2016. Learning Agile Software Development in High School: An Investigation. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16).* ACM, New York, NY, USA, 293–302. https://doi.org/10.1145/2889160.2889180

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

Seymour Papert. 1996. An exploration in the space of mathematics educations. *International Journal of Computers for Mathematical Learning* 1, 1 (01 Jan 1996), 95–123. https://doi.org/10.1007/BF00191473

Seymour Papert and Idit Harel. 1991. *Constructionism.* Ablex Publishing Corporation, Chapter Situating constructionism.

Richard E. Pattis. 1981. *Karel The Robot: A Gentle Introduction to the Art of Programming.* John Wiley & Sons.

Viera K Proulx. 2000. Programming patterns and design patterns in the introductory computer science course. In *ACM SIGCSE Bulletin,* Vol. 32. ACM, 80–84.

Yizhou Qian and James Lehman. 2017. Students’ Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Trans. Comput. Educ.* 18, 1, Article 1 (Oct. 2017), 24 pages. https://doi.org/10.1145/3077618

Mitchel Resnick. 1996. Distributed Constructionism. In *Proceedings of the 1996 International Conference on Learning Sciences (ICLS '96).* International Society of the Learning Sciences, 280–284. http://dl.acm.org/citation.cfm?id=1161135.1161173

Mitchel Resnick. 2007. All I Really Need to Know (About Creative Thinking) I Learned (by Studying How Children Learn) in Kindergarten. In *Proceedings of the 6th ACM SIGCHI Conference on Creativity &Amp; Cognition (C&C '07).* ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/1254960.1254961

Mitchel Resnick. 2014. Give P's a chance: Projects, Peers, Passion, Play. In *Constructionism and creativity: Proceedings of the Third International Constructionism Conference. Austrian Computer Society, Vienna.* 13–20.

Mitchel Resnick. 2017. *Lifelong Kindergarten: Cultivating Creativity through Projects, Passion, Peers, and Play.* MIT Press.

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

Eric S Roberts. 1995. Loop exits and structured programming: reopening the debate. In *ACM SIGCSE Bulletin,* Vol. 27. ACM, 268–272.

Jorma Sajaniemi. 2002. An empirical analysis of roles of variables in novice-level procedural programs. In *Human Centric Computing Languages and Environments, 2002. Proceedings. IEEE 2002 Symposia on.* IEEE, 37–39.

Katie Salen and Eric Zimmerman. 2003. Rules of Play - Game Design Fundamentals. *In The MIT Press Cambridge Massachusetts* (2003).

S. Schön, M. Ebner, and S. Kumar. 2014. Implications of new digital gadgets, fabrication tools and spaces for creative learning and teaching. *Special edition* (2014).

S. Sentance and A. Csizmadia. 2015. Teachers' perspectives on successful strategies for teaching Computing in school. *In IFIP TC3 Working Conference* (2015).

Swapneel Kalpesh Sheth, Jonathan Schaffer Bell, and Gail E. Kaiser. 2012. Increasing Student Engagement in Software Engineering with Gamification. *Columbia University Computer Science Technical Reports* (2012).

Teemu Sirkiä. 2012. *Recognizing Programming Misconceptions: An Analysis of the Data Collected from the UUhistle Program Simulation Tool.* Master's thesis. Department of Computer Science and Engineering, Aalto University.

W. Slany. 2014. Tinkering with Pocket Code, a Scratch-like programming app for your smartphone. *In Proceedings of Constructionism 2014* (2014).

Juha Sorva. 2013. Notional Machines and Introductory Programming Education. *Trans. Comput. Educ.* 13, 2, Article 8 (July 2013), 31 pages. https://doi.org/10.1145/2483710.2483713

Rivka Taub, Michal Armoni, and Mordechai Ben-Ari. 2012. CS Unplugged and Middle-School Students' Views, Attitudes, and Intentions Regarding CS. *TOCE* 12, 2 (2012), 8. https://doi.org/10.1145/2160547.2160551

Nath Tumlin. 2017. Teacher Configurable Coding Challenges for Block Languages. *In Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (2017).

David Weintrop and Uri Wilensky. 2015. To block or not to block, that is the question: students' perceptions of blocks-based programming. *IDC '15 Proceedings of the 14th International Conference on Interaction Design and Children* (2015).

Jeannette M. Wing. 2006. Computational Thinking. *Commun. ACM* 49, 3 (March 2006), 33–35. https://doi.org/10.1145/1118178.1118215

Jeannette M. Wing. 2008. Computational thinking and thinking about computing. *In Philosophical Transactions of the RoyalSociety* (2008).

Jeannette M. Wing. 2010. Computational Thinking: What and Why? *Link Magazine* (2010).

N. Wirth. 1993. Recollections About the Development of Pascal. *SIGPLAN Not.* 28, 3 (March 1993), 333–342. https://doi.org/10.1145/155360.155378