Informatics in Education, 2024, Vol. 00, No. 00, –

DOI: 10.15388/infedu.2024.18
ACCEPTED VERSION (WITHOUT THE JOURNAL LAYOUT).

Creative programming in architecture: a computational thinking approach

Patricia DOMÍNGUEZ-GÓMEZ[1], Flavio CELIS[2]
[1] *Architecture Department, Universidad de Alcalá, Alcalá de Henares, Spain*
[2] *Architecture Department, Universidad de Alcalá, Alcalá de Henares, Spain*
e-mail: patricia.dominguez@uah.es, flavio.celis@uah.es

Received: August 2023

Abstract. The creative programming language Processing can be used as a generative architectural design tool, which allows the designer to write design instructions (algorithms) and compute them, obtaining graphical outputs of great interest. This contribution addresses the inclusion of this language in the architecture curriculum, within the context of digital culture and alternative approaches to how digital tools are used and learned. It studies the different processes related to Computational Thinking that are triggered in the prototyping of computer applications and that lead to creativity. The similarity between architectural design and programming is analysed, both in problem solving (abstraction, decomposition, iterative revisions -debugging-, etc.) and in the use of mechanisms of a digital nature (loops, randomness, etc.). The results of the design and testing of a pilot course are shown, in which the way of teaching, learning and using this programming language is based on the graphical representation of problems through sketches.

Keywords: Computational thinking, creative programming, architecture

1. Introduction

One issue in our current relationship with technology is the obsession with technical objects. When a technological tool is analysed, efforts are focused on “how it works” and “its capabilities”, and not so much on other aspects, such as what its cultural background is or how it affects human thought processes.
In architecture, multiple digital tools have been applied in the profession, accelerating project processes (such as the Building Information Modelling methodology) or responding to the market demand of visualising architectural objects before building them (3D visualisation tools).
This architectural practice that emerged in response to the market has been transferred to education, where architecture colleges offer specialised courses in the use of information technologies. The classes are operative in nature, and finding a more global perspective on machines is rare. This approach reduces students and future professionals to mere users who lack the skill to suggest an alternative direction in the use and development of applied technologies.
In contrast, using digital tools based on the acquisition of skills for problem-solving is present in prior educational stages (primary education, secondary education). Numerous authors refer to these skills as “computational thinking” (CT) (Cansu & Cansu, 2019) (Romero, Lepage, & Lille, 2017) (Wing, 2008) (Rafiq et al., 2023).

Creative programming languages in architecture classrooms allow students to learn these skills and enrich their training in digital technologies. Herein is the window of opportunity. These languages have a graphic output, creating a space for morphological and creative experimentation (Burry, Datta, & Anson, 2000). Thus, they are a vehicle for understanding the mechanisms and concepts of digital environments (loops, iterations, algorithms, etc.).
This research paper centres on the design and execution of experimental training in a particular creative programming language for architecture students. The course aimed to introduce students to CT and programming operations to equip them to design small software applications and include them in their creative processes.

2. Background

2.1. *The relationship between man and computers: technocentrism and digital culturalization*

The history of IT has been determined by the image human beings have of their intelligence (Breton, 1989), and more so in the first stages spanning the 40s, 50s, and 60s of the 20th century. They witnessed the emergence of diverse theories on human behaviour (the intent was for machines to emulate it). This was the birth of cybernetics, and the notion of information became one of the keys to the development of Western society.
These phases witnessed the big questions concerning the relationship between man and machines. In subsequent decades the rapid advance of technology and massive expansion of IT products (both software and hardware) progressively displaced the debate to more specialised discourses and a greater interest in operational matters (Domínguez, Celis, & Echeverría, 2022).
Technocentrism, coined by philosopher, mathematician, and Artificial Intelligence pioneer Seymour Papert, refers to the fixation with the technical object, ignoring the cultural factors that frame and condition it (Papert, 1987). According to Paul Virilio, the obsession responds to a lack of lateral vision and context and is the direct consequence of the velocity characteristic of the current era: “The faster you go, the farther you project yourself to anticipate what may occur, and, simultaneously, laterality is reduced” (Virilio, 2016).
To palliate this obstinacy about technical objects, both authors point to the need to develop a criticism of technology that, in the field of computation, would be as important as “literary criticism and social criticism” are in their respective fields (Papert, 1987). That contemporary dissatisfaction with the relationship human beings have with technology, added to a growing interest in highlighting the value of other aspects beyond the merely operative ones, would give rise to the emergence of a technical culture (Virilio, 1999) in which the analysis and valuation of digital tools would focus on what they add to the human process they are a part of, and not so much “what they can do” on their own. To paraphrase Papert (1987), reducing the study of technical objects to their “capabilities” would be like asking, “Does wood produce good houses?” or “Do hammers and saws produce good furniture?”.

2.2. *The architect’s relationship with computers: using digital tools in the project*

phases of architecture

Architects began integrating Computer-Aided Design (CAD) in the 60s. The purpose of the first CAD programs was to generate tools that would produce not only graphic results (planimetries, volumetric studies, perspectives) but also lists of materials, surface counts, etc. (Cardoso, 2013). It was not until the arrival of BIM (Building Information Modelling) technology decades later that these tools that virtually emulated constructive processes materialised (Garber, 2014).

Thus, until the popularisation of BIM, most architecture firms employed CAD programs as tools for representation in their projects. Furthermore, they adopted specific programs for 3D animation, producing a boom of photorealistic infographics that became essential in attracting clients but have been strongly criticised by theorists in the discipline. To cite architect Stan Allen, "the undeniable acceptance of computers as visualisation tools is clearly imposed by the market (…) Its trajectory [referring to visualisation] is not from image to reality, but from image to image. (…) [Visualisation] does not work to transform reality, only to reproduce it" (Allen, 1995 (2009)).

CAD programs gave rise to other technologies and more specialised tools. These included parametric design tools in which architects select parameters that define algorithms that produce and dimension architectural shapes. This set of tools is located within generative design and explored in other fields, such as art practice and graphic design. It can be defined as one that uses a system based on algorithms capable of working with some degree of autonomy and influencing the result (Galanter, 2016).

What sets generative design tools apart is two-fold. Firstly, they are used in different project phases (Fig. 1). Just as there are tools intended exclusively for architectural representation (such as those employed in generating photorealistic infographics) or for technical development of executed projects, generative design programs are used in previous ideation and configuration phases of architectural shapes. Secondly, these tools extrapolate the fundamentals of computing to formal generation, putting new ways of experimentation based on programming mechanisms at the service of architects: "The computer is an abstract machine, and as it moves beyond the logics of visualization, new potentials open up" (Allen, 2005 (2009)).

Family of digital tool	**Project phase**			
	Ideation	**Configuration**	**Representation**	**Technical development**
Computer Aided Design (CAD)		X	X	
Building Information Modelling (BIM)		X	X	X
3D animation software			X	
Generative design	X	X		

Fig. 1. Most common families of digital tools used in professional practice and project phases in which they are employed. Prepared by the authors.

2.3. *The relationship between students and computers: Curriculum proposals and computational thinking in the European context*

Architecture colleges have mirrored what occurred in the professional sphere regarding adopting digital tools. Thus, the demand for highly detailed infographics resulted in

“ghastly renderings that polluted every minor school of architecture” (Frazer, 2005). However, with the popularisation of BIM technologies and the experimental interest in generative design tools, contemporary curriculums include alternative paths other than photorealistic rendering.

The curricula of the top European architecture colleges have been analysed to determine the state of the art of the subjects related to digital tools in Architecture classrooms. The following rankings have been considered in the selection of schools: CImagoIR (SCImago, 2023), Academic Ranking of World Universities (Shangai Ranking, 2023) and World University Rankings (Times Higher Education, 2023) (Fig. 2). Four universities from different countries belonging to the European Higher Education Area have been selected at the top of these rankings.

In the analysis, electives or optional subjects are included (OP), as are obligatory ones (OB), chairs (C) and masters’ courses (MC) directly related to the application of digital tools in the education and training of architects. Three scales have been identified in terms of teaching digital tools: teaching generalist tools (G), such as more traditional CAD systems or new BIM technologies; teaching parametric design tools (PD), like those using programming plugins and parametrisation in CAD or BIM platforms; and, finally, those that solely teach programming languages applied to creative processes (CP).

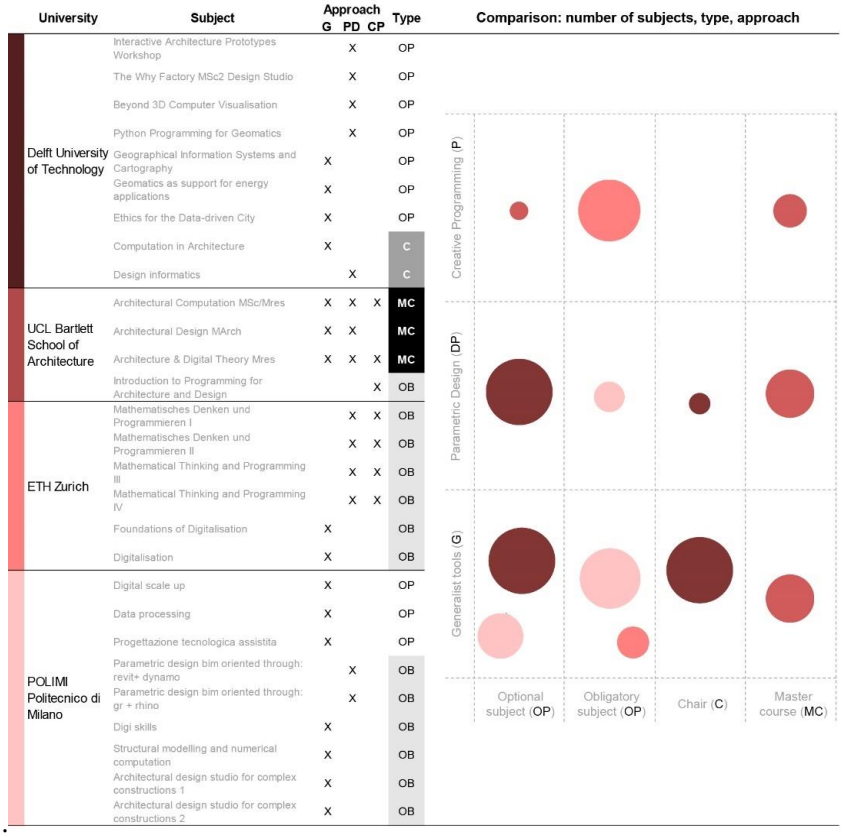


Fig. 2. Analysis and comparison of digital tools in the curricula of the four selected

universities. Prepared by the authors.

James Steele, the University of Southern California lecturer, said twenty years ago that "very few schools have curriculums where computers are integrated in all project subjects; most of them limit themselves to offering classes of technical computing. The idea of a theory of computers is still something foreign (…)." (Steele, 2001). The analysis of the curricula of top European schools revealed that generalist digital tools are still the most popular course offerings, and other tools that are more experimental in nature are timidly making their way.

Following that idea posed by Steele of a "theory of computers", the focus of lecturer Hovestadt (ETH Zurich) and his series of courses "Mathematical Thinking and Programming" (I, II, III and IV) stand out. The tool used in the classroom is Mathematica, a formulation and mathematical analysis software that enables the study of shapes from the perspective of their mathematical parametrisation. It is striking to note the underlying discourse of the theoretical classes. In his introductory class, Hovestadt compares the role of coding in current architecture to the role of drawing in the architecture of the Rennaisance (Hovestadt, 2020). In the teaching guides of his courses, he explains that "It is not about the HOW, but rather about the WHAT, not about virtuosity when dealing with digital tools, but rather about understanding coding" (Hovestadt, 2021).

In other words, using a tool does not mean users have grasped its underlying concepts. This observation is the pedagogical foundation of computational thinking (Wing, 2008) (Osio & Bailón, 2020). It is a problem-solving model that has sparked growing interest in the academic community, specially during the last decade (Rafiq et al., 2023), and is still the object of research, for which multiple definitions exist (Cansu & Cansu, 2019). According to Jeannette Wing (2011), CT "is the thought process 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." Information programmers have traditionally carried out these processes in developing applications: abstraction, algorithmic thinking, automation, decomposition, debugging and generalization (Cansu & Cansu, 2019) (Rafiq et al., 2023) (Wing, 2008). The proposal of advocates of CT as an educational competence involves transferring these thought processes to the classroom so that students acquire problem-solving skills. The vehicle for learning to internalise CT is usually a programming language adapted to an audience without specialised knowledge in computation. Furthermore, most research in the possibilities of CT is being conducted at primary and secondary educational levels.

2.4. *Programming languages as pedagogical and creative tools*

In the field of architecture, the use of programming languages to design is a practice included in computer-generative design. This way of designing is defined as a process in which, through a series of algorithmic instructions, part of the result of the project is given to the computer (Galanter, 2016). In the last two decades, generative design in architecture has been a growing field of experimentation (Caetano, Santos & Leitão, 2020), both in the professional and academic fields, and is based on the idea that the "form designer" becomes a "form programmer" through an iterative design process (Fig. 3). Professor Asterios Agkathidis proposes: "what if we did not design forms anymore

and developed algorithms instead?" (2012), and also implements different generative design processes in the architecture classroom. Although he does not specify to what extent he implements digital tools using programming components, the proposed processes, such as algorithmic patterns, are related to the mechanisms of the digital environment (Agkathidis, 2015).

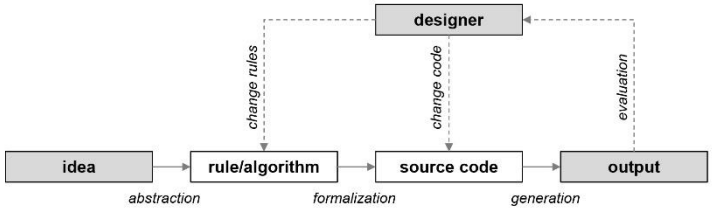


Fig. 3. Diagram of the generative design process (Bohnacker et al., 2012) cited by Agkathidis (2015). Reworked by the authors.

One of the most popular tools for generative architectural design is Grasshopper. It is a plug-in to Rhinoceros CAD software that uses a visual programming language, based on components, nodes and wires (Fig. 4). The architect can develop algorithms that generate three-dimensional shapes, and also parameterise these shapes to get different versions of the design in the CAD environment (Tedeschi, 2014).

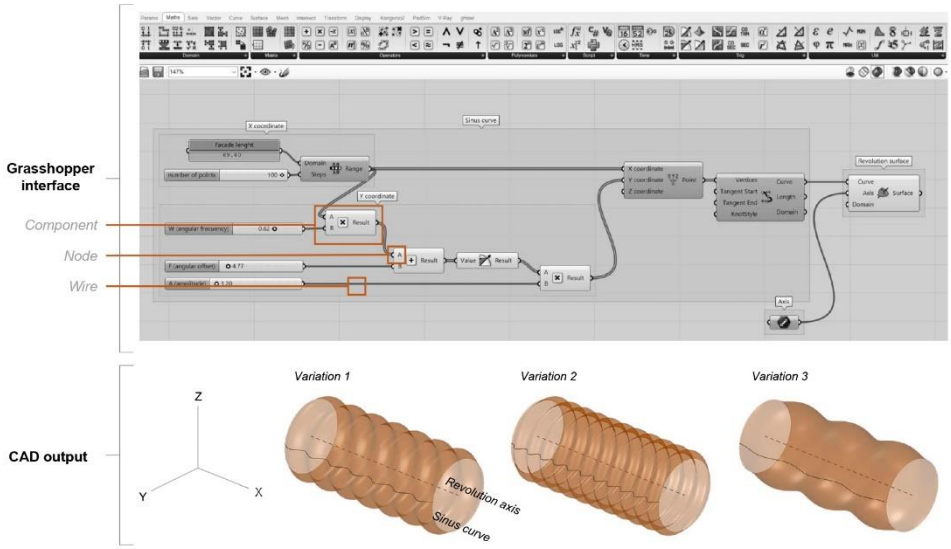


Fig. 4. Grasshopper environment and its CAD output (Rhinoceros program). Example of a three-dimensional shape generated with this visual programming language. The design variations are made by changing the values of the main variables (parameters). Prepared by the authors.

Its three-dimensional representation power has made it very popular in the field of architectural design. As a result, it is used in many of the geometry, design and computer-related subjects studied in the previous section, generally with a focus on memorising its components. The Grasshopper language is complex, and the programme

requires powerful computer hardware and the purchase of licences.

It should be noted that part of the Grasshopper language components are also basic programming components (variables, mathematical operations, Boolean operators, data arrays, etc.), which occasionally allows the user to operate with the program features typical of any generalist computer language. However, most of the components are oriented to the generation and manipulation of complex shapes (Tedeschi, 2014), so the user mostly works with the abstractions of the program interface and not so much with the mechanisms of programming.

Another tool that is included within the digital tools of generative computer design is the Processing language, the vehicle of this research.

Processing is an open-code programming language based on Java (one of the main programming languages). It is aimed at generating images, videos, and sounds and has a development environment (IDE, Integrated Development Environment). Its syntaxis is conceived to make it accessible to an audience not specialising in computer science. It also has an online community that provides technical support and acts as a learning platform, helping users learn independently. It was designed at the Massachusetts Institute of Technology (MIT) in 2001 and has been quickly adopted by digital artists and designers (Reas & Fry, 2007).

Because it enables programming shapes, exporting them to CAD formats, and even communicating with other more specific programs for formal generation, it has also been experimented with as a generative design tool in architecture. Processing in the phases of conception is desirable for architects because it allows them to program their application, generating images and shapes with the parameters and algorithms of their choice.

Teachers who have tested it in their classrooms highlight its steep learning curve and the speed at which students can program their first designs shortly after beginning the course (Fricker, Wartmann, & Hovestadt, 2008). Because Processing is a simplified Java language with visual results that facilitate learning its functions -e.g. function "line(x1, y1, x2, y2)" draws a line by inputting the coordinates of its endpoints-, it is possible to obtain results quickly even in short courses. However, this instrumental vision of the Processing language incurs upon the vices detected in teaching other digital tools, and this kind of approach would be no different from any other technical computing course.

Processing entails an advantage in teaching the fundamentals of digital environments: its interface is composed of a text editor where code is entered, a console for text output, and a graphic window that reflects the output of the written code (Fig. 5). That is, there is simultaneously a verbal representation (code) and a graphic representation, which enables the visualisation of the written abstraction (Cannaerts, 2016).

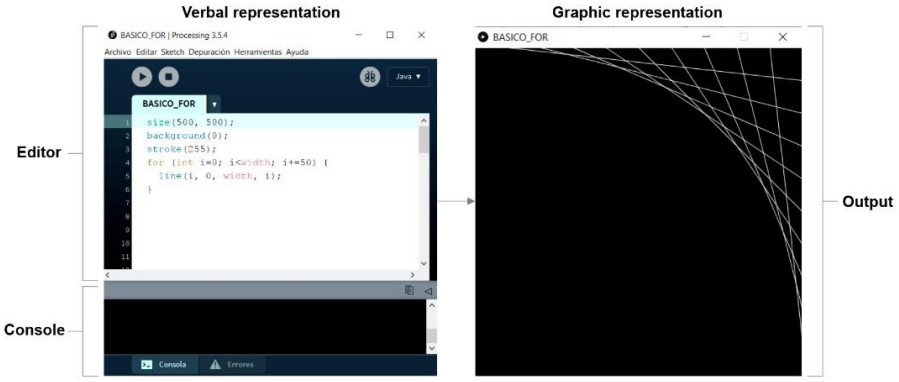


Fig. 5. Processing programming environment: text editor, console, and graphic window. Prepared by the author.

Furthermore, as the programming language is Java-based, Processing makes the basic concepts used in programming accessible. Schwill (1994) summarised these as follows: concatenation (such as data matrixes), conditionals (such as "if" or "if else" structures), iteration (control structures such as "while" or "for"), recursion (functions calling themselves), nondeterminism (programs with variable result for the same input) and parametrization.

3. Objectives

Based on the possibilities offered by the Processing programming language, the following objectives were set:

-Design an experimental training to introduce architecture students to problem-solving processes in CT and the mechanisms of digital environments using Processing as language-vehicle.

-Conduct the experimental training in the degree of Architecture, assessing the implementation, the results obtained, and the final opinions of the students.

-Evaluate the impact of teaching creative programming languages in the architecture discipline.

4. Methodology

The research centres on the production, test, and analysis of an experimental training of the Processing language in an architecture classroom, the goal of which is to generate graphic and/or audiovisual designs and use them as a vehicle to deepen understanding of digital environment concepts (common to most programming languages).

The training is designed as an intensive course lasting 18 hours and distributed into five sessions. A short duration is proposed, on the one hand, to test the steep learning curve of the language (Fricker, Wartmann, & Hovestadt, 2008), and on the other hand, to make it more comfortable and accessible to architecture students, usually immersed in a dynamic of hard and constant work. Each student is followed closely; therefore, the maximum number of students is set to 10. Students are pursuing a degree in Architecture. They are at least second-year students; thus, they are familiar with the

essential software used in the profession.
The fundamentals of the course are:

-Establish theoretical content that uses the functions of Processing language to introduce the fundamental aspects of programming. Processing language functions are selected, and priority is given to those related to algorithmic thinking.

-Devote as much time as possible to the execution of practical exercises that help students understand the problem-solving method of computer applications.

Finally, students complete an anonymous questionnaire to test their comprehension of contents and perception of the complexity of the topics covered by the course.

4.1. Establishment of theoretical content

The Processing programming language has a total of 389 functions classified according to the official platform into fifteen groups: environment, data input, typography, image, shapes, mathematics, data output, colour, transformations, structure, control, lights and cameras, data and rendering (Processing Foundation, s.f.). It is held that conducting an overview of the vast array of functions is unmanageable and counterproductive, as training devoted exclusively to the functions of the language would leave the more exciting and generalist underlying concepts in the background. Thus, the selection of elements that comprise the curriculum was carried out following this priority (Fig. 6):

-Functions relative to algorithmic thinking, that is, based on algorithms. These are "methods to solve a problem using a series of precise steps, defined and finite" (Joyanes, 2008). These functions are common to most programming languages. They are abstract structures, such as iterations (while and for structures), alternative (if and if else structures) and logical operators (*AND, *OR, *NOT).

-Direct graphic application functions (that is, those with an immediate graphic output), as these constitute the visual strength of the Processing language.

-Complementary functions that enable workflows with other CAD programs or support other files (image, PDF).

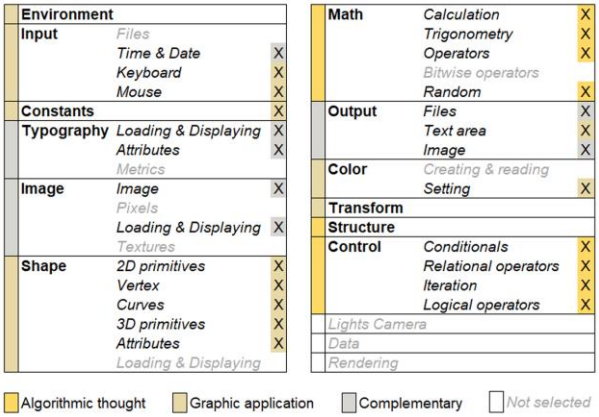


Fig. 6. Selection of course curricula from the groups of functions provided by the Processing language. Prepared by the authors.

Thus, the theoretical content cover only one-third of the functions Processing offers. This amount is more than sufficient to meet the objectives of the experimental training course.

4.2. Design of practical activities

The objective of the practical activities is for students to conduct attempts, on paper or using the written Processing code in the IDE, of computer applications with predefined visual results to a greater or lesser degree. In this development process of the application, students will encounter the fundamentals of programming, which, citing Greenberg, Xu, and Kumar (2013), are the following:

-Break a problem down into discreet steps.
-Set initial program states.
-Use constant and variable values to quantify a problem (make it computable).
-Use loops and conditional logic to control the flow of a program.

Three scales of exercises were designed depending on the time employed in each practical activity and the autonomy of each student (Fig. 7): short, guided exercises, classroom-workshop activities, and the personal final project.

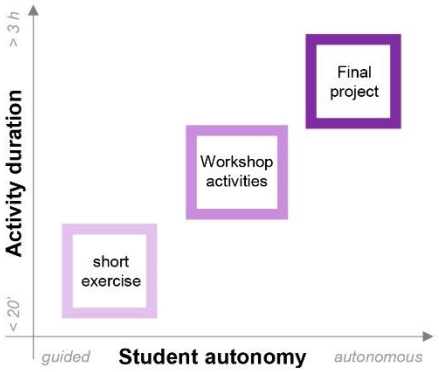


Fig. 7. Scale of practical activities of the experimental training course in Processing. Prepared by the author.

Short exercises were interspersed with theoretical explanations and constituted half of the session. They provided the students with the opportunity to put their learning into practice. The graphic results are predefined, so students, guided by the lecturer, had to analyse the problem, and suggest a code to draw the image they were presented.

The rest of the session was spent working in the classroom-workshop mode. Students solved a computer application autonomously, and the lecturer gave them feedback. In this exercise, students enjoy greater freedom and can suggest variations to a predefined topic. This consists of a transcription to the Processing environment of an existing artistic work that employs precise, clean geometries and compositive laws susceptible to computing. The works of Max Bill, Josef Albers, and Elena Asins, among others, were used as models. In the case of the latter, her work is especially interesting for the course's purpose because it uses computation processes (and, on numerous occasions, computers) in the sequencing and representation of linear geometries (MNCARS,

2020).
The personal final project is a computer application designed by students with a graphic output (static or animated) of their choice. To avoid complex applications, the focus was on using a single programming mechanism as the leading actor of the application: loops, alternative structures, mathematical operators, Boolean logic, or random functions, among others.

5. Results

5.1. Development of the pilot course

The experimental training course in Processing was conducted for nine students at the College of Architecture, enrolled as second, third-, or fourth-year students of the program. The content was distributed into five sessions (Fig. 8), and a large portion of the final session was devoted to exhibiting the results students attained.

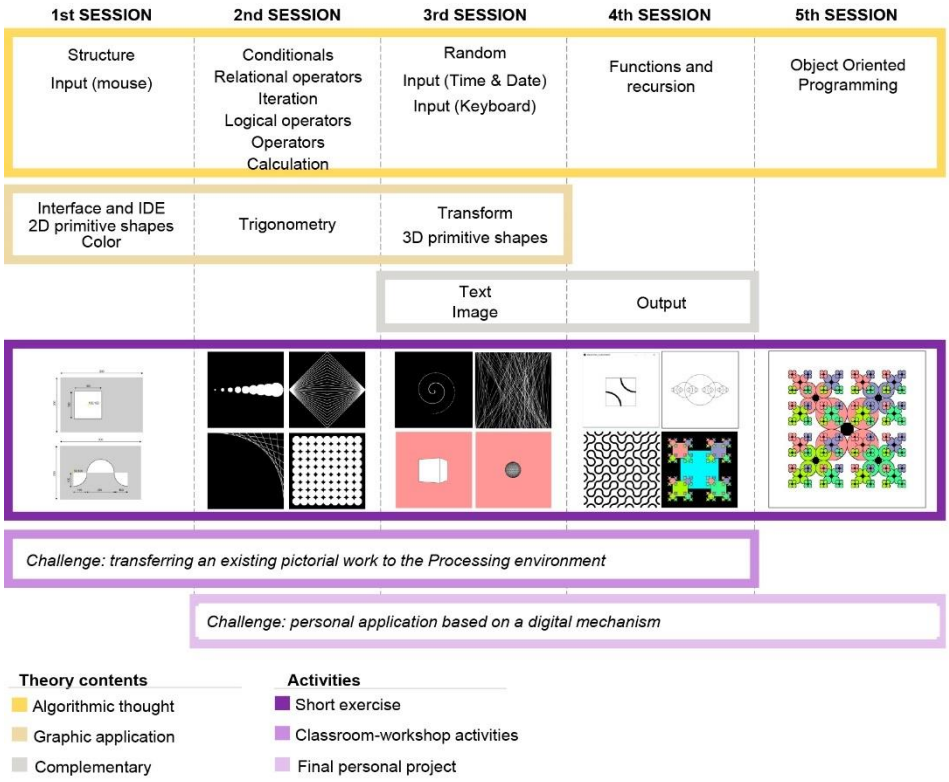


Fig. 8. Sequencing of the experimental training course in Processing. Prepared by the author.

The fundamental theoretical concepts and functions of the Processing language were explained in the first part of each session. Explanations were alternated with short, guided exercises. The exercises consisted of images (fixed or animated) produced with

Processing, but the code was not revealed to the students. Their task was to write a program to produce the image shown. Thus, the problems were graphical-mathematical in nature, and a process of backward reasoning was carried out. First, they reasoned it out on the blackboard (or on paper) to determine the parts of the problem, the variables and/or constants that intervened and the possible control elements that might be necessary (alternative, loops, etc.). The next step was to write the code in the Processing IDE and execute it, obtaining an initial graphic result. If the result hit the mark, the code was improved, and simplified using more advanced functions to eliminate lines of coding. If the outcome missed the mark or was incomplete, they had to return to the paper to try again. Next, students modified the code, yielding a new graphic result. This iterative process was repeated as many times as necessary to find the solution (Fig. 9) in a debugging process.

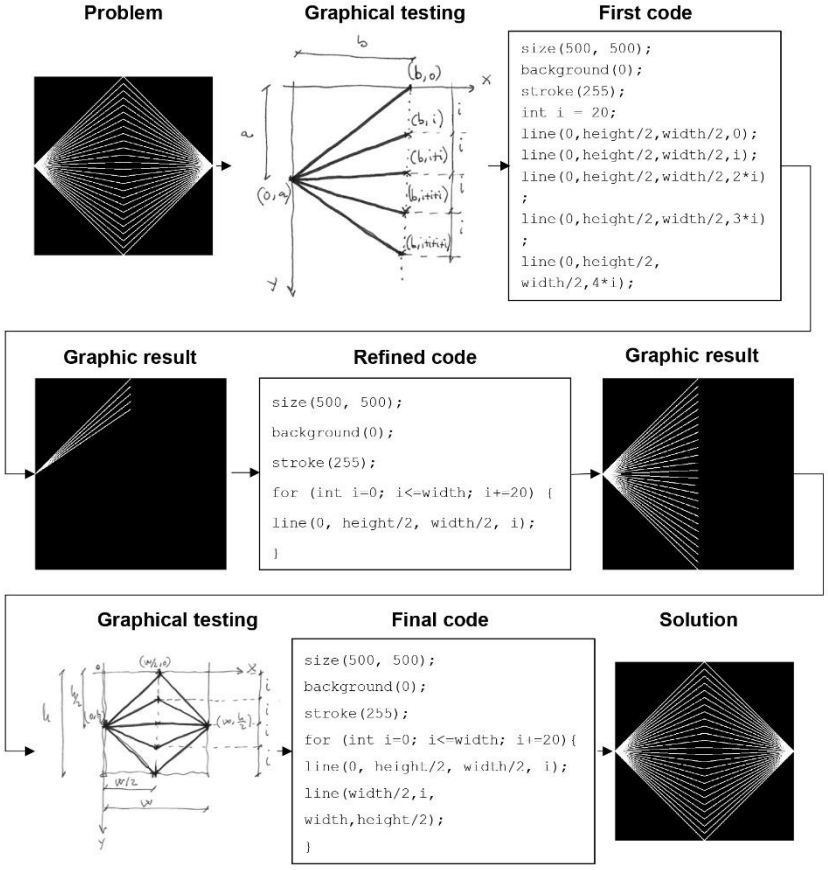


Fig. 9. Sequence of the problem-solving process of a short, guided exercise. The problem: to write the code that would yield the image shown. Prepared by the authors.

The second half of the sessions was devoted to the classroom-workshop mode in which students used paintings from the 20th century as vehicles for play and experimentation with Processing. After transcribing the works to the digital environment, students

generated a variety of results, including static images based on the original work, interactive applications which varied parts of the original work (colour, proportions, etc.) using the keyboard or mouse, or animations (Fig. 10). Applications were designed much like the short exercises, reasoning it out on paper and slowly building the different parts of the code (Fig. 11).

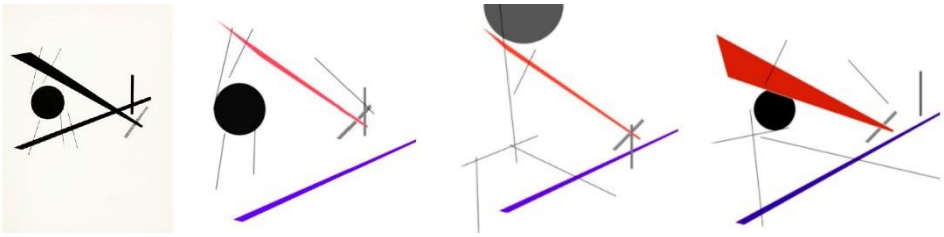
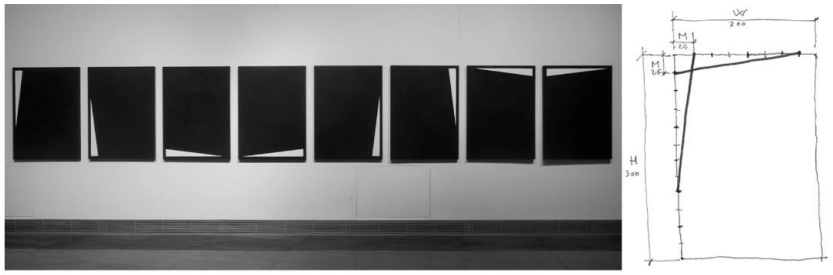


Fig. 10. Right: Vassily Kandinsky, cover of "Point and line to plane", 1926. Left: animated images generated with Processing based on Kandinsky's work. Student: Diego Bellón.



```
color bcolor = color(255, 255, 255);
void setup() {
  size (1300, 900);
  background (0);
}

void draw() {
  fill(255);
  fill (bcolor);
  bcolor= color(0, mouseX, mouseY);
  rect (100, 100, 200, 300);
  rect (100, 500, 200, 300);
  rect (400, 100, 200, 300);
  rect (400, 500, 200, 300);
  rect (700, 100, 200, 300);
  rect (700, 500, 200, 300);
  rect (1000, 100, 200, 300);
  rect (1000, 500, 200, 300);
  fill(0);
  triangle( 275, 100, 300, 100, 300, 300);
  triangle( 100, 800, 125, 800, 100, 600);
  triangle( 575, 400, 600, 400, 600, 200);
  triangle( 400, 500, 425, 500, 400, 700);
  triangle( 725, 400, 900, 400, 900, 375);
  triangle( 700, 500, 700, 525, 875, 500);
  triangle( 1000, 400, 1000, 375, 1175, 400);
  triangle( 1025, 500, 1200, 500, 1200, 525);
}
```

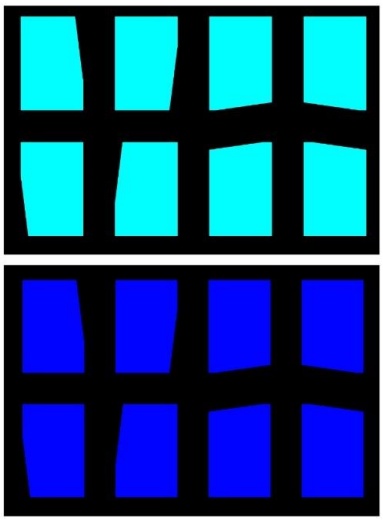


Fig. 11. Top left: Menhires (1995-1996), Elena Asins. Source: Museo Universidad de Navarra. Top right: graphic hypothesis of modulation of geometric elements in the original work. Prepared by the author. Bottom: Code and result of the interactive composition based on Elena Asins' work. Student: Carmen Puente.

Finally, the students were free to choose the software applications they wished to generate as their final personal project. Also, they could choose the programming feature that would form the centrepiece of their proposals. One popular function selected by students was the "random" function in Processing (Fig. 12). It enabled transferring part of the control to the computer. As the result was not predefined, the final compositions entailed a degree of surprise.

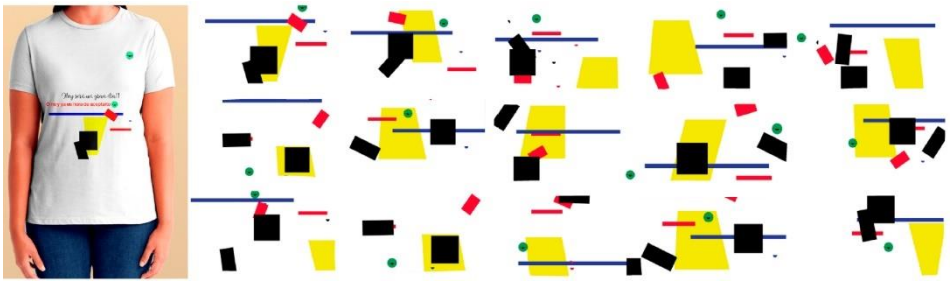
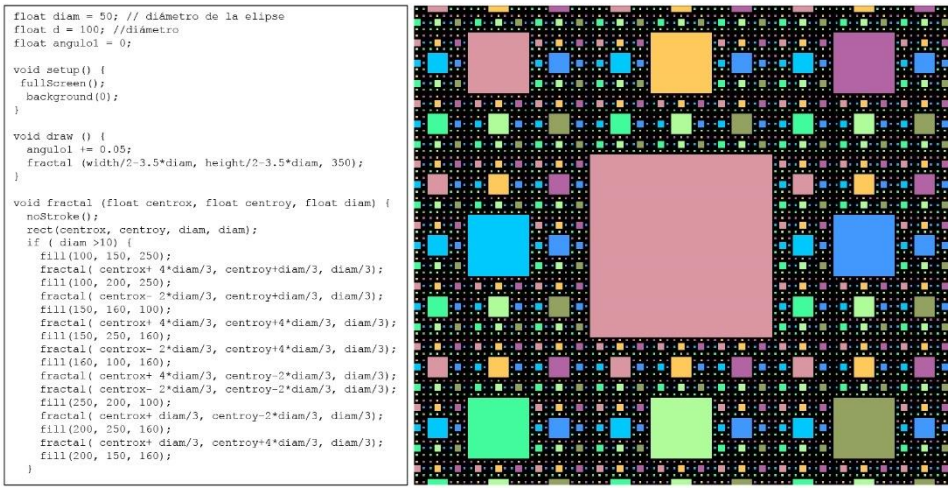


Fig. 12. "Random fashion" project: an application generating geometrical compositions based on a painting by Kazimir Malévich, using random dimension and position values. Student: Ariadna Carnicer.

Students also took an interest in modular applications that enable drawing fractal objects (self-similar geometric compositions that repeat at different scales); the reason for their appeal lies in the existence of these objects in nature and their use in current architectural projects (Fig. 13).



```
float diam = 50; // diámetro de la elipse
float d = 100; //diámetro
float angulo1 = 0;

void setup() {
 fullScreen();
  background(0);
}

void draw () {
  angulo1 += 0.05;
  fractal (width/2-3.5*diam, height/2-3.5*diam, 350);
}

void fractal (float centrox, float centroy, float diam) {
  noStroke();
  rect(centrox, centroy, diam, diam);
  if ( diam >10) {
    fill(100, 150, 250);
    fractal( centrox+ 4*diam/3, centroy+diam/3, diam/3);
    fill(100, 200, 250);
    fractal( centrox- 2*diam/3, centroy+diam/3, diam/3);
    fill(150, 160, 100);
    fractal( centrox+ 4*diam/3, centroy+4*diam/3, diam/3);
    fill(150, 250, 160);
    fractal( centrox- 2*diam/3, centroy+4*diam/3, diam/3);
    fill(160, 100, 160);
    fractal( centrox+ 4*diam/3, centroy-2*diam/3, diam/3);
    fractal( centrox- 2*diam/3, centroy-2*diam/3, diam/3);
    fill(250, 200, 100);
    fractal( centrox+ diam/3, centroy-2*diam/3, diam/3);
    fill(200, 250, 160);
    fractal( centrox+ diam/3, centroy+4*diam/3, diam/3);
    fill(200, 150, 160);
  }
```

Fig. 13. Fractal Composition. Student: Marta Martín.

5.2. Results analysis

5.2.1. Evaluation of syntactic control of the Processing language

In order to assess the understanding of the basics of the Processing language, a short individual exam was given at the end of the course. The questions, of basic level, were related to syntax (Q1, 3 points), to the formulation of conditionals (Q2, 1 point) and to the formulation of loops (Q3, 1 point). As can be seen in Figure 14, in general the students demonstrated a basic knowledge in the questions posed, finding greater difficulty in the loops.

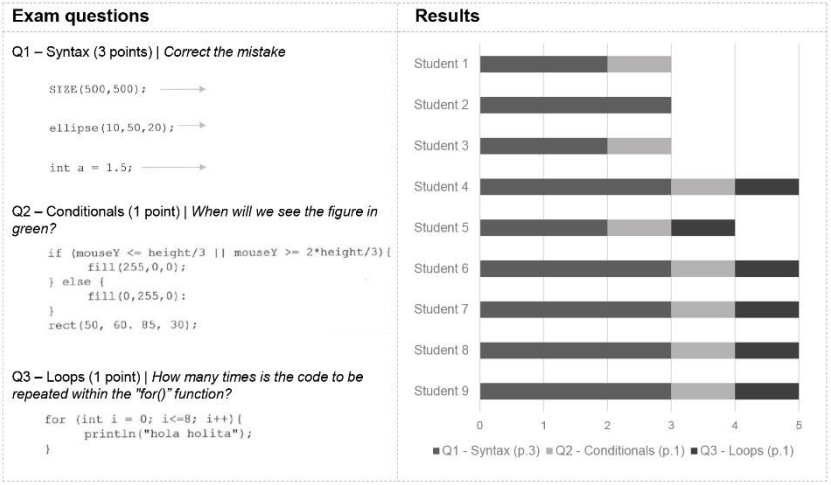


Fig. 14. Test questions and results obtained. Prepared by the authors.

5.2.2. Analysis of the students' final project

In the course's final phase, the students developed an individual project. Some of them limited their work to improving the applications resulting from transcribing existing artistic works to the digital environment. Other students, with greater motivation, developed several applications with different purposes. In total, fourteen projects are analyzed (Fig. 15).

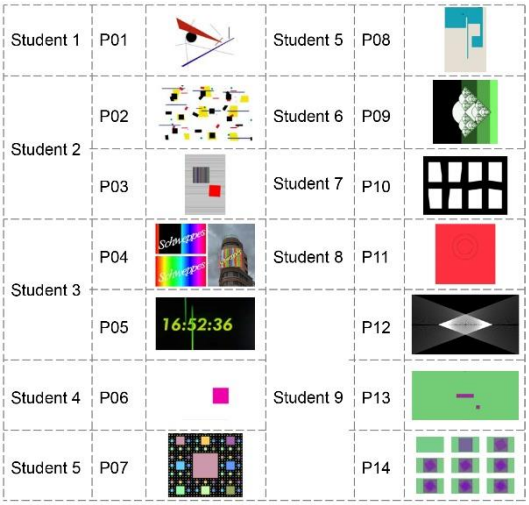


Fig. 15. List of final applications made by each student. Prepared by the authors.

On the one hand, the applications are evaluated in relation to the Processing functions most studied during the course, in order to check their comprehension (Fig. 16). For this purpose, the codes of each project are analyzed by examining the type of programming functions used (previously described in Figure 4). It is observed that very few students have used customized functions, which allow greater flexibility and modularity of the programs, and, in general, reduce the amount of code of them. This may indicate a lack of understanding of the concept of modularity, or a poor dedication to the debugging phase. Other simpler elements seem to have a greater depth, such as conditionals - an understanding of which was also demonstrated in the short test - and random numbers. Finally, the number of functions explored by each student can be related both to the comprehension of concepts and to the complexity of the resulting applications.

		Graphic application			Algorithmic thought				Number of functions
		Input (time, mouse)	Shape	Color	Random	Conditional (if - else)	Iteration (for)	Functions	
Student 1	P01		x		x				2
Student 2	P02		x		x				5
	P03	x	x	x	x		x		
Student 3	P04		x	x					4
	P05	x	x	x		x			
Student 4	P06	x	x	x		x			4
Student 5	P07		x	x		x		x	5
	P08		x		x	x			
Student 6	P09		x	x		x		x	4
Student 7	P10	x	x	x		x			4
Student 8	P11	x	x			x	x		4
Student 9	P12		x				x		6
	P13	x	x	x	x	x			
	P14	x	x			x			

Fig. 16. Analysis of the implementation of the functions most studied in the course in the students' projects. Prepared by the authors.

On the other hand, applications are assessed from the CT point of view. The analysis focuses on the main processes associated with it (Cansu & Cansu, 2019) (Rafiq et al., 2023) (Wing, 2008): abstraction, algorithmic thinking, automation, decomposition, debugging and generalization. Having monitored the work of each student, and analyzing the code and the results presented, the level of implementation of these processes in the development of their final projects is assessed.

The objective of this evaluation is not to assess the acquisition of competencies in CT; as Romero, Lepage and Lille (2017) point out, evaluating exclusively with this system "may result in assessing abilities instead of competency". The objective, more specific, is to obtain an overview of which processes associated with CT have been easier for architecture students to assimilate and could be implemented in the elaboration of written programs with Processing.

The definition of these processes, as well as the criteria used to evaluate their implementation in the development of the final work, is detailed in the rubric of Figure 17.

CT processes	Definitions	Level of implementation: Low	Medium	High
Abstraction	To reduce detail to simplify the problem, deciding "what details we need to highlight and what details we can ignore" (Wing, 2008)	Some parts of the problem are not translated into simplified data or structures, resulting in errors and/or omissions in the results.	The problem is translated into data or relatively simplified structures, producing results approximating to the desired ones.	The problem is reduced to data or simplified structures, producing the desired results.
Algorithmic thinking	To construct a scheme of ordered steps to solve problems(Rafiq et al., 2023) (Cansu & Cansu, 2019)	Unclear steps in the elaboration of the algorithm and/or omissions in these steps leading to an unexpected result.	Definition of the algorithm with some errors in the order.	Correct definition of the steps of the algorithm, including comments that order it.
Automation	To configure algorithms efficiently, instructing the computer to perform repetitive tasks (Rafiq et al., 2023) (Cansu & Cansu, 2019)	Primitive code, with little or no control over the basic automation mechanisms.	Use of simple automation mechanisms.	Advanced automation mechanisms (complex loops, functions, etc.) and variables are used.
Decomposition	Taking apart problems and breaking them into smaller and more understandable constituents ("Divide and Conquer" method) (Cansu & Cansu, 2019)	Errors in the decomposition of the parts of the problem, not achieving exactly the desired results.	Parts of the problem properly decomposed, but with certain disorder.	Parts of the problem well decomposed and clearly ordered.
Debugging	To test, trace, verify and modify results in order to refine solutions (Rafiq et al., 2023) (Melro et al., 2023).	Lack of testing and verification of each part of the program, obtaining results not very close to the desired ones.	Basic testing of the program parts. Results very close to the desired results.	All parts of the program have been tested, achieving the expected results. Effort in optimizing the code.
Generalization	To look for patterns, similarities and linkages into the different parts of the problem, in order to solve them with a common solution or algorithm (Rafiq et al., 2023) (Cansu & Cansu, 2019)	No search for common elements of the different parts of the problem. Non-modular programs. No common solutions are implemented.	Evidence of generalization, obtaining modular parts in the programs that allow some change in the variables.	Effort to generalize the different parts of the problem, using common elements such as variables, functions, etc.

Fig. 17. Evaluation rubric of the processes associated with CT. Definition of these processes and assessment criteria. Prepared by the authors.

The evaluation (Fig. 18) shows good results in the abstraction and decomposition processes, necessary to identify the parts of the problem and synthesize it, so that it can be transferred to the Processing environment. More difficulties are observed in the choice of algorithms and task automation. Although this may be partly due to a lack of fluency in the use of the Processing language, these are relatively novel thought processes for the students, with which they may not have had previous contact in previous educational stages or in other subjects in the degree program.

Regarding the generalization process, some difficulties have been identified when establishing connections between the different parts of the programs; not so much in terms of establishing common variables (practically all students have been able to

establish them), but in terms of the use of more complex mechanisms, such as custom functions written by the student or objects. These mechanisms were developed in a basic way in the course, but the results seem to indicate that they are more difficult to assimilate and may require more time in the classroom.
Finally, the students have carried out an iterative process of debugging their applications, correcting and testing each part of the program until the results were either as they expected, or at least close to their initial idea.

		Implementation of Computational Thinking processes *1 = low; 2 = medium; 3 = high*						
		Abstraction	Algorithmic thinking	Automation	Decomposition	Debugging	Generalization	Average
Student 1	P01	3,0	2,5	3,0	2,0	2,5	3,0	2,7
Student 2	P02	3,0	2,5	2,5	3,0	3,0	2,5	2,8
	P03	2,0	3,0	2,5	3,0	3,0	3,0	2,8
Student 3	P04	3,0	2,5	2,0	2,5	2,5	1,5	2,3
	P05	2,0	2,0	2,0	2,0	2,5	2,5	2,2
Student 4	P06	2,0	2,0	1,5	2,0	3,0	2,5	2,2
Student 5	P07	3,0	2,5	3,0	3,0	3,0	3,0	2,9
	P08	2,0	2,5	2,0	2,0	2,5	1,5	2,1
Student 6	P09	3,0	2,0	3,0	2,5	2,0	2,5	2,5
Student 7	P10	2,5	2,0	1,0	2,0	2,0	1,0	1,8
Student 8	P11	1,0	1,0	2,0	1,0	1,5	2,0	1,4
Student 9	P12	3,0	2,5	2,0	3,0	3,0	2,5	2,7
	P13	3,0	2,5	2,0	2,5	2,0	2,0	2,3
	P14	3,0	2,5	3,0	3,0	3,0	3,0	2,9
Average		2,5	2,3	2,3	2,4	2,5	2,3	2,4

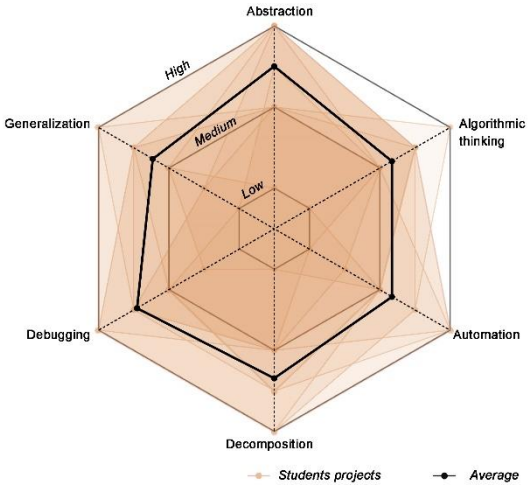


Fig. 18. Implementation level of CT processes in the development of students' final projects. Results. Prepared by the authors.

Finally, a comparison is made between the results of the three evaluations: the syntax test, the use of different programming functions and the implementation of processes associated with Computational Thinking (Fig. 19). It is observed that there is a relationship between the use of different Processing language mechanisms with the processes associated with CT, so that the greater the diversity of mechanisms, the greater the possibilities of developing processes such as abstraction, decomposition, debugging, etc.
It is also observed that the good results in the language syntax test are not necessarily related to the number of Processing mechanisms implemented in the final applications. This may be due to the fact that, although students apparently understand how the Processing language is structured, when proposing code they stick to those mechanisms that are easiest for them.

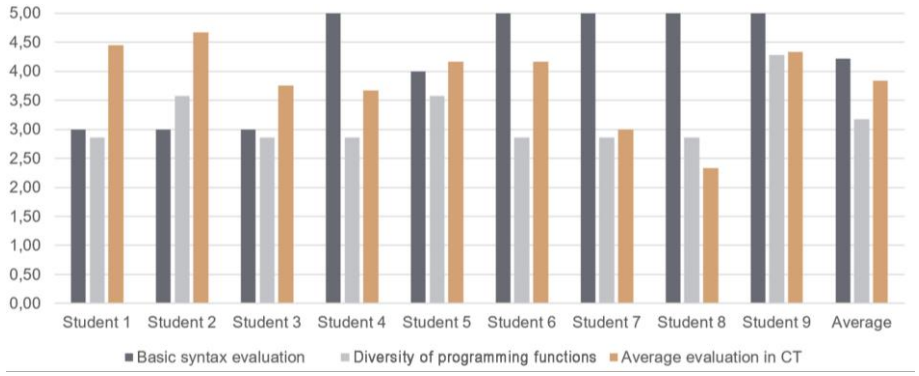


Fig. 19. Final comparison of the three evaluations (each evaluation weighted out of 5). Prepared by the authors.

5.2.3. Student assessment

Finally, students completed an anonymous questionnaire to assess their previous knowledge of programming, the degree of complexity of the course content, and the most difficult topics to comprehend. Results showed that nearly 75% of students had previous experience with programming (most encountered it in prior educational phases), demonstrating that programming languages as vehicles for learning in secondary education have been tried and tested.

As for the complexity of the curriculum, most students considered that the concepts they studied were somewhat complex. In a very detailed fashion, the topics that were most difficult to understand were the loops (for and while structures), logical operators (Boolean logic) and the execution of personalised functions (Fig. 20). Other mathematical aspects were more accessible to them. This is comprehensible, considering that Architecture is a technical degree that includes courses in calculus, algebra, and physics.

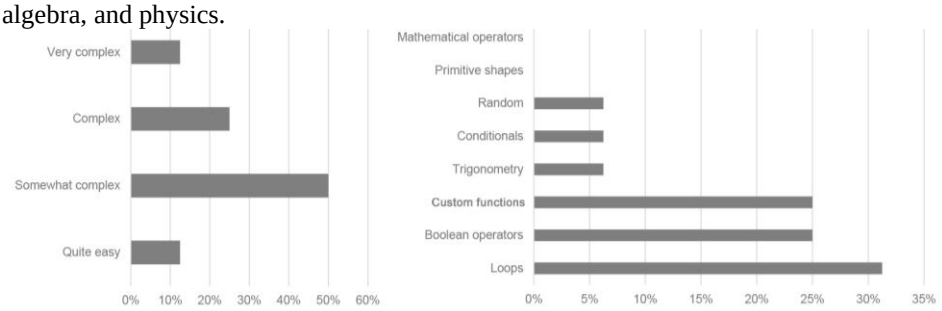


Fig. 20. Questionnaire results. Left: valuation of the degree of complexity of the content in the Processing course. Right: Valuation of the most challenging concepts to learn. Prepared by the authors.

6. Discussion

6.1. *Creativity and programming: CT as a way to conduct creativity*

The pilot course consisted of using the Processing language to generate designs by

means of the program features that the language offers, using problem-solving processes associated with computational thinking: abstraction, decomposition, etc.

Processing is called, according to its own creators (Reas & Fry, 2007), as a "creative" programming language, because of its application to the world of design and art. A more general definition of creative coding is provided by Romero, Lepage and Lille (2017): "creative programming engages the learner in the process of designing and developing an original work through coding".

But is it only creative that which is oriented towards original or artistic results? And how are computational thought processes related to creativity? To answer these questions, creativity should first be defined, and this question continues to be a source of discussion. The previous authors point out that "creativity is a context-related process in which a solution is individually or collaboratively developed and considered as original, valuable, and useful by a reference group" (Romero, Lepage, & Lille, 2017). This definition, oriented to the outcome of creativity, is strongly linked to the definition of creative processes that authors such as Torrance or Newell proposed in the 1960s. At that time, the definition of creative processes had special relevance in the field of computing, since it was hypothesized about computers that were able to think like human beings (Breton, 1989), so it was first necessary to unravel the processes of human thought. Thus, Torrance defined creative processes as those that satisfied these premises: "[1] the product of the thinking has novelty and value (…); [2] the thinking is unconventional, in a sense that it requires modification or rejection of previously accepted ideas; [3] the thinking requires high motivation and persistence (…); [4] the problem as initially posed was vague and undefined, so that part of the task was to formulate the problem itself" (Torrance, 1965).

However, other authors argue that creativity is not a process that can be evaluated by a concrete result. Marina (2006) proposes that creativity is the capacity of human intelligence to "work with unrealities" and "invent possibilities". He argues that it is not a system of answers (as some thinkers of computation and architects of artificial intelligence, such as Newell, argue), but a system of questions. In other words, it is not that there is a type of "creative thinking", but that human thinking itself is, by default and by definition, creative. Relying on this thesis, perhaps the distinction between creative thinking and other types of thinking can be set aside, and the processes associated with computational thinking can be included as concrete mechanisms that lead to creativity (Fig. 21).

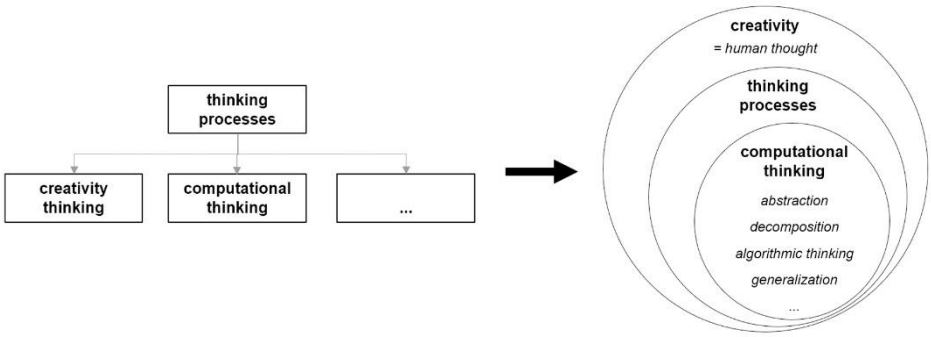


Fig. 21. Relationship between computational thinking and creativity. Left: diagram based on the distinction between creative thinking and computational thinking. Right: diagram based on Marina's thesis (2006). Prepared by the authors.

The programming exercises with the Processing language show in the results of this research that these processes (abstraction, algorithmic thinking, etc.) are activated in the development of computer applications (Fig. 22). Therefore, the contact with these processes is a "learning opportunity" (Melro et al., 2023), which goes beyond learning the programming language studied. Thus, there is a relationship between programming and creativity, where the first sparks a series of thinking mechanisms that guide the creative process.

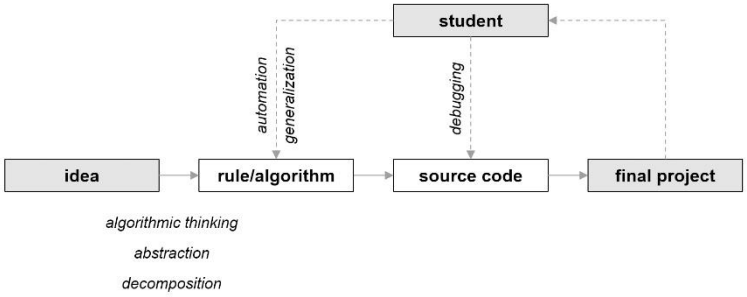


Fig. 22. Following the diagram of the phases of generative design (Fig. 3), it is shown where the CT processes intervene in the applications developed by the students. Prepared by the authors.

6.2. Questioning the Processing learning curve

Experts of the Processing programming language often highlight its steep learning curve. This is due to its simplicity (in comparison with other programming languages), its interface and the web resources of the Processing Foundation, which explains and provides examples for all the code's functions. There is a general consideration that in a short amount of time, Processing students can obtain complex graphic results (Fricker, Wartmann, & Hovestadt, 2008).

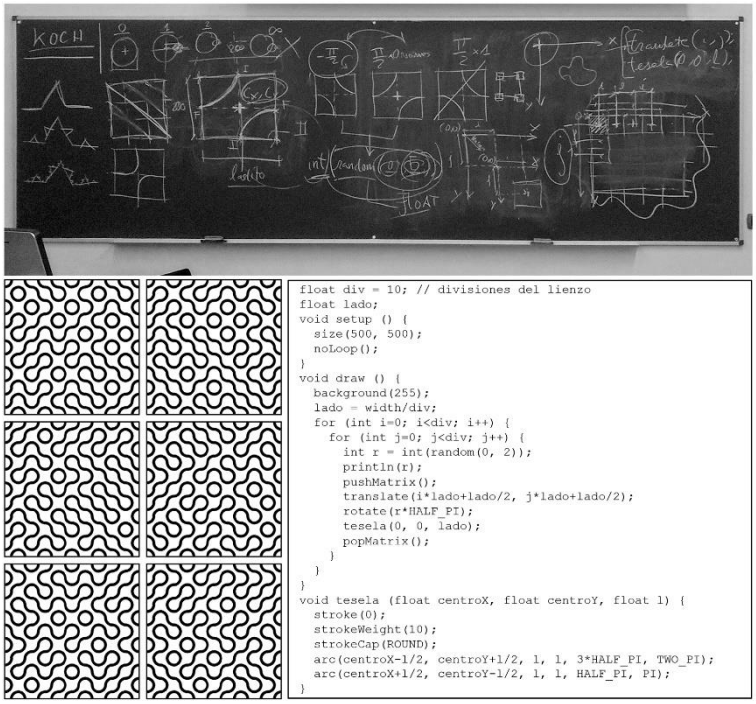
The premise, however, is arguable. The graphic results reached in this experimental course are more basic and primitive than those attained by students in other courses aimed at learning generative design tools. This may be explained by its short duration (fewer than 20 hours of training), and its pedagogical approach, geared more toward a general understanding of digital environments than a complete overview of all the Processing language functions. As revealed in the course questionnaires (Fig. 20), students found it more difficult to comprehend those concepts. No surprise there. As pointed out by Maeda (2019), the nature of digital environments follows very different rules than analogue environments, and these seem antinatural to human beings, for instance, unlimited repetition or exponential thinking. Lack of skills in the processes of algorithmic thinking and task automation were also detected (Fig. 18). As previously indicated, this may be caused by the fact that these processes are not particularly familiar to the students, unlike the abstraction or decomposition mechanisms, with which they have had contact in previous educational stages and in other subjects (such as mathematics, physics or structures).

Therefore, Processing's learning curve will also be determined by the handling of computational thought processes, and this will hardly be developed if a reasonable amount of time is not invested in its understanding and comprehension. Longer training

courses may be necessary, or programming may need to be integrated into the architecture curriculum by linking it to classical subjects such as Geometry or Architectural Design.

6.3. Paper and screen in reflective practice

It is interesting to note the use of paper or the blackboard as critical elements in analysis, both for explanations of theoretical concepts as much as in the problem-solving exercises the students worked on (Fig. 23): Processing serves to visualise abstraction (from code to image), but before that it becomes nearly indispensable to visualise the problem with small sketches to write the abstraction down (from image to code). That is why approaches that utilise creative programming languages, such as Processing, as substitutes for drawing ((Fricker, Wartmann, & Hovestadt, 2008) (Cannaerts, 2016)) may appear attractive in formal experimentation but may not be the most pedagogical method. Screens and paper feed off each other and enrich the learning and problem-solving processes of computation. Ergo their coexistence is much more convenient if the goal is to better understand digital concepts.




```
float div = 10; // divisiones del lienzo
float lado;
void setup () {
  size(500, 500);
  noLoop();
}
void draw () {
  background(255);
  lado = width/div;
  for (int i=0; i<div; i++) {
    for (int j=0; j<div; j++) {
      int r = int(random(0, 2));
      println(r);
      pushMatrix();
      translate(i*lado+lado/2, j*lado+lado/2);
      rotate(r*HALF_PI);
      tesela(0, 0, lado);
      popMatrix();
    }
  }
}
void tesela (float centroX, float centroY, float l) {
  stroke(0);
  strokeWeight(10);
  strokeCap(ROUND);
  arc(centroX-l/2, centroY+l/2, l, l, 3*HALF_PI, TWO_PI);
  arc(centroX+l/2, centroY-l/2, l, l, HALF_PI, PI);
}
```

Fig. 23. Guided exercise on mosaics using customized functions, generating a modular program. Top: blackboard explaining the concepts (drawn abstraction). Bottom left: initial images programmed with Processing. Bottom right: code solved. Prepared by the authors.

As demonstrated in the short example exercise in Fig. 9, the solving process is iterative,

going from paper to code and from code to the output graphic, only to return to the paper or the code, depending on the sort of problems the application must execute. As Richard Sennett (2009) pointed out about Linux programmers, the work process of producing computer applications is artisanal. There is “a nearly instantaneous relationship between the solution and discovery of problems”. This solution-problem-solution iteration, which in computer jargon is called "debugging", is common to craft creation processes and even to architectural design processes. In formal architectural generation, each action (sketch) leads to a solution but also to a new problem, which requires new sketches to solve (and, probably, will result in new problems to solve). Following the thesis of philosopher and pedagogue Donald Schön, it entails a very interesting and enriching reflective practice for the future professional (Schön, 1988), where each problem is solved whereby an investigation alternating action and reflection.

6.4. Programming mechanisms in artworks and architecture

The mechanisms that intervene in this iterative creative process are characteristic to CT: abstraction (reducing the complexity of the information), decomposition (breaking the problem down into simpler ones), algorithmic thinking (establishing procedures), etc. And, as mentioned above, Processing serves as a vehicle for using these mechanisms. In this way, the code, used as a medium, guides the design process (Cannaerts, 2016) and, by extension, the thinking process.
Operations characteristic to computation have been employed (combination, random, matrixes, iterations, etc.) in numerous artworks and architecture without the need for a programming language or a specific software. In the case of architecture (Fig. 24), these operations are carried out with the generation of diagrams, understood as “graphic organisation devices.” (Paredes, 2015) Using generative design tools such as Processing, in the project conception phases of architecture, allows for the production applications that generate different configuration and ordering schemes.

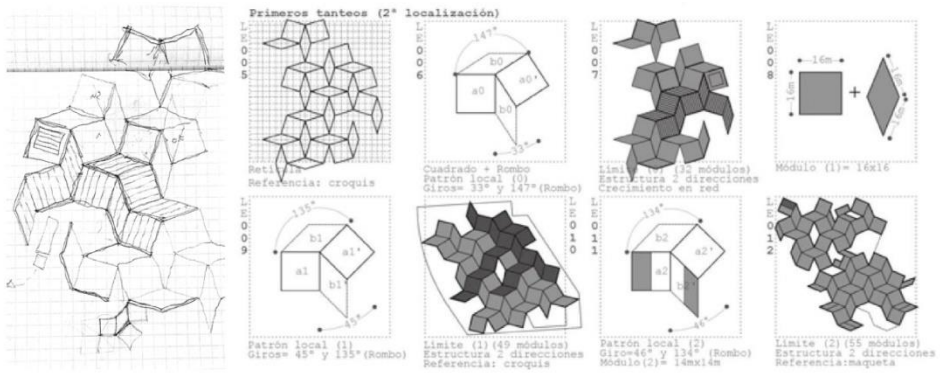


Fig. 24. Tuñón y Mansilla (2005), Museo de Arte Contemporáneo de Castilla y León MUSAC. Left: sketch of plant formalisation process (González, Mendoza, & Pina, 2020). Right: hypothesis of geometric development of the project (González, Mendoza, & Pina, 2020). The formal genesis operations conducted in the design of this building are also computable in nature, as combination or repetition.

On the other hand, the use in the experimental course of existing works as computable case studies, whether architectural or pictorial, has signified pedagogical enrichment. The work serves as a vehicle to understand programming languages (students internalise the functioning of the language while they transcribe the work to the Processing environment). The programming language contributes to comprehending the work (as it is dissected from a digital perspective. It is decomposed into simpler parts, and the algorithms that generated are found, etc.).
This way of using artistic works to break them down from a digital perspective to comprehend them and simultaneously understand the nature of programming could develop into a training on CT from a much more theoretical and radical point of view: with paper and pencil only, without any programming language to serve as a vehicle. This is what some authors refer to as "unplugged computational thinking" (Osio & Bailón, 2020)), and it follows Steele's perspective when discussing teaching a "theory of computers" applied to teaching in architecture (Steele, 2001).

6.5. Analogue processes vs. digital processes

The random function was one of the Processing functions students explored the most in their personal final projects. It includes data (parameters) that depend exclusively on a computer-generated random number. Therefore, the results that are obtained are indeterminate, generating a surprise effect either in the image or final animation and yielding new possibilities in terms of interpretations and formal configurations that designers perhaps would not have drawn on paper and that may lead them to investigate new possibilities. The underlying idea is that a dialogue in the design process between students and computers, as highlighted by Negroponte (1970) with the man-machine conversation, may enrich the creative process.
Analogue graphic processes for thinking up an architecture project are based on paper attempts and, as pointed out by Llopis (2018), "the possibilities for formal evolution are relatively moderate, insofar as the possibility of suggesting alterations to the original idea depends on our capacity to attempt them manually." Tedeschi (2014) also states that hand drawings are a part of the direct relationship between ideas and graphic signs, constituting an additive information process on paper.
Both authors, however, hold that the use of digital tools (generative design tools such as Processing) in creative processes follows a very different scheme: A system of relationships must be established with very concrete rules (algorithms) that limit the form, the way computers generate solutions in response to initial conditions (Llopis, 2018) (Tedeschi, 2014). This is what architect Kostas Terzidis denominates "algotecture" ("a term to denote the use of algorithms in architecture"), defending the use of algorithms in architectural design for the possibilities it brings in terms of establishing new relationships: "algorithms can be used to solve, organise or explore problems with increased visual or organisational complexity" (Terzidis, 2006).

7. Conclusions

The contributions made in the present research in relation to the initial background and to other ongoing research of similar characteristics cover different fields. Both in

teaching and in the ideation of the architectural discipline, creative programming is an interesting alternative to conventional architectural design-oriented programs. Most of these programs are focused on the generation of photorealistic images (Frazer, 2005), and their pedagogy suffers from a "technocentric" approach (Papert, 1987), as it is generally reduced to the development of the technical aspects of each program (Steele, 2001). The complex and diverse interfaces of each software not only require a great investment of time in their memorization, but also force designers (students or architects) to assimilate the abstractions (or metaphors) of each command and to couple them in their creative process. As Lawson points out, "the problem is that if the computer uses the wrong metaphor for describing design features, it can inhibit the creative integration" (Lawson, 2002).

From this perspective, working directly with the digital syntax and not with complex abstractions is one of the strengths of creative programming languages. Processing, in this case, represents a major difference with respect to other tools in the field of generative computer design. This language employs mechanisms characteristic of the digital environment that architects can incorporate into their design process through relatively simple algorithms, building digital diagrams of the project (Terzidis, 2006).

In addition, the methodology of prototyping software applications involves knowing how to handle the processes of computational thinking (Melro et al., 2023) (Romero et al., 2017): abstraction, algorithmic thinking, automation, decomposition, debugging and generalization (Cansu & Cansu, 2019) (Rafiq et al., 2023) (Wing, 2008). It is thus demonstrated that problem solving processes, widely known in programming environments, are also applicable to the architectural discipline, being used to drive creativity and formal generation (Cannaerts, 2016).

In this sense, one of the major contributions made in the research is the methodology of teaching and implementation of the Processing language in architecture. This has been focused on studying those programming functions oriented to build algorithms and to solve problems by combining sketch-based reasoning (problem visualization) with code writing in an iterative process. This methodology takes advantage of the visual power of Processing (graphical output) and the visual power of traditional architectural thinking (drawing), overcoming the traditional dichotomy between digital and manual teaching and promoting new integrative didactics (Raposo et al., 2022).

Future developments of this research include the implementation of creative programming in the architecture curriculum. It could be studied how it fits into existing subjects oriented to work with complex geometries or those more related to architectural ideation. Also, as sketched by James Steele (2001), a "computer theory" could be incorporated, in which perhaps the structure of the digital and its impact on architectural design could be discussed, using a programming language (in this case, Processing) as a means to approach these concepts.

References

Agkathidis, A. (2012). *Computational architecture*. BIS Publishers, Amsterdam.

Agkathidis, A. (2015). *Generative design: form-finding techniques in architecture*. Laurence King Publishing, London.

Allen, S. (2009). Velocidades terminales: el ordenador en el estudio de diseño. In: Ortega, L. (ed.), *La digitalización toma el mando*. Gustavo Gili, Barcelona, 39-

57 (Original work published in 1995).
Allen, S. (2009). El complejo digital: diez años después. In: Ortega, L. (ed.), *La digitalización toma el mando*. Gustavo Gili, Barcelona, 159-168 (Original work published in 2005).
Bohnacker, H., Gross, B., Laub, J., & Lazzeroni, C. (2012). *Generative design: visualize, program, and create with Processing*. New York: Princeton Architectural Press.
Breton, P. (1989). *Historia y crítica de la informática*. Cátedra, Madrid.
Burry, M., Datta, S., & Anson, S. (2000). Introductory computer programming as a means for extending spatial and temporal understanding, Eternity, Infinity and Virtuality in Architecture. In: *Proceedings of the 22nd Annual Conference of the Association for Computer-Aided Design in Architecture*, Deakin University, Washington D.C., 129-135.
Caetano, I., Santos, L., & Leitão, A. (2020). Computational design in architecture: Defining parametric, generative, and algorithmic design. *Frontiers of Architectural Research*, 9, 287-300. https://doi.org/10.1016/j.foar.2019.12.008
Cannaerts, C. (2016). Coding as a creative practice. In: *Proceedings of the 34th eCAADe Conference* (1), Oulu, Finland, 397-404.
Cansu, F. K., & Cansu, S. K. (2019). An overview of computational thinking. *International Journal of Computer Science Education in Schools*, 3(1), 17-30. https://doi.org/10.21585/ijcses.v3i1.53
Cardoso, D. (2013). Algorithmic Tectonics: How Cold War Era Research Shaped Our Imagination of Design. *Architectural Design*, 83(2), 16-21. https://doi.org/10.1002/ad.1546
Domínguez, P., Celis, F., & Echeverría, E. (2022). How the Approach of Digital Tools in Architecture Has Developed: The Case of Creative Programming. In: Ródenas-López, M.A., Calvo-López, J, Salcedo-Galera, M. (Eds.), *Architectural Graphics. EGA 2022. Springer Series in Design and Innovation* (Vol. 22). Springer. https://doi.org/10.1007/978-3-031-04703-9_39
Frazer, J. (2005). Ordenar sin ordenador. In: Ortega, L. (ed.), *La digitalización toma el mando*. Gustavo Gili, Barcelona, 169-179.
Fricker, P., Wartmann, C., & Hovestadt, L. (2008). Processing: Programming Instead of Drawing. Experimental Use of an Open-source Programming Language within the Architecture Curriculum. In: *Architecture 'in computro': integrating methods and techniques: proceedings of the 26th conference on education and research in computer aided architectural design in Europe*, Antwerpen, 525-530.
Galanter, P. (2016). *Generative art theory*. John Wiley & Sons, Hoboken.
Garber, R. (2014). *BIM design: realising the creative potential of building information modelling*. Wiley, Chichester.
González, A. J., Mendoza, N. M., & Pina, R. (2020). La geometría oculta del MUSAC. *Cuadernos de Proyectos Arquitectónicos* (10), 66-79. https://doi.org/10.20868/cpa.2020.10.4563
Greenberg, I., Xu, D., & Kumar, D. (2013). *Processing: creative coding and generative art in processing 2*. Friends of Ed, New York.
Guzdial, M., Kay, A., Norris, C., & Soloway, E. (2019). Computational thinking should just be good thinking. Seeking to change computing teaching to improve

computer science. *Communications of the ACM*, 62(11), 28-30. https://doi.org/10.1145/3363181

Hovestadt, L., *Digital Architectonics Primer: Introduction* (Online presentation). https://www.youtube.com/watch?v=rrsbs4e-xTE (visited on August 26, 2023)

Hovestadt, L. (2021). *Mathematical Thinking and Programming* (subject curriculum, ETH Zurich). http://www.vvz.ethz.ch/Vorlesungsverzeichnis/lerneinheit.view?semkez=2021W&ansicht=KATALOGDATEN&lerneinheitId=147470&lang=en (visited on August 26, 2023).

Joyanes, L. (2008). *Fundamentos de programación: Algoritmos, estructuras de datos y objetos*. (4th ed.). McGraw-Hill, Madrid.

Lawson, B. (2002). CAD and creativity: Does the computer really help? *Leonardo*, 35 (3), 327-331. https://doi.org/10.1162/002409402760105361

Llopis, J. (2018). *Dibujo y arquitectura en la era digital*. Editorial de la Universidad Politécnica de Valencia, Valencia.

Maeda, J. (2019). *How to speak machine: laws of design for a computational age*. Portfolio/Penguin, New York.

Marina, J. A. (2006). *Teoría de la inteligencia creadora*. Anagrama, Barcelona.

Melro, A., Tarling, G., Fujita, T., & Staarman, J. K. (2023). What else can be learned when coding? A configurative literature review of learning opportunities through computational thinking. *Journal of Educational Computing Research*, 61(4), 901-924.

MNCARS (Museo Nacional Centro de Arte Reina Sofía) (2020). Aquí no hay nada que comprender. Un documental sobre Elena Asins [Film]. https://www.museoreinasofia.es/actividades/documental-elena-asins (Visited on August 26, **2023**).

Negroponte, N. (1970). *The architecture machine*. MIT Press, Cambridge.

Osio, U., & Bailón, J. (2020). Pensamiento computacional. Alfabetización digital sin computadoras. *La Revista Icono* (14), 379-405. https://doi.org/10.7195/ri14.v18i2.1570

Papert, S. (1987). Computer criciticsm vs. technocentric thinking. *Educational Researcher*, 16(1), 22-30. https://doi.org/10.2307/1174251

Paredes, M. (2015). Diagrams as a large-scale generative systems. *EGA Expresión Gráfica Arquitectónica*, 20(25), 168-179. https://doi.org/10.4995/ega.2015.1287

Processing Foundation. (s.f.). processing.org. https://processing.org/ (Visited on August 26, 2023).

Rafiq, A. A., Triyono, M. B., Djatmiko, I. W., Wardani, R., & Köhler, T. (2023). Mapping the Evolution of Computational Thinking in Education: A Bibliometrics Analysis of Scopus Database from 1987 to 2023. *Informatics in Education*, 22(4), 691-724. doi:10.15388/infedu.2023.29

Raposo, F. J., Salgado, M. A., Butragueño, B. (2022). Del dibujo analógico al dibujo digital. La construcción virtual de la arquitectura como algo más que una implementación tecnológica. In: Jiménez, P. M., Mestre, M. y Navarro, D. (Eds). *Más allá de las líneas. La gráfica y su uso. XIX Congreso Internacional de Expresión Gráfica Arquitectónica.* Universidad Politécnica de Cartagena, Cartagena, 245-248.

Reas, C., & Fry, B. (2007). *Processing. A programming handbook for visual designers and artists*. MIT Press, Cambridge.

Romero, M., Lepage, A., & Lille, B. (2017). Computational thinking development through creative programming in higher education. *International Journal of Educational Technology in Higher Education*, 14(42). https://doi.org/10.1186/s41239-017-0080-z

Schön, D. (1988). *Educating the reflective practitioner*. Jossey-Bass Publishers, San Francisco.

Schwill, A. (1994). Fundamental ideas of computer science. *European Association for Theoretical Computer Science*, 53, 274-295.

SCImago. Scimago Institutions Ranking. https://www.scimagoir.com/ (Visited on June 21, 2023).

Sennett, R. (2009). *The Craftsman*. Penguin, London.

Shangai Ranking. Academic Ranking of World Universities. Obtenido de https://www.shanghairanking.com/rankings/arwu/2022 (Visited on June 21, 2023).

Steele, J. (2001). *Architecture and computers: action and reaction in the digital design revolution*. Laurence King, London.

Tedeschi, A. (2014). *AAD Algorithms Aided Design. Parametric strategies using Grasshopper*. Le Penseur, Brienza.

Terzidis, K. (2006). *Algorithmic architecture*. Elsevier, Oxford.

Times Higher Education. World University Rankings. Obtenido de https://www.timeshighereducation.com/world-university-rankings (Visited on June 21, 2023).

Torrance, E. P. (1965). Scientific views of creativity and factors affecting its growth. *Daedalus*, 94(3), 663-681.

Virilio, P. (1999). *El cibermundo, la política de lo peor*. Cátedra, Madrid.

Virilio, P. (2016). *La administración del miedo*. Pasos Perdidos, Madrid.

Wing, J. (2008). Computational thinking and thinking about computing. *Philosophical Transactions: Mathematical, Physical and Engineering Sciences*, 366(1881), 3717-3725. https://doi.org/10.1098/rsta.2008.0118

Wing, J. (2011). Research Notebook: Computational Thinking--What and Why? *The Link. The magazine of Carnegie Mellon University's School of Computer Science*, 20-23. https://www.cs.cmu.edu/link/research-notebook-computational-thinking-what-and-why

Patricia Domínguez-Gómez is graduated in architecture from the University of Alcalá de Henares (2016) and is associate professor at the department of architecture of University of Alcalá, in the area of architectural graphic expression. She's also a PhD candidate at the same university, focusing on technological innovations in architecture. She belongs to the Alcalá University research group 'Intervention in heritage and sustainable architecture' and to the education innovation group 'Graphic, visual and geometric tools', where she works on the application of new digital technologies in architecture and design.

Flavio Celis d'Amico is Professor at the School of Architecture (University of Alcalá, Spain). He received his B.A. from the Polytechnic University of Madrid (Spain) in 1990 and his PhD in 1998. His main areas of research are the documentation and conservation of heritage linked to bioclimatic architecture and environmental sustainability. At present he belongs to the Alcalá University research groups: 'Sciences of Archeology' and 'Intervention in heritage and sustainable architecture'. He has participated in several projects linked with heritage and sustainability as an architect and researcher. He has worked in Spain, and some other places: Egypt, Brasil, Italy, Portugal, Chile, México, Guatemala, China, Bolivia, focused on heritage conservation, documentation, and environmental sustainability.