Comprehending Algorithmic Design

Renata Castelo-Branco(✉) and António Leitão

INESC-ID/Instituto Superior Técnico, University of Lisbon, Lisbon, Portugal
{renata.castelo.branco,antonio.menezes.leitao}@tecnico.ulisboa.pt

Abstract. Algorithmic Design (AD) allows for the creation of form through algorithms. Its inherent flexibility encourages the exploration of a wider design space, the automation of design tasks and design optimization, considerably reducing project costs and environmental impact. Nevertheless, current AD uses representation methods that radically differ from those used in architectural practice, creating a mismatch that is further exacerbated by the inadequacy of current programming environments. This creates a barrier to the adoption of AD, demotivating architects from its use.

We propose to address this problem by coupling AD with adequate representation methods for designing complex architectural projects. To this end, we explore three essential concepts: storytelling, interactive evaluation, and reactivity. These concepts can be both complementary and mutually exclusive, which means compromises must be made to accommodate them all. We outline a strategy for their integration with the AD workflow, highlighting the advantages and disadvantages of each one, and pinpointing their intersection. Finally, we evaluate the proposed strategy using computational notebooks as programming environments.

Keywords: Algorithmic design · Program comprehension · Documentation · Storytelling · Liveliness · Interactive evaluation · Reactivity · Computational notebooks

1 Introduction

Algorithmic Design (AD) defines the creation of designs through algorithmic descriptions, i.e., computer programs with instructions for the machine to perform [8]. Despite its numerous advantages, there is a comprehension barrier demotivating many architects from its use. The goal of this research is to make AD a more accessible design process, allowing the industry to benefit from AD's potential to combine design creativity and design optimization. In order to do so, we will dive into program comprehension research, identifying strategies that aid the construction and comprehension of AD programs.

1.1 Algorithmic Design

AD allows the designer to delegate repetitive tasks to the computer, accelerating the production process and reducing human errors [8]. It also supports rapid change with little effort, providing considerable cost savings to the industry [51].

Another important advantage is AD's influence on performance-based design [20], which is gaining ever more emphasis with the growth of climate change awareness and cost-reduction needs. While the incorporation of analysis data in early design stages alone already helps architects achieve better performing solutions, the connection between AD and simulation tools opens the door to much faster optimization processes [34].

Despite the numerous advantages AD presents to the process of architectural creation, it relies on algorithms, written using programming languages. This representation method radically differs from the ones traditionally used in design and is therefore difficult to use by professionals of creative fields [32].

Visual programming languages present a friendlier approach to AD that simplifies the learning process and offers more adequate graphical features for the task at hand. However, they also lack scalability [7], meaning that as programs grow in complexity they become hard to understand and navigate [9,11], hindering their use in large-scale projects [28].

Additionally, and in either case, AD programs (visual or textual) frequently turn up to be the unstructured product of successive *copy&paste* of program elements, which result from the experimentation process that characterizes design thinking [51]. Hence, programming architects find it difficult to understand AD programs that represent complex designs, particularly those developed by others [30] or by themselves in the past. This research targets the development of AD programs that describe complex 3D models, hence we focus on text-based AD.

1.2 Integrated Development Environments

Much of the programming struggle mentioned above can be alleviated with an Integrated Development Environment (IDE): a computer application that aims at facilitating the programming task. Sadly, existing IDEs are tailored for traditional software development processes. As such, architects find it hard to use them as design tools.

We propose to make AD more akin to the traditional architectural practice by integrating more adequate methods of developing and maintaining algorithmic representations of complex architectural projects in current IDEs. To that end, we explore two ideas that facilitate program comprehension: (1) documentation - the task of explaining a project to facilitate its comprehension; and (2) liveliness - the ability to live test the program as it is being developed or modified to facilitate comprehension of the impact of changes.

Documentation and liveliness are two very broad concepts, which we further subdivide to better suit architectural design. More specifically, we explore: (i) storytelling, (ii) interactive evaluation, and (iii) reactivity. These three concepts contribute to the comprehension of AD programs in very different ways and they can be both complementary and mutually exclusive. In this article, we outline a strategy for their integration with the AD workflow, which we evaluate using computational notebooks as the base IDE for the experiments.

2 Related Work

The difficult task of understanding computer programs is far from being limited to the architectural community. Research in the field of program comprehension has long aimed at modeling a cognitive theory on how programs are understood [48]. The following sections present some of the theories developed over time.

2.1 Program Comprehension

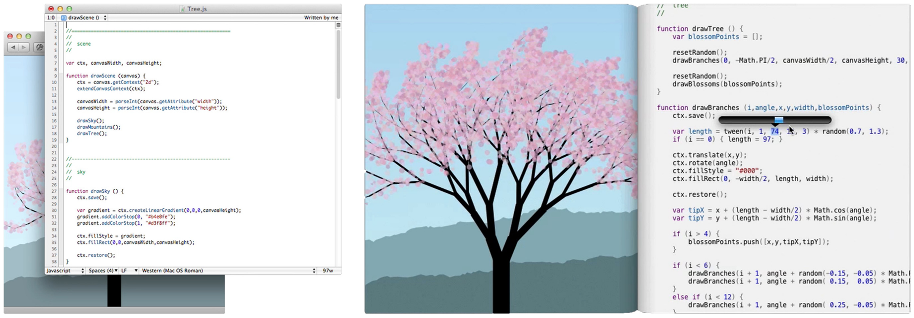
This section addresses two main branches of program comprehension: the understanding of computer programs through explanatory human-readable text and the use of graphical representations to illustrate programs' structure and behavior.

Program Documentation. In 1984, Donald Knuth proposed Literate Programming [25], a system that encourages users to build programs as a structured web of ideas, that is, by creating program parts and stating the relationships between them as they go, in whatever order they find best for the comprehension of the work. The problem with this solution is the perceived lack of efficiency: having to explain, in advance, the intended program, delays the programming task. Documentation is generally agreed to be one of the most useful, yet dreadfully tiresome parts of the job, and consequently one of the most avoided [5].

Naturally, given its importance for software maintenance, automatic documentation tools have also been developed [16]. Machine learning techniques, such as neural networks, in particular, are currently achieving considerable success in the automation of documentation [21,36]. However, for the specific case of architecture, traditional program documentation fails to ensure program comprehension, as it is geared to the production of hyperlinked texts. In AD programs, documenting complex geometry requires not textual explanations, but rather the equivalent sketch translation.

Bret Victor [50] argued that, when designing, artists think visually. However, when writing code, they must think linguistically. Not only is this translation process hard on design ideas, but most often it is also impossible to convey the entirety of artistic meaning with words. He also states that just as we created writing to make thoughts visible (a user interface for reason), or mathematical notation to make mathematical structures visible (a user interface for algebra), we must also develop IDEs that can make our designs visible [49] (Fig. 1).

Program Visualization. This specific branch of program comprehension research focuses on the use of graphical systems to facilitate the comprehension of programs. It began to have some expression in the '60s to help programmers deal with the complexity of (what was then considered) modern software [2]. The proposals included the use of various diagrammatic techniques to explain programs [18,22,33] (examples in Fig. 2).



```
//==================================================
//
//  scene
//

var ctx, canvasWidth, canvasHeight;

function drawScene (canvas) {
    ctx = canvas.getContext("2d");
    extendCanvasContext(ctx);

    canvasWidth = parseInt(canvas.getAttribute("width"));
    canvasHeight = parseInt(canvas.getAttribute("height"));

    drawSky();
    drawMountains();
    drawTree();
}


//--------------------------------------------------
//
//  sky
//

function drawSky () {
    ctx.save();

    var gradient = ctx.createLinearGradient(0,0,0,canvasHeight);
    gradient.addColorStop(0, "#b4e0fe");
    gradient.addColorStop(1, "#d3f8ff");

    ctx.fillStyle = gradient;
    ctx.fillRect(0,0,canvasWidth,canvasHeight);

    ctx.restore();
}
```

```
//  tree
//

function drawTree () {
    var blossomPoints = [];

    resetRandom();
    drawBranches(0, -Math.PI/2, canvasWidth/2, canvasHeight, 30,

    resetRandom();
    drawBlossoms(blossomPoints);
}

function drawBranches (i,angle,x,y,width,blossomPoints) {
    ctx.save();

    var length = tween(i, 1, 74, 1[illegible], 3) * random(0.7, 1.3);
    if (i == 0) { length = 97; }

    ctx.translate(x,y);
    ctx.rotate(angle);
    ctx.fillStyle = "#000";
    ctx.fillRect(0, -width/2, length, width);

    ctx.restore();

    var tipX = x + (length - width/2) * Math.cos(angle);
    var tipY = y + (length - width/2) * Math.sin(angle);

    if (i > 4) {
        blossomPoints.push([x,y,tipX,tipY]);
    }

    if (i < 6) {
        drawBranches(i + 1, angle + random(-0.15, -0.05) * Math.
        drawBranches(i + 1, angle + random( 0.15,  0.05) * Math.
    }
    else if (i < 12) {
        drawBranches(i + 1, angle + random( 0.25, -0.05) * Math.
```

Fig. 1. Bret Victor's Inventing on Principle: from a simple code editor (left) to an enhanced IDE (right) providing an "immediate connection" to what is being created (adapted from [49]).

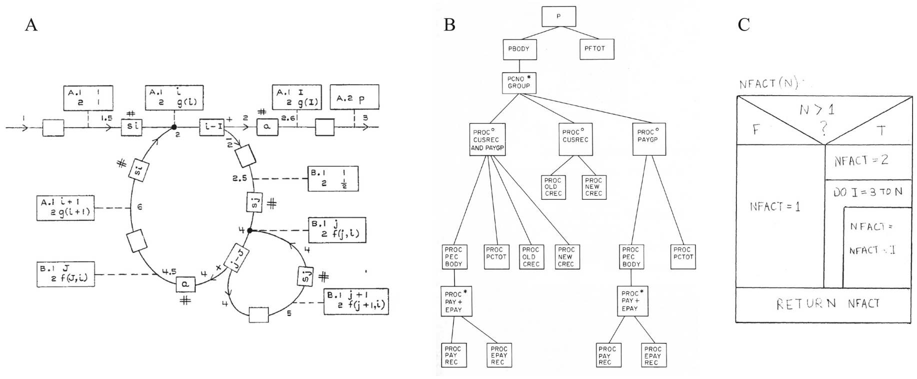


Fig. 2. A - Goldstine and von Neumann's flow-diagram [18]; B - Jackson diagram for a process payment program [22]; C - Nassi-Shneiderman diagram for the factorial function [33].

As technology evolved, so did the aspirations of the scientific community [12] and the availability of better displays, colors, and, later on, 3D representations, motivated a growing emphasis on animations over static representations of programs. Early on, Brown developed several systems that offered programmers dynamic displays of the program's fundamental operations [6].

Naturally, different programming paradigms motivate different types of visualization. For instance, declarative programming visualization requires presenting what the program does, while imperative programming also requires visualizing how it does it. Following this line of thought, different authors focused on different aspects of program visualization. Myers [32] classified program visualization systems according to what they illustrate (data, code, or algorithm) and how (statically or dynamically). Price et al. [40] further subdivided the field into a more hierarchical taxonomy, and many others followed [4,43].

Diehl [12] stressed that it is not only important to comprehend what a program does and how it does it, but also how we got to that point in the development process. The current widespread use of version control systems for both individual and collaborative work stands as proof to this, although their reliance on purely textual mechanisms hardly places them within program visualization.

A program visualization feature that fully exploits the visual nature of AD is traceability, that is, the identification of which parts of the model correspond to which parts of the program, and vice-versa [29]. This connection established between the program and respective result is of utter importance for the comprehension of AD programs [49], and several AD tools offer it already, e.g., Dynamo and Rosetta [29].

2.2 Liveliness

The dynamic component of program visualization introduced terms such as liveliness, interactivity, or reactivity, which became the main agenda for IDE developers invested in program comprehension.

Live Coding. The introduction of live coding blurred the lines between programming and explaining, advocating a real-time connection between program and result [42]. Live coding is frequently described as a creativity technique centered upon the writing of interactive programs on the fly [42]. In many cases, the liveliness requirement addresses the timing needs of live performances, such as musical ones [47]. However, its application to the programming task also helps users relate the changes in the program to their respective impact on results, thus aiding program comprehension.

In the case of AD, live coding requires the model to be quickly recomputed for every change applied to the program, which can be difficult to ensure when the program generates a complex model. For simple cases, however, there are already competent solutions integrated with AD, such as Grasshopper and Luna Moth [1].

Interactive Evaluation. Another perspective on liveliness that also aids comprehension without implying a scalability issue is interactive evaluation. By opposition to batch-evaluation, where the entire program must be loaded prior to execution, interactive evaluation motivates users to test small program fragments, debugging the program as it is being constructed [23]. It is frequently used for exploratory programming [42], a workflow typically employed when the requirements of the program are not fully defined. Hence the programming task becomes more of an exploratory experiment, a common scenario in AD.

Despite the advantages, there is a catch, particularly for beginners. Interactive evaluation promotes *ad-hoc* program construction that can introduce confusing bugs for developers unfamiliar with the obstacles of program state [15]. Experienced programmers avoid this by using a batch-oriented style in interactive systems, resetting the program after any major change, an approach that

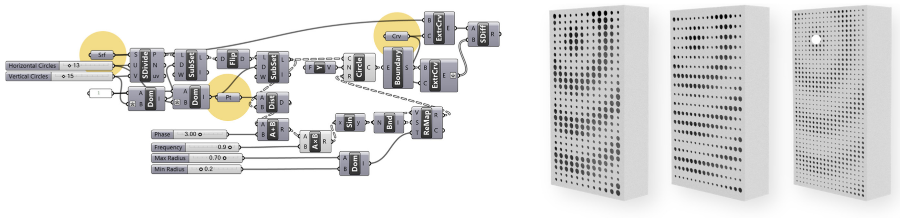


Fig. 3. Grasshopper diagram from [45].

is enforced by some pedagogical IDEs [14,15]. Once more, the programming paradigm used is also relevant, as declarative programming languages minimize the negative effects of program state when compared to imperative ones.

Reactivity. Reactivity is another take on liveliness that also advocates exploratory programming. However, the focus lies on tracking the dependencies in the program so that any change updates the entire state, as it happens in spreadsheets, for instance. The central idea is to have the system automatically manage dependencies to free the user from such burden [3]. With reactivity, since any change to the program triggers the re-evaluation of all dependencies, the program state is always consistent. As such, this paradigm also provides an alternative solution for the state problem introduced by interactive evaluation.

The data flow paradigm [27] is a good example of reactive programming. Here, programs are described by graph structures with nodes representing the functions and the wires that connect them representing the data that flows between components whenever something changes upstream. Most visual programming languages used in architecture, such as Grasshopper, are literal interpretations of this paradigm (Fig. 3). Note, however, that in what regards program comprehension, the data flow paradigm is flawed, since the node structure hides program complexity instead of explaining it, while also promoting program fragment repetition.

3 The Program Comprehension Dyad

To promote the use of AD, we propose a new design medium that allows for the creation of algorithmic descriptions in a live and documented way, making the task of understanding and changing AD programs easier for both the creator and others. To this end, we explore a program comprehension dyad: (1) documentation and (2) liveliness. These are two very broad concepts, which we fine-tune for an ideal IDE for the development of AD projects. Documentation is narrowed down to the concept of (i) storytelling, and liveliness is subdivided into (ii) interactive evaluation and (iii) reactivity. The following paragraphs summarize the definitions we consider for each of these concepts.

(1) **Documentation**, in the programming context, refers to the task of explaining the program, to facilitate later comprehension to the authors and other readers. The concept encompasses most of the research made under the program comprehension and visualization umbrellas, with emphasis on static visualizations. Documentation can be done prior, during, and after the program development, and it can also be internal and external to the program.

(i) Our proposal for the integration of documentation in the context of AD narrows this concept down to **storytelling**: the creation of a tale for the program's evolution. Storytelling thus contemplates internal documentation done while programming and aims at creating a narrative of the program development history, with program and respective documentation intertwined.

(2) **Liveliness** is the ability to live-test programs. Liveliness accommodates the dynamic or animated part of program visualization and subsequent branches. Liveliness can manifest itself in many ways, including live coding, interactive evaluation, and reactivity. Given that live coding tends to suffer from scalability issues, we focus on the two latter concepts.

(ii) **Interactive evaluation** allows a two-way flow of information between the computer and the user. With the user controlling just how lively a system is, the scalability problem becomes less relevant, although it increases the risk of program state inconsistencies.

(iii) In the context of AD, we consider **reactivity** as an automatic response that maintains state consistency by reevaluating all program parts that depend, directly or indirectly, from a part that was changed. This is more efficient than reevaluating the entire program, as is typically done in live coding, thus delaying but not eliminating the scalability problems that tend to affect complex AD programs.

Storytelling, interactive evaluation, and reactivity intersect in many ways. They can both complement and hinder each other. In the following sections, we explore this relationship and point out mechanisms to elude possible conflicts. Two case study project adaptations will be used to illustrate the proposed concepts: BIG Architects' Business Innovation Hub for the Isenberg School of Management in Amherst, Massachusetts (Fig. 4, left); and Santiago Calatrava's Liège-Guillemins railway station in Liège, Belgium (Fig. 4, right).

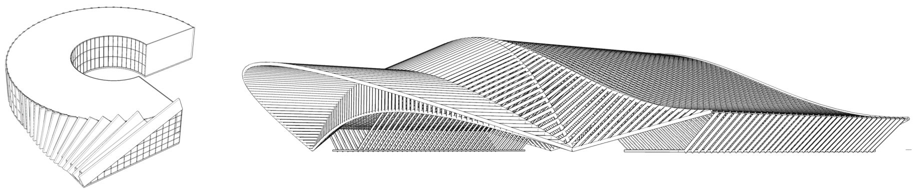


Fig. 4. Case study projects: Isenberg on the left and Liège-Guillemins on the right.

3.1 Storytelling

In an ideal IDE for AD, designers should be allowed to keep the artifacts produced along the design process in an organized fashion, explaining and documenting the resulting program. Storytelling intends to transform the program into a creative journal of the design's development that includes not only textual documentation but also sketches the architect made during a creative sprout [13]. Drawings are *"essential to both public discourse about architecture and the development of the architect's thinking"* [46, pp. 58]. Following this reasoning, we argue that the drawings produced when idealizing the design and the way it translates into a program (Fig. 5) can help document the program's evolution and expected outcome.

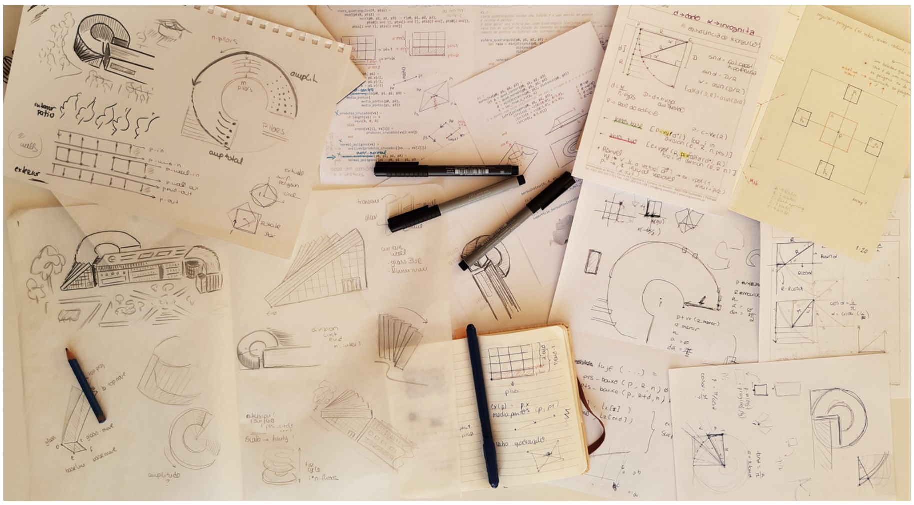


Fig. 5. Sketches made while developing the Isenberg school program.

Static images of the program's output, such as snapshots or renders of the generated model, are also relevant for the comprehension process. Very frequently, we introduce bugs in the program without noticing. Having a correct version (or a version of what the author considered to be the proper behavior of the program) available for comparison, may prove vital in debugging.

Withal, it is important to note that architectural design is by no means a linear process: upon seeing the (computational) results of their designs, architects frequently step back and change the design concept, rendering much of the documentation out of date. Storytelling embraces this issue by applying Diehl's concept of software evolution [12] to the context of architectural design: new documentation artifacts should be added whenever the program changes its intended behavior, thus keeping the history of design changes in the AD program.

3.2 Interactive Evaluation

Designers should feel motivated to construct and test their programs interactively, that is, executing programs parts immediately after (re)writing them. Interactive evaluation transforms what could otherwise be a very abstract process into a tangible and relatable one.

Interactive evaluation of AD programs typically produces visual results, which can become documentation artifacts. Given the proposed storytelling approach, we adapt our take on liveliness to accommodate the narrative style as well. Figure 6 presents the interactive development process of the slabs in the Isenberg project: an initial iteration of the slab function creates the intended contour; modifications are applied to contemplate the ground-floor exception; and later iterations convert the contour into 3D objects distributed along the height of the building. Each change is followed by a test that produces these visual results and the entire process is kept and documented in the AD program.

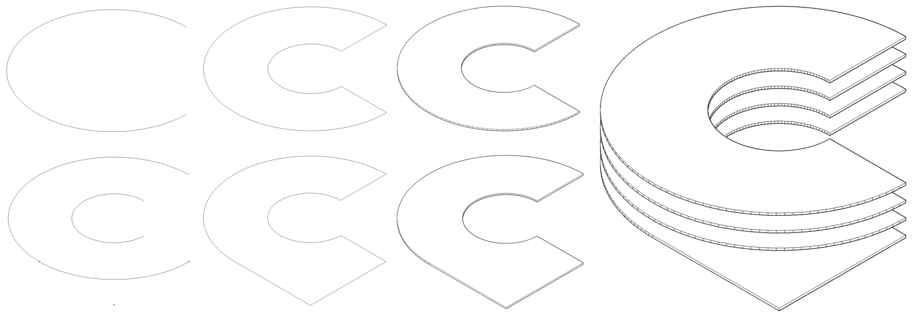


Fig. 6. Interactive test results from the Isenberg slab function evolution process.

Nevertheless, there are two major setbacks to this intersection of interactive evaluation and storytelling: (1) the resulting verbosity, and (2) the aggravated state problems.

(1) The step-by-step development process often results in an accumulation of localized tests and repeated or scattered program fragments. Refactoring and outlining techniques can help architects rearrange some of the scruffiness to obtain a cleaner and more intelligible program in the end.

(i) *Refactoring* is commonly defined as the process of improving the structure of existing programs without changing their semantics or external behavior [17]. There are several semi-automatic refactoring tools [31] that can be adapted to an AD context. Of particular interest to the case are those that help the programmer join scattered code in summarized functions.

(ii) *Outlining* techniques, i.e., the structuring and identification of what each part of the program is or does, may also serve to hide parts of the program for particular audiences and/or purposes. For instance, an outlining mechanism that identifies the tests the user wishes to activate or deactivate in any run considerably improves the program's performance and allows the same program to serve multiple presentation purposes. This goes farther than the outlining system typical of visual programming languages, such as Grasshopper (which allows readers to deactivate individual nodes on demand), by supporting a structured system of goal-oriented active/deactivated groups of program fragments.

Naturally, by using these mechanisms to reorganize the program, we are partially relinquishing the development history. However, we believe this is a necessary trade-off, since the order in which we create a story may not necessarily correspond to the order in which we wish to tell it to others.

(2) State inconsistency is a direct consequence of the use of interactive evaluation, which gets further exacerbated by the permanence of repeated definitions in the program. Outlining may ease part of the problem by deactivating outdated definitions. To avoid it entirely, we propose reactivity, which is discussed in the following section.

3.3 Reactivity

AD programs are systems of hierarchical relations between parts of the design. The functions that compose the separate parts of the building are interdependent so that the entire ensemble can function as a whole. Consequently, all building elements should morph appropriately when design parameters are changed (Fig. 7).

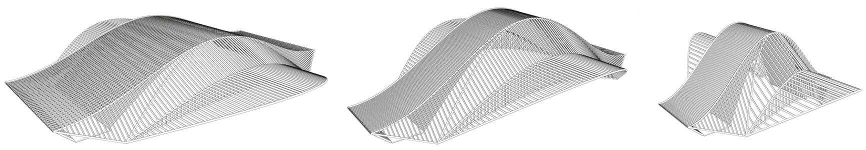


Fig. 7. Liège-Guillemins project: three possible variations of the model with different lengths, widths, and height for the central hangar.

This workflow essentially means that changes to the program frequently influence more parts of that program, other than the one we are modifying. As such, interdependencies must be considered when applying changes. However, in large programs, keeping a mental track of this system of relations is close to impossible. Hence, the program should make these relations visible for designers.

This can be achieved, for instance, by having the entire set of active tests in the program reacting to the changes that are being applied. If we can immediately visualize the impact of the changes all over the program, we are more likely to succeed in maintaining the necessary interdependencies in the AD project. Doing so in an interactive manner, using user-friendly mechanisms, such as buttons or sliders, to explore the influence of design parameters in the project's design space, further adds to the comprehension layer.

Reactivity offers this capacity by keeping track of dependencies in the program. Any change to the program triggers a reevaluation of the dependent parts, ensuring state consistency. Naturally, this means there can be no outdated definitions in the program. Once more, users must resort to outlining and refactoring to deactivate old versions of existing definitions or relinquish storytelling all together for reactivity to work. Furthermore, while reactivity does not necessarily imply the regeneration of the entire model, it does imply the constant re-processing of the dependency graph, which can become computationally intensive in large programs. Liveliness always was and will continue to be a double-edge knife. Hence, it should be used in accordance with the compromises it offers.

4 Exploratory Application

In this section we elaborate on a practical implementation of the proposed concepts in an AD workflow - storytelling, interactive evaluation, and reactivity - using computational notebooks as the base IDE for the experiment.

4.1 Computational Notebooks

Many of the problems described above are shared by the scientific community. In science, reproducibility is critical but the increasing specialization of the different areas is making it hard to reproduce published scientific results. To address this issue, the scientific community is embracing new methods of experimentation that rely more and more in computational simulation and analysis of different phenomena.

Computational notebooks have emerged in this context, promoting method transparency and data availability in the form of "executable papers" [26]. By supporting the description of computational experiments and the analysis of simulated or experimental results in an explanatory and reproducible way [44], they became a critical tool in science [24,37], not only to understand the scientific breakthroughs being presented, but also to reenact them. The following paragraphs describe computational notebooks' interpretation of the dyad concepts: (1) documentation and (2) liveliness.

(1) Computational notebooks were designed to support computational narratives, allowing users to simultaneously execute, document, and communicate their experiments through the intertwining of code and textual and visual documentation. The same notebook can serve multiple purposes, such as tutorial, interactive manual, presentation, or even scientific publication [26,38].

(2) These tools also promote interactive evaluation, a workflow that greatly benefits experimentation with data [38]. This is typically achieved with the input and output cell system: users write a fragment of code in a cell and run it immediately after to observe the result. The cell layout provides a half-solution to the heavy computation agenda of liveliness, since the code lodged in each cell is only run at the user's request.

However, interactive evaluation in notebooks motivates exploratory programing and, in this paradigm, the hunger for immediate results frequently overpowers the interest in organizing the program. Developers can add cells in a non-linear order, definitions can be intertwined with tests in the same code cells, and repeated code bits can be spread along the document. This disarray creates a complex dependency net with hidden program state, whose results frequently confuse the developer. This fact has been identified as one of the major pain points in the use of computational notebooks [10,19] and reactive notebook solutions have also been put forward to respond to the criticism [35,39].

Given the above-mentioned benefits and burdens of the use of computational notebooks as comprehensive IDEs, we propose to explore their use in the context of AD to evaluate storytelling, interactive evaluation, and reactivity. To that end, the two case studies presented above were developed in two different notebooks: the Isenberg School model in Jupyter [41], and the Liège-Guillemins station model in Pluto [39], both using the Julia programing language and the Khepri AD tool [45]. Both projects relied on notebooks' natural tendency for documentation and liveliness, yet the implementation of the three sub-concepts had to be adapted to each notebook, as they operate very differently.

Jupyter [41] is an open-source and web-based notebook. Although it was originally developed to support the programming languages Julia, Python, and R [38], nowadays, Jupyter not only offers a wide range of programming languages but it also allows users to mix them in the same notebook. Jupyter is based on the input/output cell approach and it supports repeated definitions, which allows us to easily apply the storytelling strategy.

Pluto [39] is a computational notebook conceived specifically for the Julia programing language, with one outstanding difference from typical notebooks: reactivity. Pluto was inspired by Observable [35], a reactive computational notebook for JavaScript. Both recognize dependencies between cells, so that when one of them is changed, all dependent ones are automatically updated. This means there can be no repeated definitions, which renders storytelling difficult without outlining mechanisms.

4.2 Storytelling

Storytelling defends program documentation as a way to embrace the tale of the program's evolution. When organized chronologically, the artifacts we produce to document our programs can tell the narrative of the design development process for a better comprehension of the final design solution. Figure 8 presents the modeling history of the Liege-Germmins' project.

To tell the tale of design development, instead of building on existing definitions, storytelling states that developers should consider keeping the definition history intact. In the Jupyter notebook, used to develop the Isenberg project, consecutive versions of the same functions were kept in the document in chronological order. For instance, the project is composed of C-shaped slabs, as shown in Fig. 6. All versions of the above-mentioned `slab` function were kept in the notebook, properly documented, textually and visually.

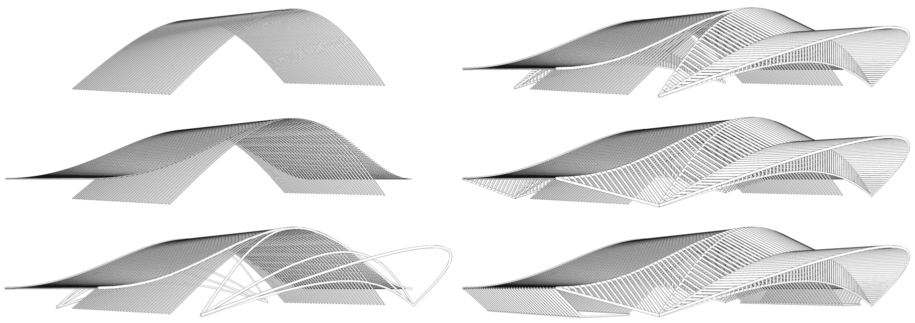


Fig. 8. Liege-Germmins' project development history: modeling sequence.

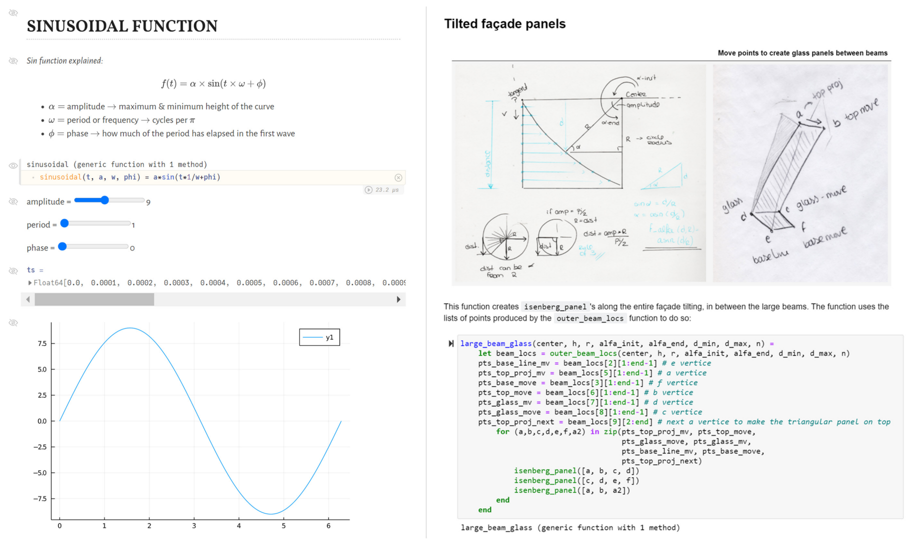
In Pluto, the same effect can only be obtained if we give each version of the function a new name, or if we deactivate old versions (by commenting them or using other outlining techniques). Being a reactive notebook, Pluto needs to maintain state consistency. Hence, it does not allow for repeated definitions.

Regarding the integration of artifacts in the program, computational notebooks have several mechanisms available. For textual documentation, we can use markdown, which can help both explain and structure the program (Fig. 9 left), and simple code comments to explain things inside function definitions (Fig. 9 right). Markdown also supports mathematical notation, which is one of the best ways to describe parametric shapes. Figure 9 presents, on the left, part of the Pluto notebook, where the `sinusoidal` function parameters are explained. This function defines the shape of the station's roof and arches.

Visual documentation can be added via HTML or through IPython's display package for Jupyter and PlutoUI for Pluto. This type of documentation can consist of drawings created during design development that reveal the author's intention towards the program's expected behavior and results (Fig. 9 right) or images generated for the sole purpose of explaining parts of the program. The latter case includes snapshots and rendered images of the generated model. These images are useful devices to make sure the program is producing the results it should. Figure 10 presents some of the snapshots saved in the Jupyter notebook after running test cells on the Isenberg program that generate isolated elements or the building as a whole.

4.3 Interactive Evaluation

Computational notebooks natively promote interactive evaluation as a form of liveliness controlled by the user. Immediate visual results can be obtained by running cells that produce geometry whenever the user wishes to test a particular code snippet. Ideally, this occurs after every new definition. Figure 11 presents an application of the interactive evaluation workflow to the the train station in Pluto: (1) the programmer (re)defines the `canopie_bars` functions, (2) implements a test case, (3) runs the test, and (4) saves the visual result as


SINUSOIDAL FUNCTION

Sin function explained:

$$f(t) = \alpha \times \sin(t \times \omega + \phi)$$

- α = amplitude $\rightarrow$ maximum & minimum height of the curve
- ω = period or frequency $\rightarrow$ cycles per π
- ϕ = phase $\rightarrow$ how much of the period has elapsed in the first wave

```
sinusoidal (generic function with 1 method)
sinusoidal(t, a, w, phi) = a*sin(t*1/w+phi)
```

amplitude = 9

period = 1

phase = 0

```
ts =
Float64[0.0, 0.0001, 0.0002, 0.0003, 0.0004, 0.0005, 0.0006, 0.0007, 0.0008, 0.000
```

Tilted façade panels

This function creates `isenberg_panel`'s along the entire façade tilting, in between the large beams. The function uses the lists of points produced by the `outer_beam_locs` function to do so:

```
large_beam_glass(center, h, r, alfa_init, alfa_end, d_min, d_max, n) =
    let beam_locs = outer_beam_locs(center, h, r, alfa_init, alfa_end, d_min, d_max, n)
    pts_base_line_mv = beam_locs[2][1:end-1] # e vertice
    pts_top_proj_mv = beam_locs[5][1:end-1] # a vertice
    pts_base_move = beam_locs[3][1:end-1] # f vertice
    pts_top_move = beam_locs[6][1:end-1] # b vertice
    pts_glass_mv = beam_locs[7][1:end-1] # d vertice
    pts_glass_move = beam_locs[8][1:end-1] # c vertice
    pts_top_proj_next = beam_locs[9][2:end] # next a vertice to make the triangular panel on top
        for (a,b,c,d,e,f,a2) in zip(pts_top_proj_mv, pts_top_move,
                                    pts_glass_move, pts_glass_mv,
                                    pts_base_line_mv, pts_base_move,
                                    pts_top_proj_next)
            isenberg_panel([a, b, c, d])
            isenberg_panel([c, d, e, f])
            isenberg_panel([a, b, a2])
        end
    end
large_beam_glass (generic function with 1 method)
```

Fig. 9. Pluto, on the left, exploring and explaining the sinusoidal function. Jupyter, on the right, illustrating Isenberg's tilted façade panels' function.

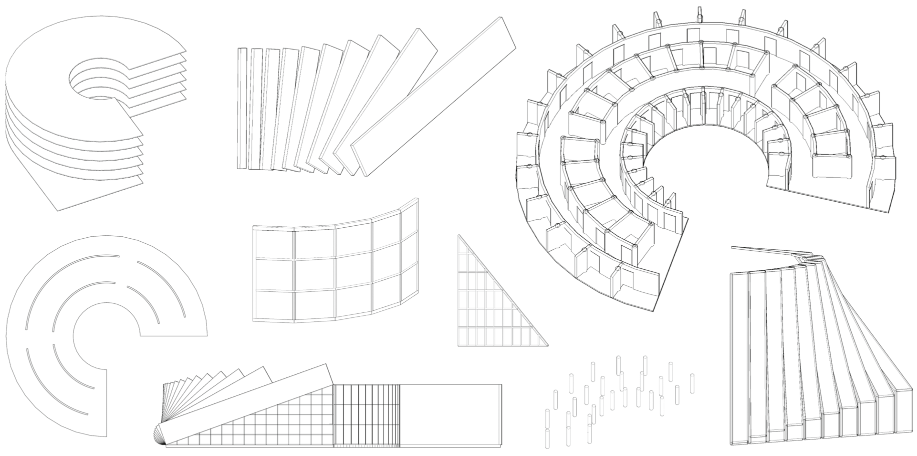
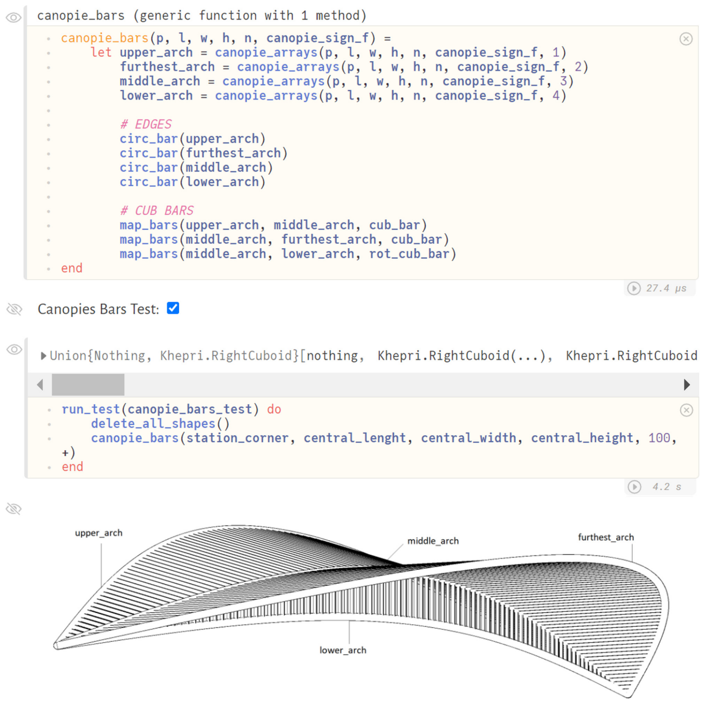


Fig. 10. Snapshots of function tests run from the Jupyter notebook, saved as documentation in the Isenberg program.

documentation in the program. Steps 1 and 3 may be retaken several times until a satisfactory result is achieved. Only then, should the user move on to step 4.



```
canopie_bars (generic function with 1 method)
```

```julia
canopie_bars(p, l, w, h, n, canopie_sign_f) =
    let upper_arch = canopie_arrays(p, l, w, h, n, canopie_sign_f, 1)
        furthest_arch = canopie_arrays(p, l, w, h, n, canopie_sign_f, 2)
        middle_arch = canopie_arrays(p, l, w, h, n, canopie_sign_f, 3)
        lower_arch = canopie_arrays(p, l, w, h, n, canopie_sign_f, 4)

        # EDGES
        circ_bar(upper_arch)
        circ_bar(furthest_arch)
        circ_bar(middle_arch)
        circ_bar(lower_arch)

        # CUB BARS
        map_bars(upper_arch, middle_arch, cub_bar)
        map_bars(middle_arch, furthest_arch, cub_bar)
        map_bars(middle_arch, lower_arch, rot_cub_bar)
end
```

27.4 μs

Canopies Bars Test: ✓

```
▸Union{Nothing, Khepri.RightCuboid}[nothing, Khepri.RightCuboid(...), Khepri.RightCuboid
```

```julia
run_test(canopie_bars_test) do
    delete_all_shapes()
    canopie_bars(station_corner, central_lenght, central_width, central_height, 100,
+)
end
```

4.2 s

Fig. 11. Pluto notebook: interactive evaluation applied to the canopie bars' function in the Liege-Germmins' project.

In this case, the user improved the resulting snapshot by identifying each bar in the image with the corresponding name in the function definition. The drawing thus became a more relevant piece of documentation, not only because it shows what the expected result of the test is, but also because it visually explains the body of the function it illustrates.

Both Jupyter and Pluto provide sets of commands to move cells up and down, and merge and split cells. These can be used for refactoring purposes. As for outlining, we developed two different mechanisms of identifying test cells in the notebooks. In Jupyter, since cells only run at user request, we simply wrapped all test examples with an outlining mechanisms that bound them to a global variable. The state of this variable defines what happens when a test cell is run: it either produces results, or the test is ignored. For Pluto, separate variables manage the state of each test, else the dependency graph would have them all running simultaneously. These variables are presented using PlutoUI's checkbox widget (Fig. 11). Checking a test box will prompt a test cell to run. While the box is checked, the test cell will re-run if any change in the program affects the code within it.

4.4 Reactivity

Reactivity means keeping track of dependencies in the program, so that immediate feedback on results can be provided whenever a change occurs. This is essential for users to understand program dependencies and guarantee consistency in exploratory programming. In the context of AD, reactivity is particularly useful for parametric manipulation (Fig. 12).

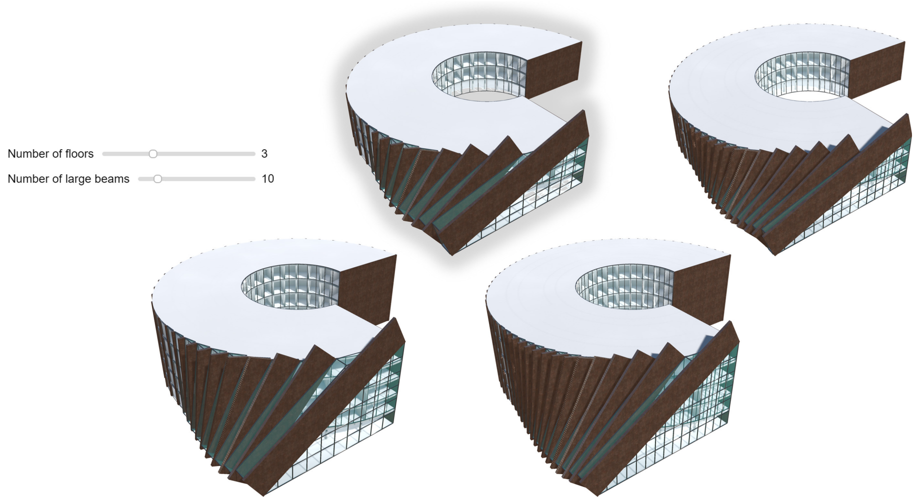


Fig. 12. Isenberg model test with sliders binding the building's parameters. Four variations: 3 to 5 floors variation vertically, and 10 to 15 tilted beams variation horizontally.

Our implementation relies on two extensions of the chosen computational notebooks: Interact for Jupyter and PlutoUI for Pluto, which allow the use of sliders, toggles, and other widgets to visually manipulate data. However, the core difference in the nature of the two notebooks means reactivity is used very differently between them.

In an innately reactive environment like Pluto, widgets bound to global variables provoke immediate updates on dependent cells. This means that in the example shown in Fig. 9 (left), when the sliders bound to the `sinusoid` parameters are changed, the graph shown beneath is immediately updated, as will be other cells in the notebook that are dependent upon these variables. If they are not specifically signaled as ignorable, they will render new results at any change in the sliders. When one dependent cell causes side-effects, such as the creation of architectural objects that is typical of AD, the cell's reevaluation needs to undo the previous side-effects to avoid accumulating them.

To best adapt reactivity to AD, we modified the Pluto notebook to track down not just cell dependencies but also their side-effects. As such, the notebook's dependency graph knows which geometry was affected by each change, so that it can be selectively deleted and regenerated, maintaining state consistency with good interactive performance.

In Jupyter, we cannot hope to achieve reactivity at the level of the notebook. Since cells are only evaluated on user demand, we may strive for partial reactivity. Interactive features, in this case, are best used for particular tests inside one cell only. Figure 12 presents an example: the widgets are bound to local variables in a test cell used to generate the Isenberg model. Any change to the sliders will prompt the regeneration of the model in that test cell.

5 Conclusion

In this paper, we explored the program comprehension dyad - documentation and liveliness - as a means to improve the comprehension of AD programs. We proposed the integration of more adequate methods of developing and maintaining algorithmic representations of complex architectural projects by fine-tuning the two concepts to better suit the architectural scenario. Three ideas emerged: (1) storytelling - explaining the program through human-readable text and meaningful imagery, intertwined with the code itself to tell the narrative of design development; (2) interactive evaluation - a scalable way of incorporating liveliness in programming, providing feedback on program results upon user demand; and (3) reactivity - a systematic approach to liveliness, which offers immediate feedback on program changes and reveals program dependencies. We proposed to incorporate the three concepts in an AD workflow and we evaluated their application using computational notebooks as the base programming environment.

The three ideas intersect in many ways, starting with interactive evaluation and reactivity being two different ways of applying liveliness to the programming task. Interactive evaluation offers a more scalable option, albeit with program state issues, whereas reactivity resolves any state problems and promotes a more interactive workflow, however, failing to scale to large projects without proper outlining mechanisms. A trade-off is needed between efficiency and consistency, but reactivity in itself will always suffer from scalability issues.

Both interactive evaluation and reactivity promote exploratory programming, which can cause program disarray. Storytelling, on the other hand, defends an organized approach to the programming endeavor, which means refactoring mechanisms must be used for the three approaches to co-exist.

Storytelling also appeals to the preservation of the development history as part of the narrative. This directly conflicts with reactivity's requirements for a consistent program state. The two can only operate simultaneously if outdated definitions are disabled. Interactive evaluation, in turn, greatly contributes to storytelling by producing artifacts that can be used as documentation, namely tests results in the form of imagery.

In sum, the three ideas provide different sets of advantages and trade-offs to the process of developing AD projects. Their integration in the AD workflow requires support from the programming environment according to the architect's own needs at any stage of the development process, as well as considering the ultimate goal for the program. AD programs may be a means of attaining a single constructive goal, they may represent a set of ideas to be re-used in the future, they may serve as presentation mechanisms, etc. Taking this in mind, some of the concepts may or may not apply to each case.

As future work we plan on investigating solutions for the ever-lasting compromise between liveliness and scale. Given that architects do not need to visualize the entirety of the project all the time, nor the entirety of the project's detail, we expect Levels Of Detail (LODs) and/or selective generation based on project areas or view cones to yield good results. This may allow architects to keep using reactive mechanisms later along the project's development. We also plan on incorporating traceability research to the proposed methodology, studying mechanisms of integrating it with the remaining concepts in a functional workflow.

Acknowledgments. This work was supported by national funds through *Fundação para a Ciência e a Tecnologia* (FCT) (references UIDB/50021/2020, PTDC/ART-DAQ/31061/2017) and a PhD grants under contract of FCT (DFA/BD/4682/2020).

References

1. Alfaiate, P., Caetano, I., Leitão, A.: Luna moth: supporting creativity in the cloud. In: 37th ACADIA Conference, pp. 72–81. Cambridge, Massachusetts, USA (2017)
2. Baecker, R., Price, B.: The early history of software visualization. In: Stasko, J., Domingue, J., Brown, M.H., Price, B.A. (eds.) Software Visualization: Programming as a Multimedia Experience, chap. 2, pp. 29–34. MIT Press (1998)
3. Bainomugisha, E., Carreton, A.L., Van Cutsem, T., Mostinckx, S., De Meuter, W.: A survey on reactive programming. ACM Comput. Surv. **45**(4), 34 (2013). https://doi.org/10.1145/2501654.2501666
4. Ball, T., Eick, S.G.: Software visualization in the large. Computer **29**(4), 33–43 (1996). https://doi.org/10.1109/2.488299
5. Bass, L., Kazman, R., Clements, P.: Software Architecture in Practice. Pearson Education, New Jersey, United States (2012)
6. Brown, M.H.: Zeus: A system for algorithm animation and multi-view editing, SRC reports, vol. 75. Digital, Systems Research Center (SRC), Palo Alto, California (1992)
7. Burnett, M.M.: Visual programming. In: Webster, J.G. (ed.) Wiley Encyclopedia of Electrical and Electronics Engineering, pp. 275–283. John Wiley & Sons, Inc. (1999). https://doi.org/10.1002/047134608X.W1707
8. Burry, M.: Scripting Cultures: Architectural Design and Programming. John Wiley & Sons, Inc., Architectural Design Primer (2011)
9. Celani, G., Vaz, C.E.V.: CAD scripting and visual programming languages for implementing computational design concepts: a comparison from a pedagogical point of view. Int. J. Architectural Comput. **10**(1), 121–137 (2012). https://doi.org/10.1260/1478-0771.10.1.121
10. Chattopadhyay, S., Prasad, I., Henley, A.Z., Sarma, A., Barik, T.: What's wrong with computational notebooks? Pain points, needs, and design opportunities. In: CHI Conference on Human Factors in Computing Systems, pp. 1–12. ACM, Honolulu, HI, USA (2020). https://doi.org/10.1145/3313831.3376729

11. Davis, D., Burry, J., Burry, M.: Understanding visual scripts: improving collaboration through modular programming. Int. J. Architect. Comput. **9**(4), 361–375 (2011). https://doi.org/10.1260/1478-0771.9.4.361
12. Diehl, S.: Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software. Springer, Berlin Heidelberg (2007). https://doi.org/10.1007/978-3-540-46505-8
13. Do, E.Y.L., Gross, M.D.: Thinking with diagrams in architectural design. Artif. Intell. Rev. **15**(1–2), 135–149 (2001). https://doi.org/10.1023/A:1006661524497
14. Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S.: How to Design Programs: An Introduction to Programming and Computing. The MIT Press, Cambridge (2001). https://doi.org/10.4230/LIPIcs.SNAPL.2015.113
15. Findler, R.B., Flanagan, C., Flatt, M., Krishnamurthi, S., Felleisen, M.: DrScheme: a pedagogic programming environment for scheme. In: Glaser, H., Hartel, P., Kuchen, H. (eds.) PLILP 1997. LNCS, vol. 1292, pp. 369–388. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0033856
16. Forward, A., Lethbridge, T.C.: The relevance of software documentation, tools and technologies: a survey. In: Symposium on Document Engineering, pp. 26–33. ACM, McLean, Virginia, USA (2002). https://doi.org/10.1145/585058.585065
17. Fowler, M.: Refactoring: Improving the Design of Existing Code. Object Technology Series. Addison-Wesley, Boston (1999)
18. Goldstine, H.H., Von Neumann, J.: Planning and coding of problems for an electronic computing instrument: Report on the Mathematical and Logical aspects of an Electronic Computing Instrument. Institute for Advanced Study Princeton, New Jersey (1947)
19. Grus, J.: I don't like notebooks (2018). https://www.youtube.com/watch?v=7jiPeIFXb6U&ab_channel=O%27Reilly. Accessed 21 Jan 2021
20. Hensel, M.: Performance-Oriented Architecture: Rethinking Architectural Design and the Built Environment. John Wiley & Sons, Inc., Architectural Design Primer (2013)
21. Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L.: Summarizing source code using a neural attention model. In: 54th Annual Meeting of the Association for Computational Linguistics, vol. 1, pp. 2073–2083. ACL, Berlin, Germany (2016). https://doi.org/10.18653/v1/P16-1195
22. Jackson, M.A.: Principles of Program Design. Academic Press Inc, USA (1975)
23. Kery, M.B., Myers, B.: Exploring exploratory programming. In: Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pp. 25–29 (2017). https://doi.org/10.1109/VLHCC.2017.8103446
24. Kery, M.B., Radensky, M., Arya, M., John, B.E., Myers, B.A.: The story in the notebook: exploratory data science using a literate programming tool. In: CHI Conference on Human Factors in Computing Systems, pp. 1–11. ACM, Montreal QC, Canada (2018). https://doi.org/10.1145/3173574.3173748
25. Knuth, D.E.: Literate programming. Comput. J. **27**(2), 97–111 (1984). https://doi.org/10.1093/comjnl/27.2.97
26. Lasser, J.: Creating an executable paper is a journey through open science. Commun. Phys. **3**(143), 1–5 (2020). https://doi.org/10.1038/s42005-020-00403-4
27. Lee, E.A., Messerschmitt, D.G.: Synchronous data flow. Proceedings of the IEEE **75**(9), 1235–1245 (1987). https://doi.org/10.1109/PROC.1987.13876
28. Leitão, A., Lopes, J., Santos, L.: Programming languages for generative design: a comparative study. Int. J. Architect. Comput. **10**(1), 139–162 (2012). https://doi.org/10.1260/1478-0771.10.1.139

29. Leitão, A., Lopes, J., Santos, L.: Illustrated programming. In: 34th ACADIA Conference, pp. 291–300. Los Angeles, California, USA (2014)
30. Loukissas, Y.: Keepers of the geometry. In: Turkle, S. (ed.) Simulation and Its Discontents, pp. 153–170. MIT Press, Cambridge (2009). https://doi.org/10.7551/mitpress/8200.003.0014
31. Mens, T., Tourwe, T.: A survey of software refactoring. IEEE Trans. Softw. Eng. (TSE) **30**(2), 126–139 (2004). IEEE. https://doi.org/10.1109/TSE.2004.1265817
32. Myers, B.: Taxonomies of visual programming and program visualization. J. Vis. Lang. Comput. **1**(1), 97–123 (1990). https://doi.org/10.1016/S1045-926X(05)80036-9
33. Nassi, I., Shneiderman, B.: Flowchart techniques for structured programming. SIGPLAN Not. **8**(8), 12–26 (1973). https://doi.org/10.1145/953349.953350
34. Nguyen, A.T., Reiter, S., Rigo, P.: A review on simulation-based optimization methods applied to building performance analysis. Appl. Energy **113**, 1043–1058 (2014). https://doi.org/10.1016/j.apenergy.2013.08.061
35. Observable Inc: Observable: Make sense of the world with data, together (2021). https://observablehq.com. Accessed 21 Jan 2021
36. Oda, Y., et al.: Learning to generate pseudo-code from source code using statistical machine translation. In: 30th International Conference on Automated Software Engineering (ASE), pp. 574–584. IEEE/ACM, Lincoln, NE, US (2015). https://doi.org/10.1109/ASE.2015.36
37. Perez, F., Granger, B.E.: IPython: a system for interactive scientific computing. Comput. Sci. Eng. **9**(3), 21–29 (2007). https://doi.org/10.1109/MCSE.2007.53
38. Perkel, J.M.: Why Jupyter is data scientists' computational notebook of choice. Nature **563**(7729), 145–146 (2018). https://doi.org/10.1038/d41586-018-07196-1
39. van der Plas, F., Bochenski, M.: Pluto.jl (2021). https://github.com/fonsp/Pluto.jl. Accessed 21 Jan 2021
40. Price, B., Baecker, R., Small, I.: An introduction to software visualization. In: Stasko, J., Domingue, J., Brown, M.H., Price, B.A. (eds.) Software Visualization: Programming as a Multimedia Experience, chap. 1, pp. 3–28. MIT Press (1998)
41. Project Jupyter: Jupyter (2021). https://jupyter.org. Accessed 21 Jan 2021
42. Rein, P., Ramson, S., Lincke, J., Hirschfeld, R., Pape, T.: Exploratory and live, programming and coding: a literature study comparing perspectives on liveness. Programm. J. **3**(1), 1:1–1:33 (2018). https://doi.org/10.22152/programming-journal.org/2019/3/1
43. Roman, G.C., Cox, K.C.: A taxonomy of program visualization systems. Computer **26**(12), 11–24 (1993). https://doi.org/10.1109/2.247643
44. Rule, A., Tabard, A., Hollan, J.D.: Exploration and explanation in computational notebooks. In: CHI Conference on Human Factors in Computing Systems, pp. 1–12. ACM (2018). https://doi.org/10.1145/3173574.3173606
45. Sammer, M.J., Leitão, A., Caetano, I.: From visual input to visual output in textual programming. In: 24th CAADRIA Conference, vol. 1, pp. 645–654. Wellington, New Zealand (2019)
46. Scheer, D.R.: The Death of Drawing: Architecture in the Age of Simulation. Taylor & Francis, Milton Park (2014). https://doi.org/10.4324/9781315813950
47. Sorensen, A., Gardner, H.: Programming with time: cyber-physical programming with impromptu. SIGPLAN Not. **45**(10), 822–834 (2010). https://doi.org/10.1145/1932682.1869526

48. Storey, M.A.: Theories, methods and tools in program comprehension: past, present and future. In: 13th International Workshop on Program Comprehension (IWPC), pp. 181–191. IEEE Computer Society, Washington, DC, USA (2005). https://doi.org/10.1109/WPC.2005.38
49. Victor, B.: Inventing on principle (2012). https://vimeo.com/38272912. Accessed 21 Jan 2021
50. Victor, B.: Stop drawing dead fish (2012). https://vimeo.com/64895205. Accessed 21 Jan 2021
51. Woodbury, R.: Elements of Parametric Design. Routledge, Milton Park (2010)