




RESEARCH ARTICLE

Open Access



Diogene-CT: tools and methodologies for teaching and learning coding

Giansalvatore Mecca , Donatello Santoro , Nazzareno Sileno and Enzo Veltri* 

*Correspondence:
enzo.veltri@unibas.it
Università degli studi della
Basilicata, Potenza, Italy

Abstract

Computational thinking is the capacity of undertaking a problem-solving process in various disciplines (including STEM, i.e. science, technology, engineering and mathematics) using distinctive techniques that are typical of computer science. It is nowadays considered a fundamental skill for students and citizens, that has the potential to affect future generations. At the roots of computational-thinking abilities stands the knowledge of computer programming, i.e. coding. With the goal of fostering computational thinking in young students, we address the challenging and open problem of using methods, tools and techniques to support teaching and learning of computer-programming skills in school curricula of the secondary grade and university courses. This problem is made complex by several factors. In fact, coding requires abstraction capabilities and complex cognitive skills such as procedural and conditional reasoning, planning, and analogical reasoning. In this paper, we introduce a new paradigm called ACME (“Code Animation by Evolved Metaphors”) that stands at the foundation of the Diogene-CT code visualization environment and methodology. We develop consistent visual metaphors for both procedural and object-oriented programming. Based on the metaphors, we introduce a playground architecture to support teaching and learning of the principles of coding. To the best of our knowledge, this is the first scalable code visualization tool using consistent metaphors in the field of the Computing Education Research (CER). It might be considered as a new kind of tools named as code visualization environments.

Keywords: Coding tools and techniques, Learning environments, Educational support, Computers and education

Introduction and motivation

The ability to face problem-solving challenges of STEM disciplines—i.e., science, technology, engineering and mathematics—is nowadays universally considered as a crucial skill (Siekmann and Korbel 2016; Marginson et al. 2013; Watt 2016). *Computational thinking* has been defined as the capacity of undertaking a problem-solving process in various disciplines using techniques that are distinctive of computer science (Wing 2006). At the core of these techniques stand the skill of computer programming, commonly called *coding*. We may therefore say that coding abilities are a fundamental building block of any computational-thinking based approach to teaching. In view of this, this paper concentrates on the important problem of developing new methods and tools

to simplify the teaching and learning of coding. This is a complex problem, that poses several important challenges. Among these, the prominent one is that acquiring coding skills requires abstraction capabilities. In fact, it has been argued that learning to program is difficult (Qian and Lehman 2017), and that so far “three decades of active research on the teaching of introductory programming have had limited effect on classroom practice” (Pears et al. 2007). The main reason for this is that several of the typical coding tasks require capacity of abstraction:

- Data structure design: the first one is the conceptualization of the data of the problem at hand, and the choice of a proper representation in terms of the primitives and data structures.
- Program construction and interpretation: the second one is concerned with the development of algorithms and of the actual source code. A crucial requirement to carry this step is the ability to construct a model of the code execution, i.e. of the semantics of the program. The execution of a program consists in a sequence of actions executed by the machine. From the perspective of the developer, this happens largely within a black box, and therefore at first it is difficult to understand for novice programmers. To become good programmers, students need to develop the ability to make a mental model of the semantics of their programs.
- Test and interpretation of results: the third task, once the code has been designed, consists in testing the code in order to verify its correctness, possibly identifying and removing bugs. Debugging is largely related to program interpretation—students must develop the skills to identify in which way inconsistent results or errors at runtime are related to the program source code.

A number of studies (Dehnadi 2006; Ma et al. 2007) have investigated the complexity of learning to perform these tasks. While there are largely different points of view, all studies concur in pointing out that this is a difficult problem with an extremely high learning risk, as discussed in the next section.

Previous approaches and their limitations

Many different approaches have been proposed throughout the years to facilitate the task of teaching computer programming (Bers et al. 2014; Marín et al. 2018) and some efforts have been made in classifying different tool approaches to make easier the learning of programming (Kelleher and Pausch 2005). We first briefly mention that methodologies for teaching coding can be seen as specific instances of the more general class of methodological approaches to education (Grossman et al. 2009), including the classical theories of *behaviourism*, *cognitivism* and *constructivism*.

For the purpose of our analysis especially interesting are *robot-based programming platforms* (e.g., Lego Mindstorms). These have proven to be quite successful, especially among children. In these platforms, each student has a physical robot, often assembled by her/himself. Using a personal computer and a visual programming language, students can issue sequences of commands to trigger actions of the robot (i.e., rotate the wheels of x degrees, turn to the left of y degrees, activate the infrared sensor, and so on), and see the results. This approach has a great advantage, namely immediate feedback. In

essence, students visually perceive interactively the effect of each instruction they add to their code. It is easy to see that this greatly simplifies program interpretation, one of the main problems as discussed above, and eases the abstraction barrier to coding.

The success of the adoption of physical—i.e., hardware—robots for the purpose of teaching coding has inspired a class of software approaches, called *microworlds*. These can be seen as a special category of a larger class of *software-visualization environments*, that also includes *algorithm-visualization tools* and *program-visualization tools* (Sorva et al. 2013). *Microworlds* are programming environments centered around robot-based metaphors (Bers et al. 2014)—have been widely used for the purpose of introducing children to programming. The microworlds are based on constructivist learning theory, that predicates that learning must rely on building things that are tangible and shareable (Ackermann et al. 2009). The original Lego computing platform (Harvey 1997) was based on nothing more than on a turtle-like programmable robot. The cMinds project (Tsalapatas et al. 2012), has also used robots to visualize algorithms to primary-school students.

The goal of *algorithm-visualization tools* (Hundhausen et al. 2002; Urquiza-Fuentes and Velázquez-Iturbide 2009) is to provide a graphical representation of the inner structure of an algorithm, e.g., binary search or select sort. Students may interact with this graphical representation and manipulate it. In this, these approaches may be seen as an instance of constructionism theories. Notice that most of the recent platforms for teaching coding, e.g., code.org¹ or Scratch², use a combination of microworlds, i.e., coding tasks that are based on the programmable-robot analogy, ranging from mice and mazes to Angry Birds, and algorithm-visualization tools, under the form of block-based programming. Block-based programming tools like provide building blocks corresponding to basic programming constructs and allow students to assemble them using drag and drop.

Finally, the purpose of program-visualization tools (Sorva et al. 2013; Kölling et al. 2003; Moreno et al. 2004) is to provide a simplified, graphical debugger that allows to interact with the runtime execution of code. They are often integrated into a custom educational Integrated Development Environment (IDE) and provide a complete working environment for the novice programmer.

Recently, some tools have been developed to facilitate the learning of computer programming combining block-based programming with text-based programming. Greenfoot (Kölling 2010) is a tool that can be installed and permits to learn object-oriented programming. It allows creating “actors” which live in “worlds” such to build videogames. These “objects” can be created through a Graphical User Interface (GUI) and then can be programmed with real Java code. Code Genie (Jawad et al. 2018) is a web tool, where the GUI is developed to mimic a classical IDE. With Code Genie is possible to learn JavaScript: users learn how to call functions with the right parameters to paint images, but also learn how to use conditional statements and loops. The GUI helps them to write code just by clicking on a button and then modifying the parameters or they can

¹ <http://www.code.org>.

² <https://scratch.mit.edu/>.

write directly the source code. Code Genie uses a gamification strategy based on “share” and “like” such to create a genuine competition among the users.

Among these approaches, robot-based tools—both hardware and software ones—have proven to be the most successful. Experience tells that these approaches are effective in the early stages of learning, due to two key ideas:

- the idea of providing immediate visual feedback through an actual movement of the robot for each instruction that is added to the code (e.g., “move the robot one step to the left”);
- the idea of making it more explicit what is the ultimate goal of a program, so that coding is more concretely perceived as a sequence of actions that brings the system from an initial state to a desired target state (e.g., “bring the animated character outside of the maze”).

Despite these powerful intuitions, these approaches fall short as soon as we move to more ambitious learning goals. In fact, hardware-based platforms are inherently limited in scope, since they can only be used to command robots, and are not suitable to write general-purpose programs. Software-based microworlds are usually confined to toy examples, and do not capture the complexity of an industrial-level programming language as Python or Java.

It is also important K-12 students are “digital natives”, since they were born after Web and mobile services had become integral part of our everyday life. They are accustomed to using sophisticated apps in their everyday life. To make computational experiences interesting to them, there is a strong need that coding is based on current technology, so that they are capable to produce artifacts that are comparable to those they manipulate daily. This, in turn, requires to make them familiar not only with basic coding techniques, but also with more advanced concepts, like object-oriented programming, graphical user interfaces, and, to some extent, also Web and mobile applications.

Contributions

This paper makes several important contributions towards the goal of fostering the adoption of innovative tools and methods to teach computer programming in secondary schools and university 1st-year courses:

- It develops a set of consistent metaphors to introduce the concepts of programming. By adopting the metaphors, instructors can work with students through the concepts of programming with a very concrete, visually perceivable counterpart to all of the inner workings of a programming language, in order to ease the approach of students to coding, reduce their cognitive load and the “black box” effect associated with source code.
- Differently from other proposals that have similar goals, the metaphors consider both introductory coding, i.e., procedural-programming concepts like variables, assignments, control structures and so on, i.e., programming-in-the-small, and object-oriented programming concepts like components, classes, objects, responsibilities and messages, i.e., programming-in-the-medium. This is an important feature of our

method, that can be adopted as a holistic approach to build a complex coding curriculum.

- In addition to this methodology, we also develop a set of executable tools. The basic building blocks for our toolset are represented by animation actions and animation programs. The metaphors are translated into a library of animation actions over an animation scene. The scene contains the main actors of the animation—like the mechanical arm or the calculator, and their possible interaction. Each animation action animates one basic operation carried out by the processor or virtual machine during the execution of a piece of code, as a set of visible actions of the actors on the scene. In this way, each execution of a target program becomes an animation program on the scene.
- We develop an engine for animation-program execution. This allows to construct and run animation programs offline, or in detached mode. In this mode, the animation program is not directly attached to the target-program source code, and therefore it does not depend directly on the target programming language. This has the advantage of leaving instructors with ample flexibility in choosing programming languages, possibly allowing them to mix and compare programs in different languages.
- In addition to offline mode, we also develop a complex development environment to run animations in online or attached mode. The development environment supports the full cycle of Java source code development and execution, with the important addition of automatically generating animation actions and animation programs for the target code. To do this, we develop a complex infrastructure that represents one of the main technical contributions of this work.

Based on these ideas, we believe Diogene-CT represents the first of new class of tools, that we may call *code-animation tools*, that tries to combine program-visualization tools with microworlds, thus providing the advantages of both while removing their limitations.

The Diogene-CT methodology and tools are the product of a long experience with innovative methods for teaching programming conducted over the last 15 years within introductory programming courses at the University. In the last couple of years, the adoption of animation programs has been successfully tested both the “Procedural Programming” course and in the “Object-Oriented Programming” course. This allowed us to gain precious insights about their strength and limitations, and to refine and significantly improve the method.

The Diogene-CT approach

We intend to exploit the advantages of earlier approaches within our proposal, while at the same time removing the limitation connected to the scope of the programming platform. To do this, we maintain the two main intuitions of educational robotics and microworlds, i.e., (a) immediate visualization of the effect of instruction and (b) program as a way to bring the computer to a visually clear target state. At the same time,

we deeply transform them, by constructing our metaphor³ not based on some fictional microworld, or some external physical robot, but rather on the actual workings of a programming language.

We have the ambitious goal of developing a methodology and a set of tools capable of providing visual feedback for arbitrary programs written using programming languages of state-of-the-art platforms (primarily Java), and both for introductory procedural-programming, and more complex object-oriented programming. To the best of our knowledge, this is the first proposal in this direction. Achieving the goals we have set is far from trivial and we shall proceed in a modular fashion. Modularity is a keyword for the entire project. The project is modular in several respects, as follows:

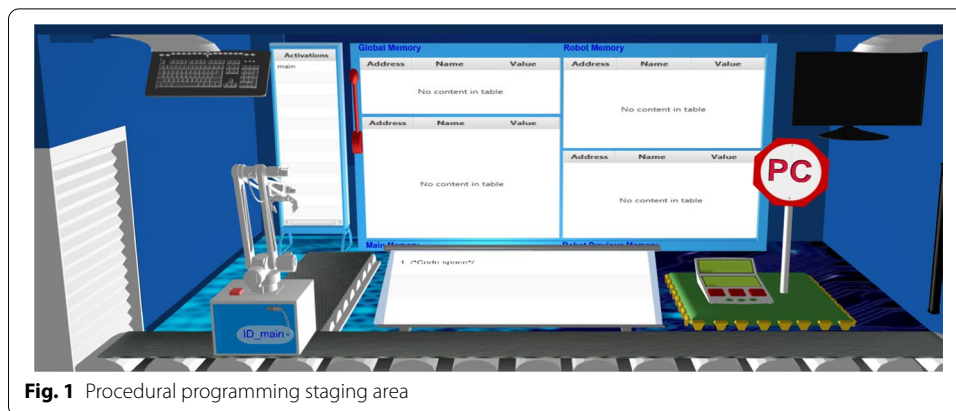
- To start, the methodology is based on two different but interrelated metaphors. The first and basic one is centered around the main constructs of procedural programming, like storing value by a variable or evaluating a mathematical expression. The second and more advanced one is constructed around the principles of object-oriented programming, like constructing objects of exchanging messages for method invocation. Also, the tools developed in the project allow to seamlessly deep dive from one metaphor into the other. This richness of the method allows students to gain a better understanding of the actual operational aspects of programming, by reducing at the same time their cognitive overload.
- In addition to this, the set of tools developed within the project supports different usage scenarios. In fact, Diogene-CT is both a teaching methodology and a set of executable tools. Teachers may decide to only adopt the teaching methodology as a basis for their lectures, or also the tools that come with it.
- Also, the animator tool may be run both in *offline* (or detached) and *online* (or attached) mode. In offline mode the teacher typically develops some code, then picks up a usage scenario for it—e.g., calculating the size of the circle given its radius, with a radius of 3 cm—and uses the Diogene-CT animator to give life to the execution of that particular usage scenario, in an offline fashion for the actual execution of the code. This is the simplest use of the animator. Much more challenging is the online mode, in which the Diogene-CT animator is provided with a piece of source code written in Java and allows the student to run the code and interact with it while animating all events triggered by the code.

These ideas are described in the following sections.

The mechanical arm metaphor for procedural programming

As a first tool, we introduce a metaphor for introductory procedural programming or programming-in-the-small. The aim is to teach the basic concepts of any programming language, like variables, assignments, control structures and so on. More specifically, we develop a metaphor for the execution of code instructions based on the use of mechanical arms. A typical scene generated by the animator for this metaphor is shown in Fig. 1.

³ By “metaphor” we intend a representation that describes a concept by referring to something that has similarities to that concept.



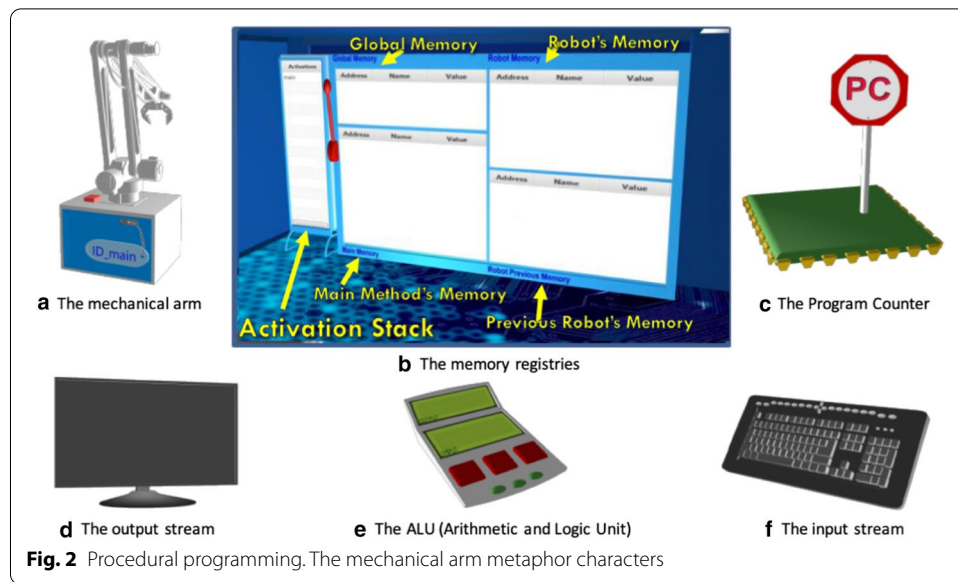
The main elements of the metaphor are as follows:

- The Random Access Memory (RAM) is exemplified as a large spreadsheet of cells that can be named and can store values (to represent variables).
- The Arithmetic Logic Unit (ALU) is depicted as a calculator capable of performing the typical computations required in numerical and Boolean algebra.
- The standard input and standard output are represented by pipes connected to the keyboard and console screen, respectively. Animations show streams of characters flowing through these pipes when they are activated.
- Most important, the actions performed by the processor are depicted as movements of a group of mechanical arms. The mechanical arm represents an execution context, typically a function, and is responsible for animating operations carried out by the processor or virtual machine, like: (a) changing the contents of the memory spreadsheet; (b) operating the calculator to perform computations; (c) activating the pipes to the console or from the keyboard.
- Finally, to clarify the importance of the control flow and the role of control structures, a lollipop sign (resembling the ones used by street policemen or crossing guards) indicates at any moment of the animation the number of the next instruction that the mechanical arm is supposed to execute.

There are more elements to the animation (especially related to procedure/function invocation and parameter passing), as discussed in the following sections, but what we have discussed so far should be enough to give an intuition of how code instructions are animated: each instruction causes a set of actions by the mechanical arm on the scenes, guided by the lollipop. Students can see all of the intermediate states of the program animation, and this removes the “black box” effect associated with the traditional execution of code by the machine, and makes it more transparent to students, adding the immediate feedback effect that is typical of robots and microworlds. At the same time, it is highly flexible and can scale from simple, toy examples to fairly complex computations.

Scene of the mechanical-arm metaphor

The mechanical arm metaphor scene is represented in Fig. 1. The main stage where the metaphor actors play together is composed of background elements that represent



a typical assembly line. The conveyor belt is a sort of guide for wired puppets. The robot arms move to the left and to the right, forward and backward thanks to the conveyor belt movements. The source code projector shows how the desired source code lines are linked to the showed animations. The left portcullis will raise when the passing argument animation is executed and a new mechanical robot will take the place of the running one (a new method will be executed). The metaphor characters are: the executor robot, the memory registries, the lollipop sign, the calculator, the keyboard and the screen. The main character is the executor robot in Fig. 2a that represents the running method. The label on the plate of its body contains the name of the method. It interacts with the other scene objects and it executes each code line included in its body. The memory registries (Fig. 2b) are the components that have the shape of a huge blackboard. The blackboard is divided into five zones:

- the activation stack is the place where the runtime environment of the program keeps track of all the functions that have been called. It contains the list of method names that have been activated and that are waiting for their termination;
- the global memory panel contains all the global variables and constants instantiated in memory;
- the main method memory panel contains all the variables and constants of the main method that have been instantiated in memory;
- the robot memory contains the variables, the constants and the parameters received from a generic submodule and maintains their state during the execution;
- the previous robot memory contains the variables and the constants of a submodule whose execution is not ended. It also contains the parameters passed to another module. It is useful to help learners in studying the passing arguments behavior and the state of the memory. When necessary, it updates the state of the variables during the execution of the submodules that are using the parameters by

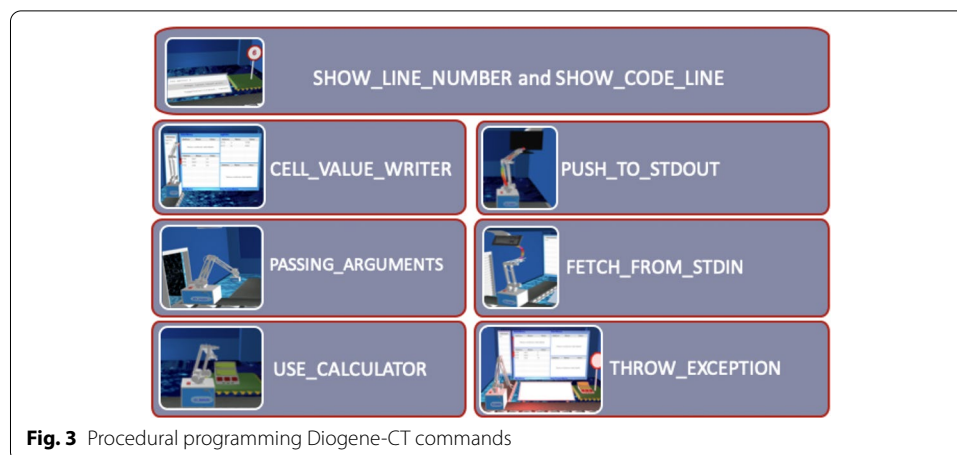


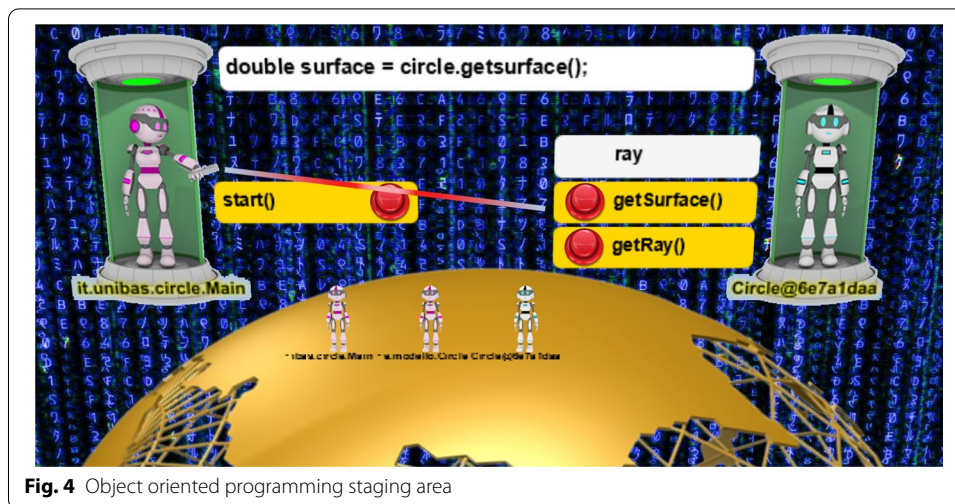
Fig. 3 Procedural programming Diogene-CT commands

reference. The row of each panel contains an element (variable or constant) and information about its address, name and value.

The lollipop sign in Fig. 2c contains the running source code line number. It helps the learner keep track of the state of the program execution and indicates the next instruction that will be executed by the mechanical arm. The calculator Fig. 2d is used by the mechanical arms to evaluate expressions and do calculations. It is useful to understand computer logic in evaluating the expression avoiding the typical beginners' errors. The keyboard Fig. 2e indicates the user input. The robot arm acquires the user's data flow and uses its content (for instance it creates new variables in the memory zone). The screen Fig. 2f represents the user output. The robot arm prints the user's data flow on the screen display.

Commands of the mechanical-arm metaphor

Figure 3 represents the commands needed to animate all the mechanical arm metaphor aspects. The execution of a program consists of a sequence of actions carried out by the machine. The semantics of the program is expressed through the following commands: show line number and show code line, cell value writer, passing arguments, use calculator, throw an exception, fetch from stdin and push to stdout. The `SHOW_LINE_NUMBER` and `SHOW_CODE_LINE` commands are needed to display the source code row number using animation on the top of the lollipop sign. They also display the source code lines inside the projector source code panel. Then, in the `CELL_VALUE_WRITER` the mechanical arm reaches the memory registries. It pulls the lever injecting new memory rows into the destination tables. By the `PASSING_ARGUMENTS` the mechanical arm reaches the left portcullis that stands up and a new mechanical robot takes the place of the running one (a new method will be executed). If the method arguments are passed, the animation illustrates the parameters migration from the starting memory registry to the destination one. The memory registries are updated consequently. The mechanical arm reaches the calculator and presses the buttons simulating the calculation or the expression evaluation with the `USE_CALCULATOR` command. The calculator displays the result on its screen and, if necessary, the cell value writer animation is executed to

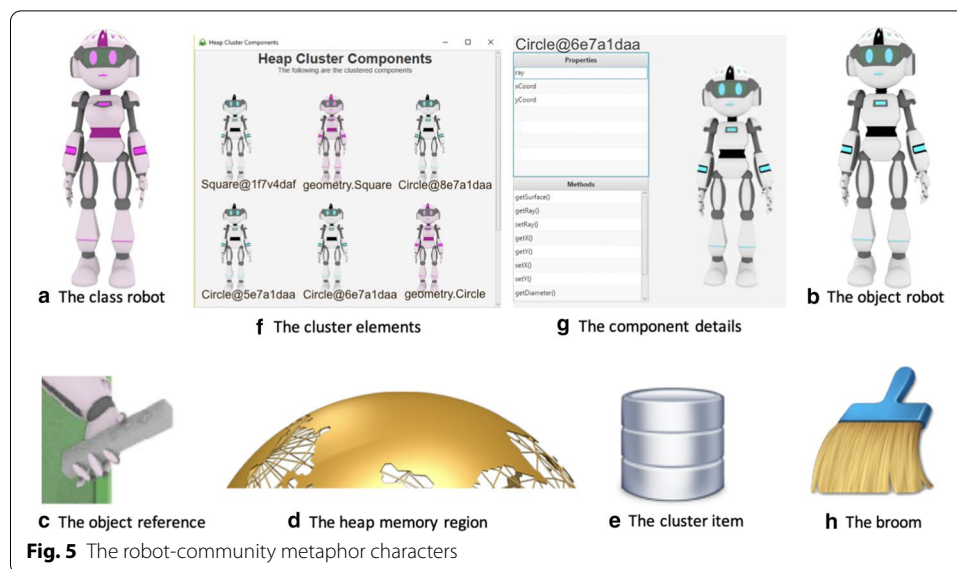


store the results. By the `THROW_EXCEPTION` command a red blinking light on the stage indicates the occurred software exceptions. The mechanical arm simulates alarm situations going up and down on the stage. In the `FETCH_FROM_STDIN` the robot moves toward the keyboard and fetches the byte stream. If it is necessary, the cell value writer animation will be executed to store the input elements. Viceversa, in the `PUSH_TO_STDOUT` the robot moves toward the screen and prints the output stream on it. Notice that these commands are typically used by instructors, rather than by students, to generate animations in detached mode.

The robot-community metaphor for object-oriented programming

As we mentioned, the ultimate goal of the project is to bring students to the level of mastering advanced technologies, and primarily object-oriented programming, the key to all state-of-the-art programming platforms. Experience tells that learning object-oriented method is challenging, even for students that have already acquired basic programming skills. Basic programming—or programming-in-the-small—deals with learning the main instructions offered by the programming language, the techniques to represent data using the program variables, and the process of code development and execution. This is what our mechanical-arm metaphor is focused on. In addition to this, object-oriented programming requires that students acquire proper knowledge of programming-in-the-medium techniques, that is, how to organize the code of an application as a set of components—classes and objects, how to assign responsibilities to components, and how to exchange messages among them. In the end, each component will be described in terms of basic instructions, but the choice and design of components is a crucial design step per se. We develop a second robot-like metaphor to handle this transition. A typical scene of this second kind is reported in Fig. 4.

More specifically, similarly to what we did above, we design a metaphor of object-oriented applications as communities of cooperating robot, each robot being a component (class of objects); the typical concepts of object-oriented programming (reference, message, method, package, binding etc.) will be explained in terms of this metaphor (e.g.: a reference to an object is a remote control for the object; a message between objects



is a message exchanged between robots etc.). The Diogene-CT animator is designed to tightly integrate the two metaphors, in such a way that students may animate a piece of object-oriented code to have a high-level view of the cooperative behavior or objects, and at any moment zoom into the behavior of each single instruction by switching to the procedural, mechanical-arm animation to have a low-level view of the execution of the actual instructions.

In the rest of this section, we will introduce the metaphor for object-oriented programming (OOP) and we will go on illustrating the scene and the commands of the robot-community metaphor. Our metaphor shows the execution of code instructions based on the communities' interactions of the cooperating robots. The robot-community metaphor is strictly tied to the object-oriented computer programming concepts. It uses similar real-world elements to easily explain the typical concepts of object-oriented programming. It also benefits from the mechanical arm metaphor interaction so that the learner can automatically zoom-in procedural programming details and zoom-out to the highest object-oriented interaction level.

Scene of the robot-community metaphor

The robot-community metaphor scene is represented in Fig. 4. The main stage where the metaphor actors play together is composed of background elements that represent the virtual world of robots. The communities of cooperating robots are the places where classes and objects are picked up and materialized in the holographic tubes. The white rectangle on top is used to show the single running source code line. The object-oriented metaphor characters are: the robot girl, the robot boy, the remote control, the heap, the cluster of components and the broom. The first character is the robot girl. Every time a class constructor or a static method is executed the avatar that symbolizes the class Fig. 5a is materialized into the holographic tube. It semantically represents a generic OOP class and its animations tend to illustrate the class behaviors: static or objects method invocations, properties usage, and constructors' execution. Thus, the robot boy

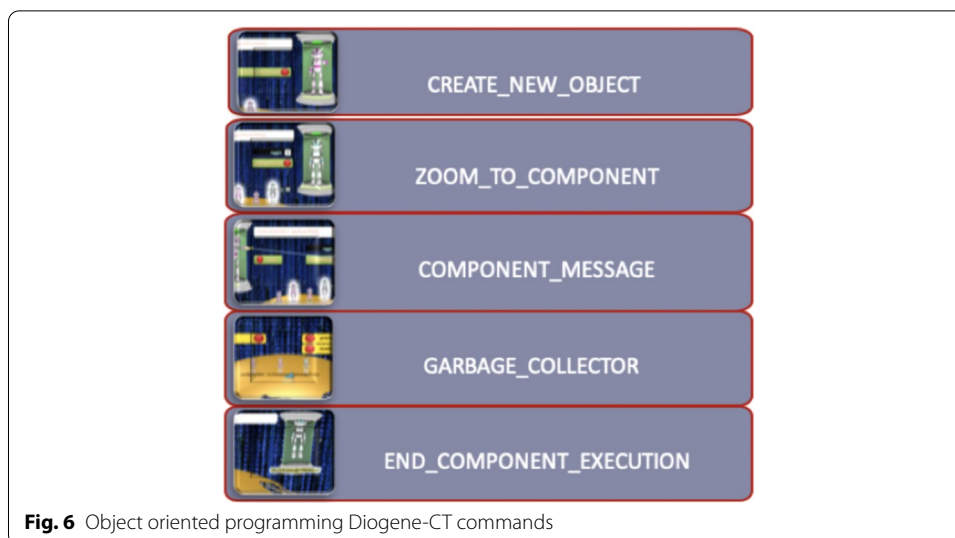


Fig. 6 Object oriented programming Diogene-CT commands

is generated by the robot girl. Every time a class instance is instantiated or an object reference is used, the avatar representing the object Fig. 5b is materialized into the holographic tube. It semantically represents a generic OOP object and its animations tend to illustrate the object behaviors (i.e.: static or object method invocations and properties usage). The use of the remote control semantically represents the object reference Fig. 5c and the exchange of messages. The laser light reaches the opposite part of the scene lighting up methods or properties of the referenced element. The remote control helps learners understand the basic elements of object-oriented programming. The fundamental mechanism of the program execution is the exchange of messages among components. The heap is the element of the metaphor that represents the town where robots live and interact in Fig. 5d. When a class is first used or every time a new object is instantiated, their little avatar is positioned on the globe. The existence of the little elements allows learners to explore the state of the components and to keep track of the changes. The cluster item Fig. 5e groups the components to help manage the virtual space. When more than a fixed number of elements is disposed of simultaneously they will be grouped and then accessible clicking on the cluster item icon. The content of the cluster can be displayed using a pop-up window Fig. 5f and the details of each element Fig. 5g can be further explored. The use of the broom Fig. 5h semantically illustrates the garbage collector execution.

Commands of the robot-community metaphor

Figure 6 contains the commands needed to animate all the robot-community metaphor aspects. The execution of a program consists of a sequence of actions executed by the machine. The semantics of the program is expressed through the following commands: create a new object, zoom to the component, component message, garbage collector and the end component execution. In the command `CREATE_NEW_OBJECT` the robot girl hologram appears inside a tube on the scene. It presses the button on its chest simulating in this way the invocation of a specific constructor method. The factory class disappears and is replaced by the new robot boy instance; the remote control representing the

object reference is then materialized into the hand of the component that started the process. The `ZOOM_TO_COMPONENT` command simply materializes the selected component into the cylinder. In the `COMPONENT_MESSAGE` the concept of message passing comes alive. The fundamental mechanism of the program execution is the exchange of messages among components. The animation shows a laser light that reaches the opposite part of the scene lighting up the methods or properties of the referenced element. If the target component represents an object, then the remote control is animated otherwise the index finger indicates the target and a red light will be emitted starting from this one. In the `GARBAGE_COLLECTOR` the use of the broom on the heap memory region allows to collect the components, to remove and to clean up the community from the objects whose remote control is no longer available. The `END_COMPONENT_EXECUTION` command simply makes components disappear from the cylinder they are placed in.

Gamification

To further highlight the flexibility and richness of the method, we mention that one additional important feature of our metaphor-based approach is that it naturally lends itself to gamification. Gamification is the technique of teaching problem-solving using games, and is already proven to be effective in computer programming (da Silva et al. 2015; Bayliss 2009).

In the Diogene-CT approach, the goal of a game is to write a computer program that solves a problem, e.g., a math problem or a simple simulation of a scientific phenomenon. The game consists of designing the target state of the scene, i.e., the one in which all results have been computed, and asking students to physically enact the actual animation—either procedural or object-oriented. Students can work in group. Each of them is responsible of performing the actions of an element of the scene—say, the mechanical arm or the lollipop, or the calculator, or to conduct the actions of a component robot.

Offline/detached mode

At the core of the project stands the Diogene-CT animator. This is the module responsible for generating the actual visual animation of the code. The animator relies on a library of actions:

- The action library for procedural programming includes all primitives needed to animate the moves of the mechanical arm, e.g., to change the status of the memory, to use the calculator, to access the keyboard and so on.
- The action library for object-oriented programming includes the primitives to create new objects, send a message to a class, send a message to an object and so on.

Actions can be composed to form animation programs. These are complex animations involving a sequence of actions, typically corresponding to the execution of a piece of actual source code, that we call the target program. The main intuition behind the use of Diogene-CT is to generate an animation program for each execution of the target program. This can be done in two different modes, depending on whether the animator has access to the source code or not. As we discussed above, the simplest usage of

the Diogene-CT animator is in offline, or detached mode. In this mode, the animator runs separately from the target program, i.e., it does not require to have access to the actual source code. This has advantages and limitations:

- Since it is not tied to the actual source code of the program, the animation can be built for any language or piece of code, without technological constraints. This is a clear advantage in terms of generality of the approach.
- At the same time, since the animator does not know what code it is animating, the teacher needs to code animation programs by hand, by composing a sequence of actions. This, in turn, requires fixing specific application usage scenarios.

Also in this case, offline mode is supposed to be used mainly by teachers to create animations for their lectures. Let us discuss a practical example. We consider the problem of computing the area of the circle based on the size of the radius. Suppose we write a piece of MatLab code like in Listing 1.

Listing 1: circle.m

```
radius = 3;
circle = 2 * 3.14 * radius;
fprintf('circle = %f',
       circle);
```

Listing 2: circle.c

```
#include <stdio.h>
int main() {
    float radius = 3;
    float circle = 2 * 3.14 *
        radius;
    printf("circle = %f", circle
        );
}
```

Notice that the only visible output of this piece of code in the MatLab environment is the following string:

```
circle = 18.84
```

It is possible to animate this code by an animation program composed of the following sequence of actions (to simplify things we skip line codes shown by the lollipop and we use a pseudocode):

```
Arm allocates variable radius in memory
Arm stores 3 in variable radius
Arm allocates variable circle in memory
Arm uses calculator to compute 2 * 3.14 * 3 //(result: 18.84)
Arm stores result in variable circle
Arm sends to the standard output string circle = 18.84
```

This animator program needs to be built by hand, stored in a file with the appropriate format, and fed to the animator. The animation program is independent from the programming language, i.e., it is a perfectly fine animation for the equivalent piece of C code listed in Listing 2.

Similarly, this could be used to teach the workings of a simple piece of FORTRAN or Python (procedural) code, which is an important feature of our approach. On the other side, let's now consider a slightly different version of the C code, as in Listing 3.

Listing 3: circle2.c

```
#include <stdio.h>
int main() {
    float radius, circle;
    scanf("%f", &radius);
    if (radius > 0) {
        circle = 2 * 3.14 * radius;
        printf("circle = %f", circle);
    } else {
        printf("Incorrect value");
    }
}
```

It can be seen that now the value of variable `radius` is taken from the standard input, rather than being hardwired to 3 as it was before. Now, the animation program is still a valid animation for this C code, but only limited to one specific usage scenario, i.e., the one in which the user interactively provides 3 as the size of the radius. As soon as we intend to explore a different scenario—for example what happens when the user provides 0 as a value for the radius—we need to build a different animation program, as follows:

```
Arm allocates variable radius in memory
Arm allocates variable circle in memory
Arm reads from standard input the value of variable radius
Arm stores 0 in variable radius
Arm uses calculator to compute 0 > 0 //(result: false)
Arm sends to the standard output string Incorrect value
```

A similar example can be built for object-oriented programming, using for instance Swift and Java as programming languages. Limits and opportunities of the offline mode should be clear at this point: it is very flexible since it can be used to animate a wide variety of programming languages. At the same time, the construction of animation programs may become a labor-intensive and error-prone activity. Note, in fact, that any change to the source code of the target program requires to maintain the animation programs associated with it (one for each usage scenario).

Online/attached mode

As an alternative, Diogene-CT provides the online or attached animation mode. In this mode, the animator works as a fully-fledged application-development environment, in the sense that it has access to the actual source code. The code can be edited within the environment and run. During execution, the animator captures all events triggered by

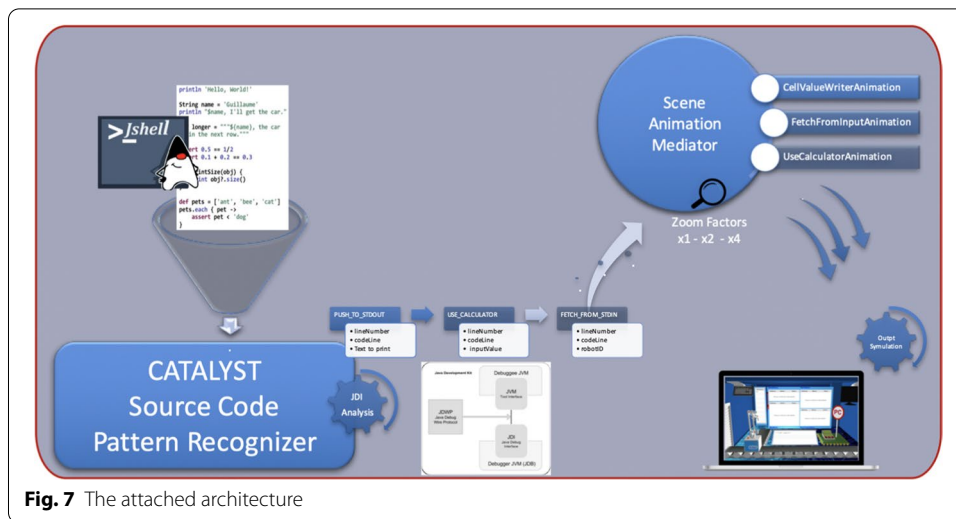


Fig. 7 The attached architecture

the target program and animates them on the fly, i.e., it dynamically builds the corresponding animation program. This mode of execution is dual to the offline one. It does not require to write any animation code—actions and animations are automatically generated based on the interactive execution of the target program. Similarly, it can handle all usage scenarios, since it adapts to the actual behavior of the target source code. On the other side, it should be apparent that this requires a very strong coupling with the programming language and the relative software development kit. This mode requires the development of a suitable driver for each programming language. Currently, Diogene-CT only supports JShell, and therefore the online mode can only be used to animate Java code—both procedural and object-oriented. The development environment lets the learner write arbitrary source code and see its dynamic animations.

The added value of this mode is expressed by the automatic generation of the animation actions for the target code and the consequent animation program execution. To do this, we have developed a complex infrastructure that represents one of the main technical contributions of this work. The attached architecture (Fig. 7) shares The Scene Animations Mediator component with the detached architecture.

The idea to develop a module called Catalyst is borrowed from mechanical engineering. The principle was to elaborate a generic source code input and to produce a well-defined output: the Diogene-CT commands. The Catalyst is designed to be as general as possible. At this moment, the implementation is focused on the JShell programming language but it can be replaced with another implementation that could elaborate whatever programming language. During the execution, the Catalyst captures all events triggered by the target program and animates them on the fly (i.e., it dynamically builds the corresponding Diogene-CT animation commands). The events are captured by the debugger logic. In effect, the debug information—that describes the source code—is used to generate animations. In detail, the catalyst has to transform the source code into commands output and to keep track of the complete source code execution intercepting the target key points. In this way, dynamic breakpoints are placed to gather all the necessary parameters to build the commands that will be sent to the mediator.

Experiments

As we mentioned, the methodology and tools discussed in this paper have been developed throughout the years within programming courses from the bachelor degree in Computer Science at University. More specifically, the adoption of animation programs has been systematically adopted in two courses.

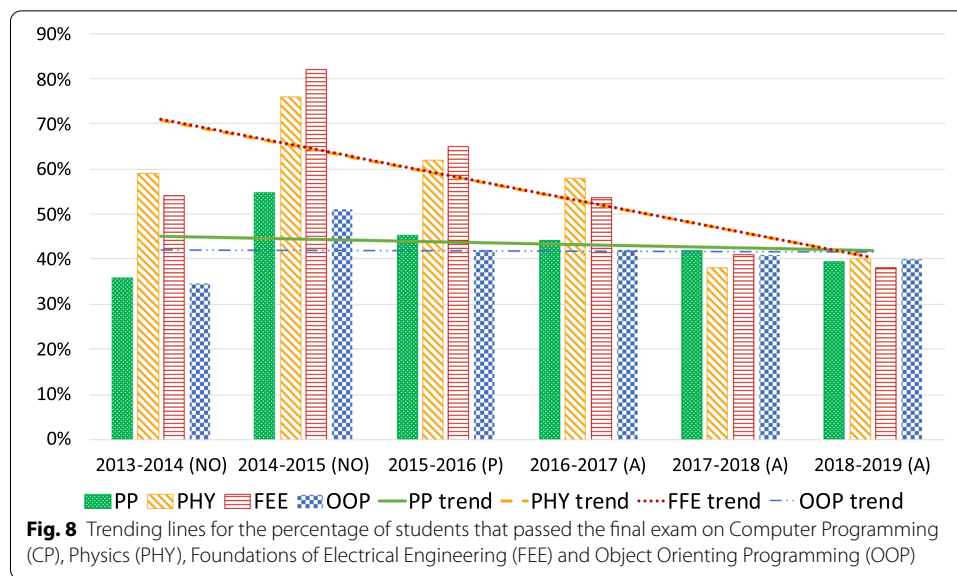
- i. The *Procedural Programming course (PP)*, i.e., Coding 101, the introductory course to programming (1st year of the Computer Science curriculum). The PP course deals with introductory procedural-programming techniques in several languages, primarily C/C++, and secondarily MatLab, covering the basics, control structures, functions and modular programming, arrays and records, pointers, and an introduction to data structures, with a focus on the list data structure. The final exam is composed of two parts: a written test with 30 multiple-choice questions to be completed in 40 min; a practical programming test requiring to implement a simple application with a console-based user interface that implements one or more algorithms on a collection—i.e., a list—of records. This is a 1st-year course, and we assume that most of the students do not have any programming background.
- ii. The *Object-Oriented Programming course (OOP)*, i.e., the introductory course to object-orientation (2nd year of the curriculum). The OOP course introduces classes, objects, references, methods and messages, application layers, exceptions and regression tests in several languages, primarily Java, and secondarily C#. The final exam is similar in structure to the one of the PP course: a written test with 30 multiple-choice questions to be completed in 40 min; a practical programming test requiring to implement a medium-complexity application logic and the relative regression tests. Students need to choose the most appropriate data structure, i.e., list, set or map.

In this section we report some experimental results gathered in this framework. We discuss two experiments: (i) first, an analysis of course-completion stats, in order to assess the actual benefits of our approach in terms of learning outcomes; and (ii) second, a user study conducted among a group of students to gain insights about their perception of the strengths and limitations of the method.

Effectiveness on course-completion rates

As a first set of experimental results, we report course-completion rates for different groups of students. We compared our test group, students that have been taught coding in PP and OOP with the help of the Diogene-CT method and tools, to a control group of students that didn't. In order to do this, we measured the percentage of students that passed the final exam for the PP and OOP course from academic year 2013–2014 to 2018–2019. While instructors and course contents remained largely unchanged, the teaching methodology has evolved significantly through these years:

- In 2013–2014 and 2014–2015 no Diogene-CT techniques were employed within the course yet. We shall make reference to these cohorts as **NO**.



- The cohort of 2015–2016 was a transitory one, in the sense that the robotic-arm metaphor was adopted to support the introduction of programming concepts, but the Diogene-CT toolset was not production-ready at the time, and therefore more rudimental tools were used. We shall make reference to these cohorts as **P**, for “partial”.
- Finally, the methodology and the toolset were fully adopted in 2016–2017 and subsequent years. We shall make reference to these cohorts as **A**, since all of the toolset were used.

Course-completion ratios are shown in Fig. 8 with the column PP and OOP. More specifically, we report the percentage of students enrolled in the course that successfully passed the final exam at the end of the academic year. Since these data were gathered within real everyday academic experience, we couldn’t organize a canonical control group. Our intuition, however, is to use the results of the **NO** cohorts as controls for the **A** (and **P**) cohorts.

This poses several challenges, as it is known that knowledge, attitude and maturity of students may significantly vary throughout the years, and this unavoidably affect performance. In fact, our empirical and rather coarse observation was that the overall quality of students has somehow decreased during these years. To make this intuition more precise, we selected two other courses from our Computer Science curriculum that also had a stable set of contents and team of instructors from 2013 to date, namely Physics (PHY), 1st year, and Foundations of Electrical Engineering (FEE), 2nd year. Both these courses adopt a rather traditional teaching methodology, mostly based on theory-based frontal lessons with some additional exercises, and written tests.

To elaborate on the results in Fig. 8, in addition to PP and OOP, we also report the percentage of students that passed the final test for PHY and FEE. We also report trending lines for each data series.

It is easy to see from the chart that our intuition is confirmed, i.e., both in the case of the PHY course and of the FEE course the percentage of enrolled students that passed the final exam has almost consistently decreased in the years (see the PHY and FEE trending lines). The 2013–2014 represents an exception with respect to this trend, in the sense that the overall performance of the students was particularly poor.

Interestingly, this negative trend is not observable in both PP and OOP courses (PP and OOP trending lines are quite horizontal), again with the exception of 2013–2014. There were small decreases, but overall the percentage of students that passed the exams remained fairly stable, as confirmed by the trending line.

In a framework of generally decreasing success rates, as shown by the PHY and FEE results, we believe this is a clear indication of the fact that the introduction of innovative tools as the ones described in this paper positively impacted learning outcomes. This confirms that the Diogene-CT method and toolset can be an effective support in teaching coding.

Effectiveness on learning outcomes

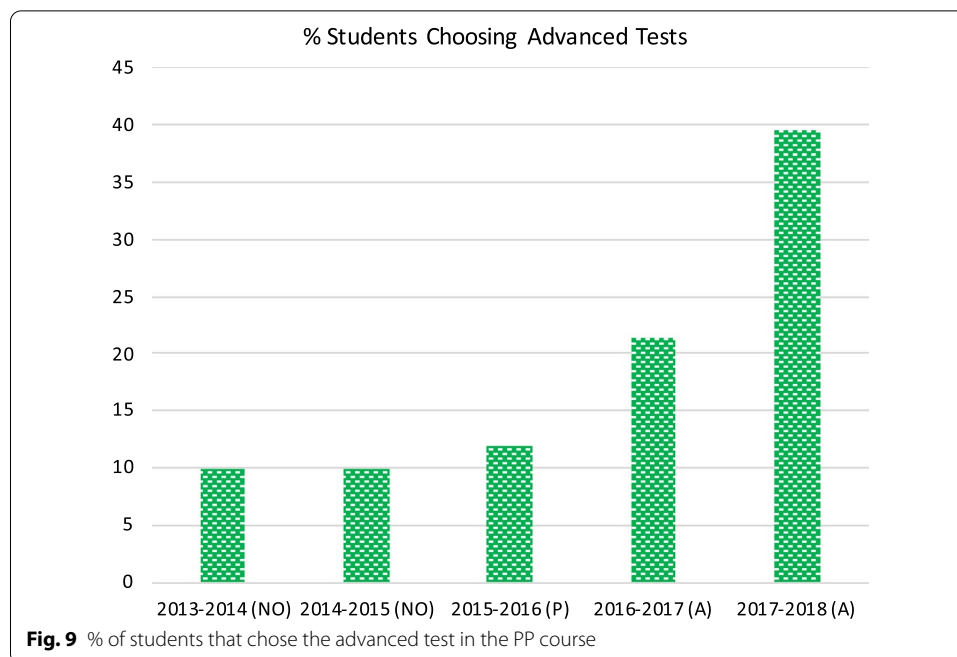
In this section we report a second set of experimental data about the effectiveness of the Diogene-CT platform with respect to learning outcomes. In order to do this, it is important to note that the tool has been used in two courses of the undergraduate computer-science curriculum at University—PP and OOP—for which teaching over the last decade has also been based on other experimental methodologies.

More specifically, we adopt a teaching methodology that is loosely inspired on SOLO—Structure of Observed Learning Outcome (Biggs and Collis 2014). The most relevant aspect of the teaching method with respect to this paper is the organization of the final exam.

Exams are based on a written, multiple-choice test, and on a practical, coding test. During the practical tests students are required to implement a fragment of application logic on a personal computer in our teaching lab. They use the software-development kit used throughout the course—e.g., gcc suite for C/C++ programs, or the JDK for Java programs—and need to deliver a piece of software that is both syntactically and logically correct.

The final exam offers a choice between two different practical tests:

- the first, called the *base* test, is based on a simpler specification, and more standard algorithms and data structure; here the focus is on verifying that the student is able to write logically-correct code with a correct methodological approach in order to implement a simple functional specification; loosely speaking, this test allows students to get a C or D grade.
- the second, called the *advanced* test, has a more complex specification, less standard algorithms and data structure, it may require to code in more than one programming language, and has a strong emphasis on test-driven development and regression testing; the goal is to verify if the student is able to handle a more complex specification with advanced coding techniques; this test allows students to get an A or B grade.



The organization of the final exams is presented to students very early during the course (typically during the first lesson), and students are completely free to choose the test to solve during the exam. Grades vary based on their choice, as discussed above. More important, it is made very clear, during the lessons, whether arguments are relevant to one level or the other.

Figure 9 shows the percentage of students that chose the advanced test and passed it with respect to the total number of students taking the test in the PP course for each academic year. Notice that grades for the advanced test reflect the increased complexity of the exam and are therefore consistently higher than those for the base test.

It can be clearly seen that introducing the metaphor has brought to a significant increase in the number of advanced tests, and therefore an advancement in grades and learning outcomes. We believe that, on the one side the adoption of the Diogene-CT methodology and tools made more clear the concepts introduced during the lessons, thus generally improving learning, and on the other side they fostered a deeper involvement of students in course activities, with a strong impact on learning outcomes.

To prove our insights, we also conducted a t-test (Welch t-test) on the group where our approach was not adopted (2013–2014 and 2014–2015 years) and the one where our approach was completely adopted (2016–2017 and 2017–2018 years). For each group, we counted the number of chosen base tests and advanced tests. The null hypothesis we want to reject is that using our approach does not change the number of chosen advanced tests. Using a 0.05 confidence interval we obtained a two-tailed P value equals 0.0226. By conventional criteria, this difference is considered to be statistically significant, and we can reject the null hypothesis.

User study

To further validate our approach, we conducted a user study among students. We involved two groups of students:

- Group 1 was composed of 3rd-year students that had attended both courses, i.e., PP in 2016–2017, and OOP in 2017/2018.
- Group 2 was composed of 1st-year students attending PP in 2018–2019.

We conducted a survey with each group. The goal of the survey was to gather feedback from the students about the ACME methodology. We made two different questionnaires: the former on the use and effectiveness of Diogene-CT and ACME in PP; the latter on OOP. Each questionnaire contained five questions, the first four of which were multiple-choice (with options Very Poor; Poor; Good; Very good) and the last one open-answer.

The PP questions were as follows:

- Q1. Do you think that the use of metaphors and animations facilitates learning the basic concepts of procedural programming (variables, assignments, inputs and outputs)?
- Q2. Do you think that the use of metaphors and animations facilitates learning control structures?
- Q3. Do you think that the use of metaphors and animations facilitates learning the concepts of modular programming (use of sub-programs and passing parameters)?
- Q4. Do you think that the Diogene-CT metaphor helps learn several programming languages, since it is not linked to a specific programming language?
- Q5. What aspect would you like to improve on the mechanical arm metaphor and its animations?

The OOP questions were as follows:

- Q1. Do you think that the use of metaphors and animations facilitates learning the basic concepts of object-oriented programming (components, classes, objects, constructors, packages)?
- Q2. Do you think that the use of metaphors and animations facilitates learning the notions of reference and message?
- Q3. Do you think that the use of metaphors and animations facilitates learning concepts related to inheritance and polymorphism?
- Q4. Do you think that the Diogene-CT metaphor helps learn several programming languages, since it is not linked to a specific programming language?
- Q5. What aspect would you like to improve on the robot-community metaphor and its animations?

Twenty-four questionnaires were collected Group 1 about PP (Fig. 10). All of the students who participated in the survey had successfully completed their attendance of the PP course and passed the final exam. As it can be seen in Fig. 10, feedback were largely positive. More specifically:

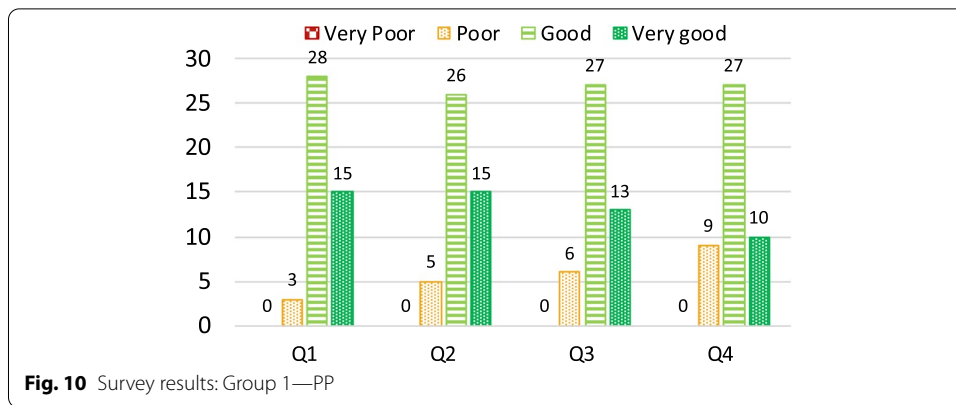


Fig. 10 Survey results: Group 1—PP

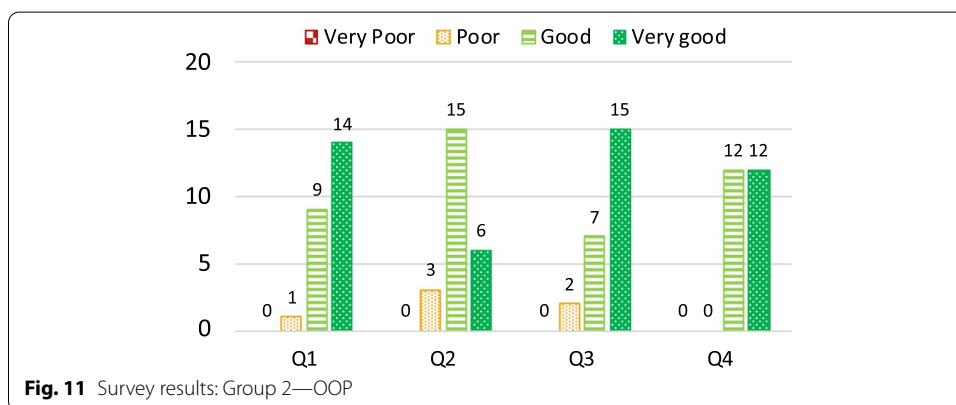
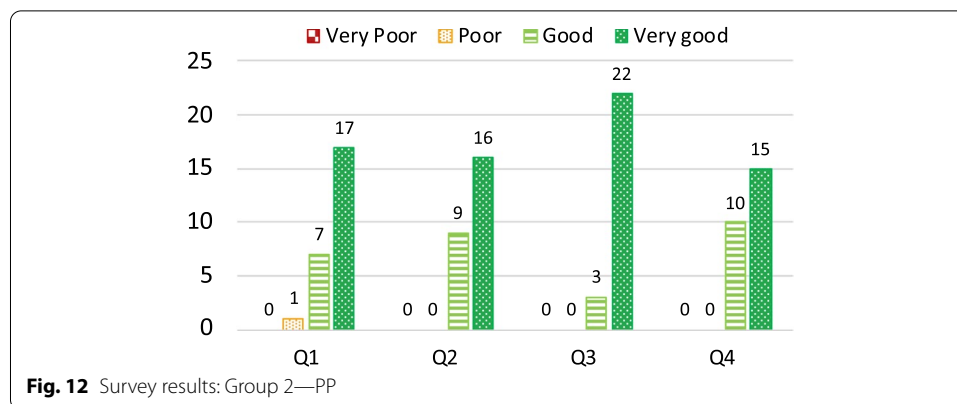


Fig. 11 Survey results: Group 2—OOP

- Answers to questions Q1 (impact on learning basic concepts) and Q3 (impact on learning modular programming and functions) were largely positive. This is an important result since modular programming and the use of functions are typically perceived by students as a difficult topic, yet a fundamental one.
- Also, all answers to question Q5 (language independence) were positive, i.e., all of the participants acknowledged that language-independence is a crucial feature of the approach.
- The question that gave results more mixed was Q2 (impact on control structures). Here, while the overall judgment was largely positive, students perceived a smaller benefit associated with the methodology. This is understandable since the only scene element directly related to control structures in the procedural-programming metaphor is the lollipop.
- Answers to Q5, the open-answer question show that students consider very important the availability of the attached mode to play with code and exercise at home.

Twenty-five questionnaires were collected from Group 2 about OOP (Fig. 11). Here all answers were largely positive. It is interesting to note that students showed great appreciation for the impact of Diogene-CT and ACME while learning inheritance and polymorphism, two difficult OOP topics. Also, in this case, open answers to Q5 show



that the students consider the availability of the attached mode to be very important. Some of the students also suggested to evolve the metaphor to further ease the learning of these subjects: (a) relationship between a class and its objects; (b) packages and constructors; and (c) the Model-View-Controller (MVC) pattern.

Finally, 46 questionnaires were collected from Group 2 about PP (Fig. 12). In this case, answers were less enthusiastic than those expressed by Group 1, although they were still largely positive (no “Very Poor” options are chosen, and only an overall 14.28% of “Poor” options). This is most likely because these students were still attending the PP course at the time of the survey, i.e., they were still struggling with learning PP concepts. Open answers to Q5 were mainly concerned with requests for improvements and extensions, i.e.: (a) improve string readability; (b) smoothen animations; and (c) speed-up transitions, especially for the most obvious animations. Some of the students commented that ACME can be considered a useful tool for newbies, but it is less effective with more experienced students. Also, in this case, answers confirmed the importance of the attached mode to practice in autonomy.

Based on these ideas, we believe that the results of the survey confirm the effectiveness of the approach when tested on the field. We will consider all comments and suggestions to refine and improve the method and the tools.

Summary of experiments

Diogene-CT can be seen as an holistic approach to the problem of supporting instructors and students in the task of teaching and learning coding. At the roots of this approach are three main components:

- a set of consistent metaphors, that serve the purpose of simplifying the approach to computer-programming concepts;
- a collection of executable tools, that make the attempt of introducing the metaphors in actual coding curricula more concrete;
- a pedagogical approach centered around the metaphor and toolset.

We believe that our experiments show that the combination of these features has brought clear advantages to our computer-science students. Results in section “[Effectiveness on course-completion rates](#)” show that course-completion rates have kept

stable, in a general context of student cohorts of decreasing quality. Even more important, data in section “[Effectiveness on learning outcomes](#)” show how participation in courses was more qualified, and this was reflected by results in final tests. Finally, our user study show that students perceive this approach as a valuable addition to teaching.

Discussion

As discussed in the previous section, our experiences with the system in introductory coding courses, both for procedural and object-oriented programming, have shown that the Diogene-CT approach can be highly effective. At the same time, they have provided precious insights about possible developments of the tool.

We typically use quite heavily the tool and the metaphor in the first part of the Procedural-Programming course. This part usually covers approximately 40 h of teaching, with 2:1 lessons to lab ratio, and goes from introductory concepts, like variables and data types, to functions, modular programming, arrays and lists. In the course of these initial lessons, the animations give students a very concrete perspective on the actual operational semantics of instructions, and allow instructors to discuss in great detail both simple concepts—like variables or parameter passing—and more complex ones, like recursion or pointers. The tool, when used in attached mode, also greatly helps student in debugging their first attempts at solving simple exercises, by providing a clear and graphical depiction of their logical errors.

However, we have noticed that, as soon as students become more experienced with coding and start dealing with medium-complexity tasks—like developing menu-based user interfaces, or writing algorithms on collections—they tend to rely less on the tool. In fact, the robot-arm metaphor may become boring when applied to very large code-bases, since its perspective is very fine-grained, and therefore medium and upper-level students may get the impression that it slows them.

To solve this problem, in the second half of the course we tend to use the tool essentially in detached mode. We record videos of long animations of medium-complexity code, and then cut the full videos to extract the most significant parts that can be useful to students. In this respect, we believe that it would be very useful to introduce breakpoints, in the style of debuggers, to enable users to fast-track to specific points within the code, and a trace-like function to start and stop the generation of animations at these points.

The tool is perceived as a very useful addition again as soon as students are exposed to object-oriented programming. The introduction to objects and classes usually covers approximately 20 h of our Object Oriented programming course, also in this case with a 2:1 lessons to lab ratio. Throughout these lessons, the possibility of exploring code executions by looking at the sequence of messages and the heap state is typically appreciated by students. Similarly, the tool be very useful when introducing concepts like object hierarchies and binding. In these cases, Diogene-CT can be used to emphasize the implicit association between object in a hierarchy through the “super” property.

Again, when projects become more complex and the heap increases in size the tool tends to become somehow less effective. Also in this case, debugger-like functionalities would very useful in order to restrict the collection of objects to focus animations on.

It is important to emphasize that not all important concepts can be properly addressed using Diogene-CT alone. For example, methodological guidelines like information hiding, the notion of interface vs implementation can only indirectly be emphasized during animations. Similarly, the comparison of strictly algorithmic aspects, like, for example, hashing vs sorting, is not among the main goals of Diogene-CT.

In this respect, we believe that Diogene-CT can be effectively used in conjunction with other tools, like, for example, algorithm visualization tools or UML-based code-visualization tools to complement and reinforce their abstractions.

Conclusions and future work

This paper introduces the Diogene-CT methodology and associated toolset to support teaching and learning computer-programming skills in school curricula of the secondary grade and higher. Based on the analysis of related works in this field, we argued that Diogene-CT represents the first of a new breed of approaches, that we called code-animation environments.

We discussed our practical experiences in the framework of the Computer Science bachelor degree at the University. Our user study among students that experienced the method in their computer-programming courses gave us largely positive feedback, thus confirming the effectiveness of the approach.

As future work, in addition to extensions discussed in the previous section, our goal is to develop Diogene-CT into a fully-fledged tool for teaching computational-thinking skills in secondary schools. This will require to extend the scope of the methodology into more interdisciplinary topics related to STEM. In this way, students would be exposed to a new topic (e.g., integrals in math, or levers in physics); a set of practical problems would be presented (e.g., compute the integral of a function or calculate one lever) and the learners will work in groups to develop a computer program to solve the problems by using Diogene-CT in attached mode.

Acknowledgements

Not applicable.

Authors' contributions

On behalf of all authors, the corresponding author states that all the authors have contributed equally. All authors read and approved the final manuscript.

Funding

Not applicable.

Availability of data and materials

Not applicable.

Competing interests

On behalf of all authors, the corresponding author states that all the authors they have no competing interests.

Received: 7 October 2020 Accepted: 10 February 2021

Published online: 05 March 2021

References

- Ackermann, E., Gauntlett, D., & Foundation, C.W.L. (2009). Defining... systematic creativity. Retrieved from <https://www.legofoundation.com/media/1078/systematic-creativity-report.pdf>.
- Bayliss, J. D. (2009). Using games in introductory courses: Tips from the trenches. *ACM SIGCSE Bulletin*, 41(1), 337–341.
- Bers, M. U., Flannery, L., Kazakoff, E. R., & Sullivan, A. (2014). Computational thinking and tinkering: Exploration of an early childhood robotics curriculum. *Computers & Education*, 72, 145–157.

- Biggs, J. B., & Collis, K. F. (2014). *Evaluating the quality of learning: The SOLO Taxonomy (Structure of the Observed Learning Outcome)*. Cambridge: Academic Press.
- da Silva, T.R., Medeiros, T., & da Silva Aranha, E.H. (2015). The use of games on the teaching of programming: A systematic review. In *CibSE* (p. 474)
- Dehnadi, S. (2006). Testing programming aptitude. In *PPIG*, (p. 9).
- Grossman, P., et al. (2009). Research on pedagogical approaches in teacher education. In *Studying teacher education* (pp. 437–488). US: Routledge.
- Harvey, B. (1997). *Computer science logo style: Symbolic computing* (Vol. 1). Cambridge: MIT Press.
- Hundhausen, C. D., Douglas, S. A., & Stasko, J. T. (2002). A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing*, 13(3), 259–290.
- Jawad, H.M., de Laski-Smith, D., & Tout, S. (2018). The code genie programming environment. In *2018 IEEE International Conference on Electro/Information Technology (EIT)* (pp. 0163–0168). <https://doi.org/10.1109/EIT.2018.8500194>.
- Kelleher, C., & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR)*, 37(2), 83–137.
- Kölling, M. (2010). The greenfoot programming environment. *ACM Transactions on Computing Education (TOCE)*, 10(4), 14.
- Kölling, M., Quig, B., Patterson, A., & Rosenberg, J. (2003). The bluej system and its pedagogy. *Computer Science Education*, 13(4), 249–268.
- Ma, L., Ferguson, J., Roper, M., & Wood, M. (2007). Investigating the viability of mental models held by novice programmers. *ACM SIGCSE Bulletin*, 39(1), 499–503.
- Marginson, S., Tytler, R., Freeman, B., & Roberts, K. (2013). Stem: Country comparisons: International comparisons of science, technology, engineering and mathematics (stem) education. Final report.
- Marín, B., Frez, J., Cruz-Lemus, J., & Genero, M. (2018). An empirical investigation on the benefits of gamification in programming courses. *ACM Transactions on Computing Education (TOCE)*, 19(1), 4.
- Moreno, A., Myller, N., Sutinen, E., & Ben-Ari, M. (2004). Visualizing programs with jeliot 3. In *Proceedings of the Working Conference on Advanced Visual Interfaces* (pp. 373–376).
- Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M., & Paterson, J. (2007). A survey of literature on the teaching of introductory programming. In *ACM Sigcse Bulletin*, vol. 39 (pp. 204–223). ACM.
- Qian, Y., & Lehman, J. (2017). Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education (TOCE)*, 18(1), 1.
- Siekman, G., & Korb, P. (2016). Defining stem skills: Review and synthesis of the literature. In NCVET. Commonwealth of Australia, AU.
- Sorva, J., Karavirta, V., & Malmi, L. (2013). A review of generic program visualization systems for introductory programming education. *ACM Transaction on Computing Education*. <https://doi.org/10.1145/2490822>.
- Tsalapatas, H., Heidmann, O., Alimisi, R., & Houstis, E.N. (2012). Game-based programming towards developing algorithmic thinking skills in primary education.
- Urquiza-Fuentes, J., & Velázquez-Iturbide, J. Á. (2009). A survey of successful evaluations of program visualization and algorithm animation systems. *ACM Transactions on Computing Education (TOCE)*, 9(2), 1–21.
- Watt, H. (2016). Session d: Promoting girls' and boys' engagement and participation in senior secondary stem fields and occupational aspirations.
- Wing, J. M. (2006). Computational thinking. *Communications of the ACM*, 49(3), 33–35.

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Submit your manuscript to a SpringerOpen[®] journal and benefit from:

- Convenient online submission
- Rigorous peer review
- Open access: articles freely available online
- High visibility within the field
- Retaining the copyright to your article

Submit your next manuscript at ► [springeropen.com](https://www.springeropen.com)
