

# An Enhanced Programming Environment for Generative Design

Guilherme Ferreira  
 Instituto Superior Técnico  
 Universidade de Lisboa  
 guilherme.ferreira@tecnico.ulisboa.pt

**Abstract**—In the area of generative design (GD), programs are becoming increasingly complex and harder to understand, communicate, and share, enlarging the gap between them and the architectural concepts they implements. To overcome this problem, we need to develop documentation techniques and program comprehension tools to the Generative Design domain. This thesis proposes two programming tools for GD systems, namely (1) sketch-program correlation tool that allows architects to use sketches and combining them with code, and (2) immediate feedback tool that accelerates the effect of a code change and its visualization. We found that while the first tool turns program documentation in a less tiresome task, consequently minimizing the lack of documentation in the GD programs; the second tool enhances the program visualization, creating a new medium that helps people to design programs.

**Index Terms**—Immediate feedback; Image correlation; Dr-Racket; Rosetta

## I. INTRODUCTION

Challenges in understanding programs are all too familiar since the early days of computing. At that time we wrote programs in *absolute binary* [1]. It is a numeric representation typically expressed by a sequence of zeros and ones, meaning that the programs were represented as a sequence of instructions and addresses both written in binary. To understand a program in this form is almost impossible.

Since early days, as Fred Brooks pointed out in his influential essay [2], we have come to accept that *there is no silver bullet* to understand a program. Fortunately, in recent years, the field of *program comprehension* [3] has evolved considerably, because a program that is not comprehended cannot be changed, shared and communicated.

The area of program comprehension has shown that to understand a program, a *silver bullet* may not be required. This field came up with several theories that provide rich explanations of how people understand programs. For instance, the *top-down* theory [4] says that to comprehend a program, the programmer must create a mental model of the program's structure and behavior. This model is a set of hypothesis which the programmer confirms or rejects based on evidence found in the code.

In response to these theories or in parallel with them, many environments and innovative tools were created or updated. Some of the examples are: sophisticated frameworks to support rapid construction and integration of tools [5], advanced programming environments with intelligent user

interface [6], [7], [8], [9], and simple tools designed for learning environments [10], [11], [12], [13], [14], [15]. In parallel to these advances, there are other fields interested in program comprehension. For example, in the Architecture field, new tools [16], [17] are being proposed to support *generative design*: a procedural method for generating architectural models [18], that also suffers from program understanding problems.

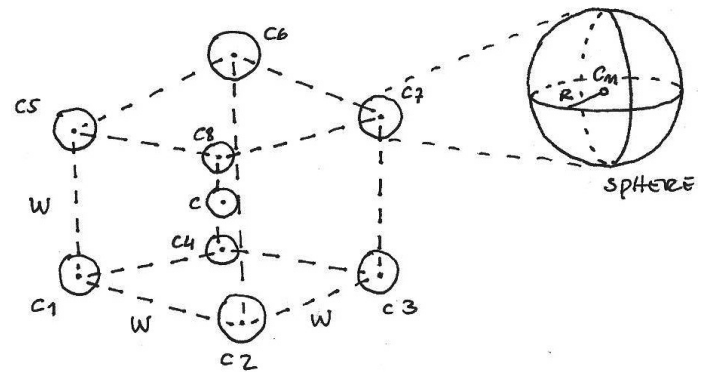


Fig. 1. An architectural sketch.

Regardless the area, people follow two basic steps to build a program: first imagining its details, then implementing them. This is a natural process for programmers that is commonly performed in their heads. Architects, by contrast, prefer another medium to express their ideas: diagrams/sketches [19], because it is a compact medium to convey complex ideas. For example, Figure 1 shows a sketch of a geometric model which would be more complex, if it were described in text. These drawings are also helpful in the end of design conception, because they clearly document the design decisions, the relationship between different parts of the design, and the impact of external factors in the final shape.

The generative design programs, by definition, can itself be considered a description of a design, as it formally specifies the modeling process of the design. However, this formal specification can only be easily understood for simple design problems. Consequently, the situation becomes the same of any sufficiently complex program, then it is helpful to have *program documentation*.

Many problems related with program comprehension could be mitigated, if the programs were properly documented. Source code comments is the most important artifact to

understand a system and to maintain, as showed in [20]. Unfortunately, writing documentation is perceived as a tiresome task and, thus, is frequently avoided [21], which negatively affects software development. A result of the lack of program documentation is that programmers must spend a significant amount of time separating relevant ideas from the irrelevant ones.

We think that, by creating well designed tools, it is possible to improve *program comprehension* and *program documentation*. We plan to address this problem in two ways: (1) minimizing the lack of documentation in the programs, by turning program documentation in a less tiresome task, and (2) creating a new medium to help people design programs, by anticipating the effect of their actions in the program output.

## II. OBJECTIVES

This thesis addresses two challenges:

- minimize the lack of documentation in GD programs.
- improve the program comprehension by facilitating program experimentation.

To overcome these challenges, we investigated the flow which people follow to design programs in GD, analyzing better techniques to help programmers at each conceptual task of this process. The goal is to develop and implement innovative tools which support and encourage new ways of thinking, and therefore, enabling programmers to see more easily and understand their programs.

Our approach to achieve this objective was, at first, analyze how programming tools can improve program comprehension. The *Learnable Programming* [22], [23] approach has shown interesting insights in this direction. Secondly, based on this analysis, and in order to prove our ideas, we implemented two interactive tools tailored for generative design programs:

- 1) *Sketch-program correlation tool*, which encourages architects and designers to reuse their conceptual sketches, such as that shown in Figure 1, to visually document their programs. These sketches are correlated with the program source code in a way that significantly reduces the effort to read the code. Therefore, it allows users to acquire a better mental model.
- 2) *Immediate feedback tool*, which gives a new medium for architects and designers to create new ideas by continuously reacting with changes in their models. This tool minimizes the latency between writing the code and executing it, consequently this encourages users to experiment ideas quickly, augmenting their comprehension about the program.

This thesis produced the following expected results: *i)* a specification of each tool, its purpose and how this tool is designed to support its purpose, *ii)* an implementation of a prototype, and *iii)* an experimental evaluation with a comparison to other similar tools.

## III. RELATED WORK

A *programming system* has two fundamental parts: the *programming language* that users should know to create a program, and the *programming environment* that is used to

write and test programs. Undoubtedly, both parts are equally important to build a program and understand it. However the boundaries between these components can be blurred, for this reason, the related work goes beyond programming environments including also programming languages and their design aspects.

### A. Processing

Processing [12] is a programming language and environment designed to teach programming in an optic context. Processing has become popular among students, artists, designers, and architects because it acts as a tool to get non-programmers started with programming through instant visual feedback.

The Processing programming language is built on top of Java, but it removes much of the verbosity of Java to make the syntax accessible to novices. The language provides simple access to external libraries, such as OpenGL, through single entry points, such as `setup` and `draw`. Therefore, this allows novices to quickly prototype, learn fundamental concepts of programming, and, eventually, gain the basis to learn other programming languages.

The programming environment contains a simple text editor, a text console to present errors, and a run button. The run button compiles the Processing code and executes it. Moreover, in the default mode, the program result is displayed in a 2D graphical window, the render can be configured to present the result in 3D, or other sophisticated methods using *shaders* to recur directly to the graphic board.

Actually, with a few changes, the Processing code can be exported as an application for different platforms, such as Java, JavaScript, and Android. For example, to export a Processing program for JavaScript, it is only necessary to create a HTML page and include the Processing code as a script of this page. Then the Processing code will be automatically parsed and translated to JavaScript. To maintain the usual render capabilities of a Processing program, it will use the HTML5 canvas with WebGL.

The popularity of Processing is explained by the benefits of these features, besides of being a domain-specific language. However, it has drawbacks that can discourage its use, such as the following:

- *Weak metaphor*. The Processing programming language, by contrast with the above systems, has none strong metaphors that allow the programmer to translate his experiences as a person into programming knowledge.
- *Poor decomposition*. Processing discourages the elementary approach to solving a complex problem by breaking it into simpler problems because drawing and input events link to single entry points. Thereby the behavior of sub-modules must be tangled across these global functions, making it difficult to achieve clean decomposition.
- *Poor recomposition*. Processing discourages combining two programs. The designer cannot just grab and use part of other programs because variables must be renamed or manually encapsulated, and the `draw` and mouse functions must be woven together. Even worse, Processing has global modes which alter the meaning of the function

arguments. For example, two Processing programs can specify its colors in different modes and each mode has its proper purpose of `fill` function arguments. Combining those programs will be almost impossible.

- *Weak readability.* The syntax of a Processing program represents a significant barrier for reading. For example, the function which draws an ellipse on screen is written as `ellipse(50,50,100,100)`. The reader must look up or memorize the meaning of every single argument.
- *Fragile environment.* The programming environment is fragile because it does not attempt to solve any of the above issues related to the language and its implementation.

### B. DesignScript

DesignScript [16] is a programming language and environment designed to support GD with textual methods. It is mainly used by architects and designers to generate geometric models using a script. Then when the script executes it creates new models in a computer-aided design (CAD) tool. DesignScript is an Autodesk<sup>1</sup> product initially proposed to be used within AutoCAD, nowadays it provides the same functionality on top of Revit, another Autodesk product used for building information modeling (BIM). In short, a BIM model is similar to a CAD model, but it covers more than just geometry. It also includes spatial relationships, properties of building components, such as manufacturers' details.

The programming language is an associative paradigm. The variables are abstract types that can represent numeric values or geometric entities. These variables are in a graph of dependencies. When a change in a variable occurs it forces the re-evaluation of the graph, as shown in Figure 2, consequently variables has always updated values. This feature is useful, especially in a modeling environment, because it provides continuous feedback to the designer as the model is modified.

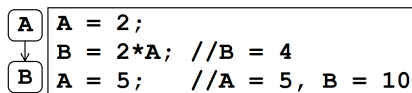


Fig. 2. Associative interpretation

The DesignScript's programming environment provides a text editor, an interpreter, and a simple debugger. The language interpreter executes each time that the designer clicks on the run button. Then all the script is interpreted, and its result produces geometric entities rendered in the CAD. The continuous feedback feature works only in debug mode because, in this way, the script is interpreted line by line. Thus, each update to a variable will change its dependencies and will recompute the model. However, in debug mode the code cannot be edited, so this feature is worthless during code editing.

In the DesignScript's debug mode, users can inspect the variable values by adding *watchers* to them. A watched variable is showed in a particular tab, as shown in Figure 2. In case the variable represents a geometric model, the own design

will be highlighted in the CAD when the variable is selected. It creates a certain *traceability* between patterns in the CAD and code in the editor. In this way, the user is able to correlate which model a variable corresponds. However the inverse, starting to form the model and finding the correspondent variable, is unsupported.

DesignScript also supports a typical mechanism of *live* programming environments: the sliders. The sliders are widgets which facilitate giving new values to the program input. This way, designers can create new models reacting to these changes. However, in the DesignScript's sliders, the changes are reflected in the models only when the programmer leaves the slider. Until then, the designer should imagine how the model would be with the new value, which is completely against the purpose of sliders.

Moreover the DesignScript language, despite being presented as pedagogic, has some drawbacks. It does not carry any manly metaphor which helps beginners start with the language. Additionally, the associative paradigm represents a barrier for sharing code: it discourages the recomposition of modules because new modules can change the previous one. The environment provides poor mechanisms that help people to find bugs in the code, and finally, DesignScript is confined to produce geometry in a single CAD tool.

### C. Monkey

Monkey<sup>2</sup> is a programming environment designed to support GD. Like DesignScript, Monkey is used to edit, debug and interpreter scripts. However, Monkey uses RhinoScript as its programming language and Rhinoceros3D<sup>3</sup> (or Rhino for short), a lighter CAD than AutoCAD, to generate the geometric models.

Monkey is implemented as a .NET plugin for Rhino4 and provides a programming environment to write and debug scripts. The RhinoScript is based on Microsoft's VBScript language (a descendant of BASIC), and like VBScript, it is a weakly typed language. One of the major drawback with this language is the fact that users must beware with the data passed in their functions at all time because RhinoScript can accidentally cast variables into inappropriate types. Therefore, it creates errors difficult to find, especially for people who are learning to program.

Monkey is based on general-purpose programming environments. It provides typical features of those environments, namely syntax highlighting, auto-completion, and error highlighting. The organization of code into trees is also similar. However, the programming environment and language do not provide any well-designed feature which helps beginners to start with programming. The offered features are based on general-purpose systems, instead of being tailored for GD.

### D. Rosetta

Rosetta [17] is a programming environment designed to support GD that is based on DrRacket [13]. Like Monkey,

<sup>1</sup><http://www.autodesk.com/products>

<sup>2</sup><http://wiki.mcneel.com/developer/monkeyforrhino4>

<sup>3</sup><https://www.rhino3d.com>

Rosetta provides its environment detached from the CAD. Rosetta is a step forward from the previous systems, because it solves the portability problem among CAD tools. In Rosetta a GD program can be written in various programming languages (frontends) and the geometric models can be rendered by different CADs (backends). As a result, designers are free to write their programs in their selected frontend which, upon execution, will generate the same geometry for the various backends.

Rosetta has been used to teach programming in architecture courses. Tailored to this end, Rosetta uses DrRacket as its programming environment. The DrRacket environment serves some functions, but the most important are that the student can start immediately to learn to program. For instance, the environment is set up with just three lines of code.

The Racket language is also an advantage of Rosetta's environment because it encourages the use of the scientific paradigm for writing algorithms. In this way, students that learn simple programming techniques, such as recursion, can create robust models. Additionally, as the students progress, new programming languages are also available to learn, such as JavaScript, Python, Processing, and so on.

The Rosetta's environment provides some interesting tools for GD, such as a programming flow tracer, similar to the DesignScript's watcher. It highlights models in the CAD upon selection of expressions, it also supports the inverse, selecting the mode in the CAD and shows the expression in the code editor. Another interactive tool is the slider, an attempt to provide immediate feedback to the designers. It uses the DrRacket slider, associating the slider callback to the function that generates the entire model, so each time the slider change a new model will be created. However, this process must be performed manually.

Undoubtedly Rosetta's environment goes further than the textual environments for GD presented in this report. However, it presents some drawbacks which may discourage the learning in general. Beginning with the usual programming language: Racket. The syntax of a Racket program represents a significant barrier for reading. For instance the function which draws a circle in Rosetta is written as `(circle (xy 0 0) 1)`. The reader must lookup or memorize every argument. Using the Rosetta's documentation the reader will spend even more time because it is in a book mixed with architecture topics.

### E. Grasshopper

Grasshopper<sup>4</sup> is a programming language and environment designed to support GD using a visual language. Grasshopper provides an alternative way to programming. By definition, it is a bi-dimensional representation consisting of iconic components that can be interactively manipulated by the user according to some spatial grammar [24]. For example, the boxes in Figure 3 are components which receive the input (left ports) perform some operations and return the output (right port). The components link to other elements establishing a *dataflow* paradigm where the input of an element is the output of another.

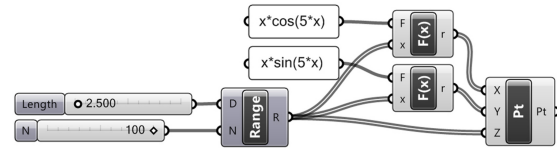


Fig. 3. A program in Grasshopper that computes the 3D coordinates of a conical spiral. Each time the left sliders are dragged a new coordinate is calculated.

Like Monkey, Grasshopper is implemented as a *plugin* for Rhino<sup>3</sup>. However, Grasshopper tailors the Rhino's environment with specific GD tools. These tools are state of the art because they implement important principles for design models, such as the following:

- *Get immediate feedback.* As the user interacts with the components, by adding and connecting them, the result reflects immediately in the CAD model. It facilitates the design conception because the user's intentions are immediately visible.
- *Facilitate program input.* To facilitate the process of design exploration, Grasshopper provides sliders which connect at the component input. Dragging the slider causes a change propagation through components. The components are re-executed with the new slider value. Combined with the above feature new models are generated immediately.
- *Correlate the program with the generated elements.* Like DesignScript's watcher, by selecting a component, its geometry is highlighted in the CAD. It allows designers to understand a program better by figuring out the roles for each element.
- *Show comparisons between models.* Grasshopper provides a special component that, when connected to the output of another component, replicates the geometry. This mechanism is useful for design exploration because it maintains in the CAD's background an old replica of the changed geometry. It adds a context at each change so that the designer can compare the result of his change in the new geometry based on the old one.

Mainly, the Grasshopper interactivity depends on the immediate feedback tool. However, this tool will never scale for arbitrarily complex programs because the CAD's render is not designed to process the massive amount of information generated by GD methods. Other systems, such as DesignScript and Rosetta, improve this problem by sidestepping most of the functionality of traditional CAD tools and focusing only on the generation and visualization of geometric models. These systems provide a backend based on OpenGL that is independent of a full-fledged CAD application, but, in Grasshopper, there is no so backend.

Moreover, the traceability among components is just in one direction. From the designer perspective, it would be more useful start from the geometry and find which component implements it, but it is unsupported. However, despite the usefulness of model comparison in design exploration, this feature is also unsupported.

<sup>4</sup><http://www.grasshopper3d.com/>

## F. Dynamo

Dynamo<sup>5</sup> is a programming language and environment designed to support GD. Like Grasshopper, Dynamo provides an alternative way to programming. However Dynamo, like DesignScript, is implemented on top of Revit, an Autodesk product for BIM.

Dynamo provides a set of tools similar to Grasshopper, particularly a searching table, which provides quick access to the primitives of the language, such as the components and widgets. This feature encourages designers to explore the available parts and try current components.

In general, Dynamo and Grasshopper are programming environments and visual languages popular among novices in programming. The smooth learning curve and perhaps the style of the user interface (UI) elements are attractive for beginners. However as the visual programs become broad and complex, it requires more time to understand, maintain, and adapt to new requirements, than the textual programs as showed in [25]. Despite spending more time and effort to learn a textual programming language, the learners have their time quickly recovered once the complexity of the design task becomes sufficiently large.

In the surveyed systems, the typical representation of code is textual. This representation is typically static and, to be understood, requires the reader to know the vocabulary of the programming language. For a novice, it is simply a barrier to learning. On the other hand, the representation of programs as graphical components or mathematical forms lowers this barrier, because the information is visually more perceptible, but it becomes incomprehensible as the program grows.

## IV. IMPLEMENTING THE PROPOSED TOOLS

The problem addressed in this thesis is to design and implement an interactive programming environment for generative design that covers the needs of GD community. The approach followed, suggests two interactive tools: (1) *program-sketch correlation tool*, which correlates sketches with code, as a result, it significantly reduces the effort to read the code, and (2) *immediate feedback tool*, which executes the program upon changes, thereby creating an interactive environment to users quickly test their ideas and, eventually, improve their program comprehension.

### A. Program-sketch correlation tool

There are two ways to correlate a generative design program with its produced model. The first is using a sketch in the program that illustrates the intended model. The second is using the generated model, correlating the elements of the shape with fragments of the code. The first type of correlation did not exist in Rosetta, consequently this is a contribution of this thesis.

The **program-sketch correlation tool** was implemented using the power of DrRacket syntax check to bind annotated identifiers. In this way, the images resources, already supported

in the text editor, now becomes a new category of rich-media expressions included in the syntax check annotated types. As a result, programmers can add in their functions, images that illustrate the purpose of that function. Therefore, this is perfectly suitable for functions used in GD programs, because usually the output of these functions can be visually represented by a sketch.

For example, Figure 4 shows a typical example where the architect defines a function that creates a cube with spheres in its vertices. In this example, the architect can start by searching for each function argument, moving the mouse over them, to find the meaning of these arguments in code. Inversely, he can start from the image by moving the mouse over it, to find the meaning of these arguments in code. Furthermore, programmers can use images in their programs, and the image inserted in the function body acts as if it was part of the program. However, internally it continues to be a mere code comment that does not affect the correct operation of the function.

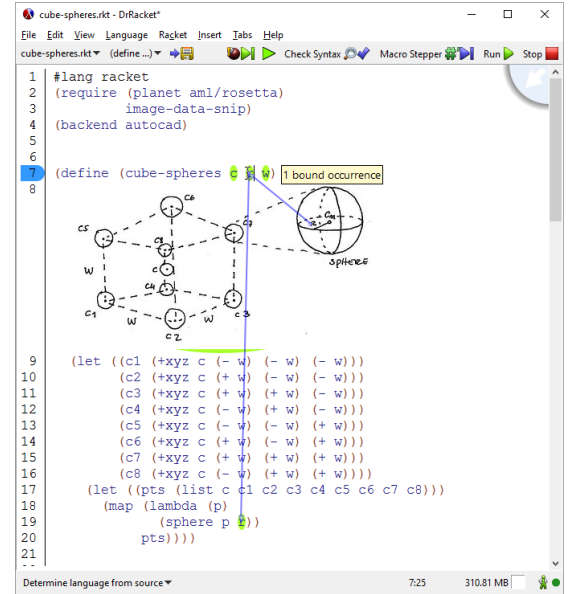


Fig. 4. Relating function arguments with image. The highlighted argument (under the cursor, in green) is illustrated in the image (on the R symbol, using blue arrows).

Although images are a suitable media to represent geometric objects, textual information can also be appropriate in some cases. For example, during the development of the program, usually, several functions are created. Consequently, the architect may not have a sketch to document them. Furthermore, the process to draw a new sketch, import it to the computer, and insert it in the code, is clearly more laborious than writing a simple piece of text that describes the function. Using this method he can later complete his code with a proper sketch, but until then the function is still documented, and he can share his code without any trouble.

Thinking about this situation, I extended the program-sketch correlation tool to support also source code comments. Similarly to the correlation with images, users can insert a string that explains the method. This string can be annotated

<sup>5</sup><http://dynamobim.com/>



with special characters to correlate its characters graphically to the function identifiers. For example, Figure 5 portrays a typical situation where the architect is reading the function comment and moves the mouse over an annotated identifier (`@orthogonal-cones`). Thus, immediately, some arrows are displayed that point to the definition of that function, and also to its use in the program context.

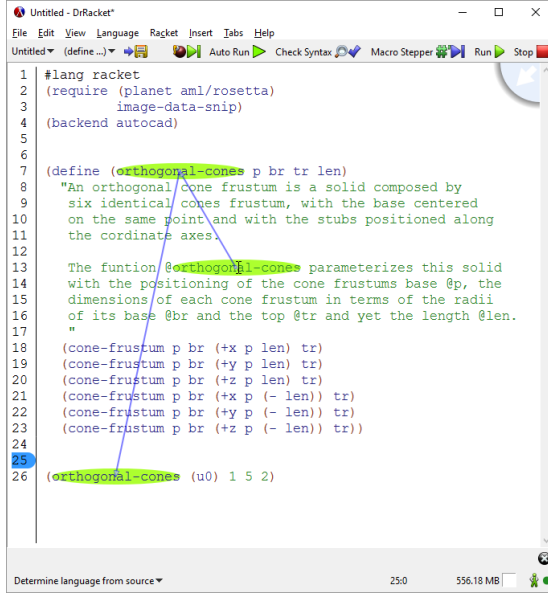


Fig. 5. Relating code comments with function arguments. The highlighted function name (under the cursor, in green) is linked to its definition and utilization in the source code (in green, using blue arrows).

Regarding the usability of this tool, an inherent concern already noted is related to the use of images in the text editor. Images may have arbitrary dimensions and may occupy large areas of the text editor, which may cause bad experiences in the utilization of this tool, affecting the readability of the code, and, even worse, disturbing the user in his primary task: programming.

To avoid this problem, I propose a straightforward mechanism that allows collapsing the images. As a result, images are reduced to the height of a character sentence, using considerably less space than before (e.g. see Figure 6). Once the picture is collapsed the programmer can easily navigate through the code, improving his awareness of the surrounding context in a file. Furthermore, this feature can be enhanced by DrRacket collapse *S-expressions tool* to elide code. However, differently from moderns code editors that collapse the code in blocks, when the architect wants to view part of the surrounding context, he either have to over the collapsed code to show a tool tip, possibly occluding relevant code, or expand the collapsed code possibly moving relevant code off-screen. DrRacket omits code by replacing it with "`(...)`", this way the surrounding context is still legible but uses less space. Inversely, users can expand their code as well as their images by a simple click. Note that all information placed in the picture, i.e. the annotated positions where the arrows will point, continue as before. However, when an image is collapsed this change is not persisted in the stored file, it

just sets the dimensions ports of the window visualization. Consequently, the programmer needs to perform this operation every time he starts a new session.

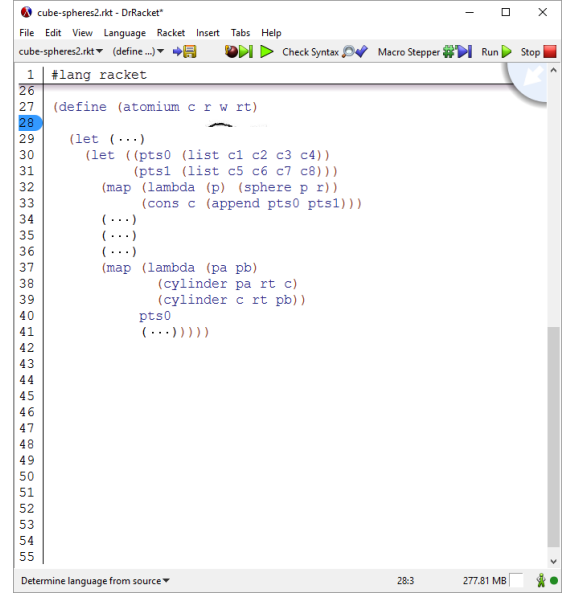


Fig. 6. Collapsing the image and some s-expressions. The code is collapsed, tidy, and the surrounding context is still legible.

In order to automatically recognize symbols in images and binding them with code identifiers, the presented architecture suggested the use of an optical character recognition (OCR) engine. Thus, the OCR engine is intended to receive the images, process their optical characters, and return the characters found with their respective coordinates on the picture. The first approach taken followed this description, sending the sketches, to the OCR. Unfortunately, this process was unfruitful, mainly because the technologies used in the current OCR engines have several problems to identify handwritten symbols (especially in the case of mathematical annotations) and they fail to recognize correctly the symbols used in the sketches in (almost) 100% of the times.

However, to finish the implementation of this tool without depending on the correct functioning of the OCR engine, I implemented a *fallback* mechanism that allows users to perform the OCR work. This way, when the first image of a function body is inserted in the code editor, it is blindly annotated with the same position for all function parameters. As a result, DrRacket check syntax will draw binding arrows, from the center of the image, to the function arguments, and vice-versa. For example, after the picture is inserted user can set its parameters. At this point, it is up to the programmer to change the bind position of each parameter to the correct position; the tool does not require this action.

Further, to finalize this process and have the right associations between image symbols and function identifiers, an alternative method was implemented. So, if the programmer clicks in any space of the picture, a GUI will appear showing three fields: (1) the parameter field, that is empty by default, and it is intended to specify the name of the clicked symbol, which, of course, must match with the function

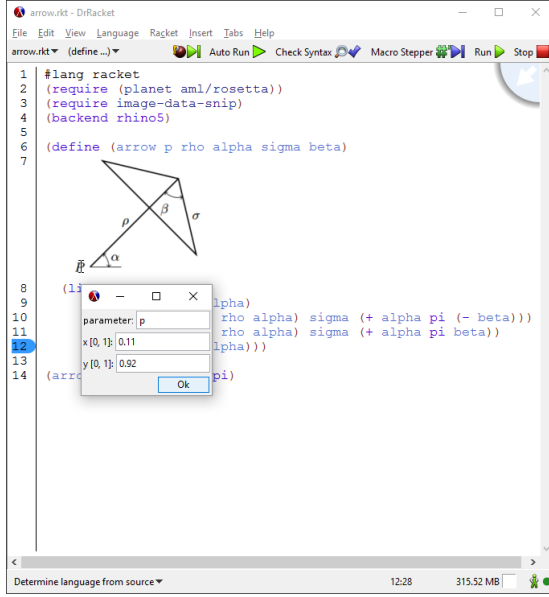


Fig. 7. Choosing the correct position for the function parameter  $p$ . The coordinates of the point  $p$  (illustrated in the image) as well as its correspondent function parameter is shown in the GUI.

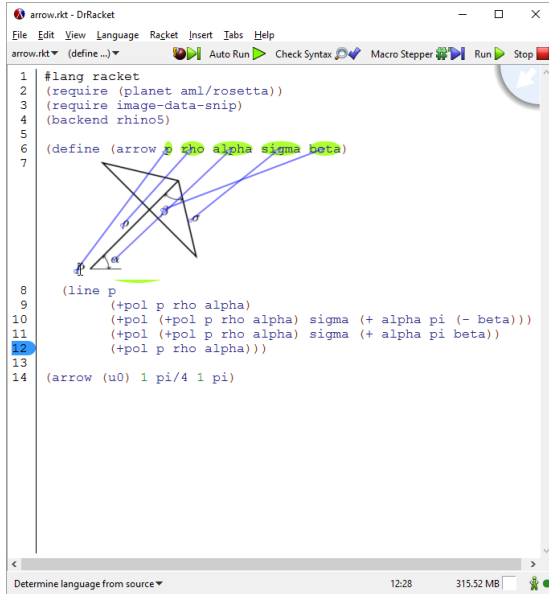


Fig. 8. Finishing the process setting all the image symbols in their proper place.

parameter name; (2) the  $x$ ; and (3) the  $y$  fields, that defines the width and height of the coordinate respectively (as shown in Figure 7). These areas are filled by default with the clicked position got from the image space. However if the user wants to specify another value, it is also possible.

To correct the others associations the programmer must repeat this process for each parameter. For example, Figure 8 shows the final result where all parameter is correctly identified. As a result, DrRacket can draw arrows directly to each one of them. Note that; this process is only performed once because the information associated with the image is stored with the source code file.

## B. Immediate feedback tool

As discussed previously, despite the apparent advantages of using a GD approach to model geometric objects, this method presents, at least, one important drawback that affects the design process. Architects cannot interact with their models as they build it. Any change in the model must be performed first in the code, and only after the program is compiled and executed the alterations will be available on the model. This process apparently takes more time than the traditional approach of manually building the model in the CAD tool. Besides, it discourages the artistic work because the changes are more costly than just manipulating a GUI element.

**Immediate feedback tool** aims to overcome this limitation, trying to promote the creative work in the design phase. This tool seeks to reduce as much as possible the time between a change in the code and its visualization in the model. Several techniques were suggested particularly considering the current state of GD environments.

Regarding some of these proposed methods, and trying to put them in practice, I implemented the immediate feedback tool on top of DrRacket. Once DrRacket is the programming environment used by Rosetta, any improvement achieved in this programming environment will be useful for both systems. So, I started by extending the current DrRacket environment developing an external tool, i.e. a *plugin*. The *plugin* was used to access the DrRacket application programming interface (API) without having to modify it.

Therefore, to use this tool users can install it separately, and uninstall it at any time. Once the tool is installed, a new icon will appear on the top of DrRacket Tool Bar. This icon has a symbol (similar to the DrRacket Run icon) and a description. However it has a responsive behavior: if the DrRacket window is too small the description will be omitted to save space and maintain the interface cohesive. When the user clicks on it, its color changes meaning that it is in action. Immediately after clicking, apparently nothing will occur until an expression is ready for evaluation.

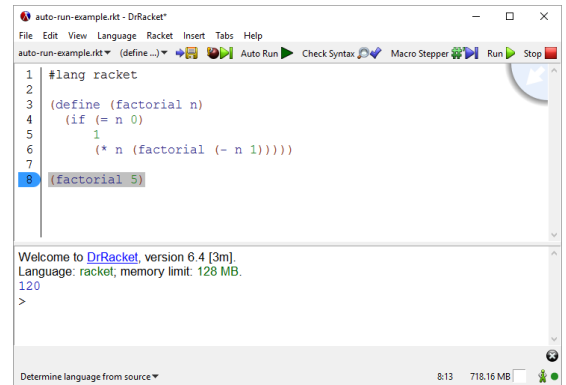


Fig. 9. Immediately after closing the right bracket of the highlighted expression. The Interactions Window is shown with the function result.

For example, Figure 9 shows an example where the *Auto Run tool* is activated and the programmer is writing the factorial function. At each character insertion, this tool is desperately trying to execute the code. However, to perform

this action, the code needs to be syntactically correct. On the other hand, the Check Syntax is validating the code, catching any syntax error and showing them at the bottom bar. Furthermore, when Check Syntax finishes its validation, and the code is syntactically correct, the AutoRun tool can operate. So it gets all the code validated previously, creates a callback, and sends it to the DrRacket evaluator. As a result, the code is immediately executed, as shown in Figure 9.

Unlike the factorial function that returns a numeric output, the functions in GD return a visual object. In this context, this tool is even more useful, because the changes made in the code causes a visual effect in the geometric object. The combination of this technique with the generated objects can provide an interactive environment, to experiment new ideas and test parametric models quickly.

However, the experimentation of program inputs relies heavily on the keyboard, breaking the fluidity of this process. For this, I used the DrRacket a widget mechanism to provide sliders at the program input. Thus, users can drag the slider to change the program input, rather than delete and insert a new character. Each shift in the slider causes an execution of the program with the new slider value. In GD programs this will generate a new geometric model. As a result, users can check if that value created the desired shape.

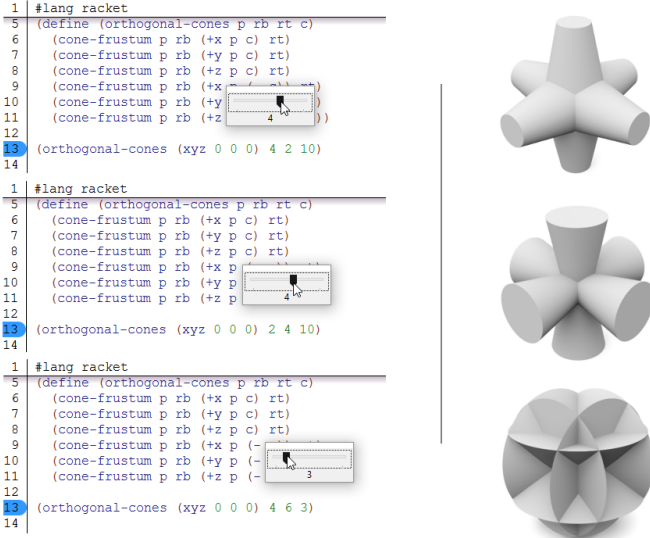


Fig. 10. Using the slider widget with *Auto Run tool* active, to experiment some input values. The highlighted function call (at line 13, on the left) is being changed by the slider (over the pointer). On the right, are the generated shapes rendered by AutoCAD.

Figure 10, shows the use of this tool to generate some geometric shapes. Despite the notable difference among the created forms, the function used to produce them is the same (this function is also shown in Figure 5). Therefore, the programmer can experiment several input values by just dragging the slider, once he achieved an intended design he can copy that line of execution (as a code comment for instance) and continue to experiment other values. The slider appears on top of a selected input number, to facilitate this process. It has a compressed window without any extra buttons, as a way

to be integrated with the text editor. Thus, users only need to click back into the text editor to close this window and be back to edit code.

Regarding the usability of this tool, the faster will be the backend response as better will be the user's experience. For example, consider the function `orthogonal-cones` (shown in Figure 5), its execution time is relatively straightforward (e.g. in AutoCAD backend it takes approx. 1 second). This is a tolerable delay between a model update, however using a more compounded geometry, such as the Mbius Truss, or even a complex geometry, such as a building, the execution time can grow considerably, and the immediate feedback becomes almost impossible.

## V. IMPLEMENTATION DETAILS

### A. General architecture

Figure 11 presents the general architecture of the solution, in a publish-subscribe view. There are two different interactions in this architecture, the first presented by a publish-subscribe, and the second by a client-server.

- 1) The main functionality of the proposed environment is made through a publish-subscribe interaction. The DrRacket UI event manager acts as an event bus for user-interface events (such as button clicks, character insertion, etc.). From this event bus, I subscribe only the UI events which are relevant to the system, defining which components will handle them. It is done at load time when the event manager reads the *plugin* configuration file (i.e. *info* file). When users are working on the editor, an UI event is generated and dispatched via implicit invocation to the action handler objects that subscribe to that event.
- 2) Despite the presented solution does not include the automatic character recognition, this module is part of the planned architecture, and can be considered, for a future integration. Thus, a client-server interaction will be needed to automatically recognize the manuscript symbols present in the image, using to this end an external OCR engine. Thereby in the suggested architecture the symbol identifier component calls this service to handle the recognition of symbols in the image. However, in the implemented architecture this component generates by itself the OCR data and all other modules continue to work as before.

The tool core component, in Figure 11, receives, three relevant kinds of DrRacket events. For each of these events, the programming environment is changed executing the following actions:

- **on-change**: when DrRacket detects that the editor has been modified, it sends the contents of the editor over to action handlers. In this case, it sends this information to the online expansion handler where the code is expanded. *Executed action*: sends a `execute` event to the editor frame, if the action handler expanded the code successfully.
- **on-paint**: this event is sent just before and just after every element is displayed in the editor. Handling this



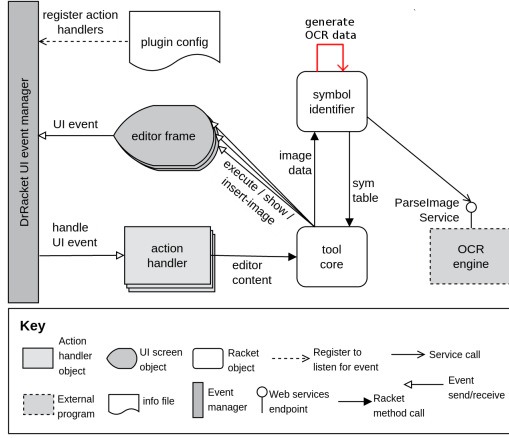


Fig. 11. Diagram for a publish subscribe view of the proposed architecture. The arrow, in red, on top of the `symbol identifier` module shows the implemented architecture.

event provides a way to add arbitrary graphics to the editor screen. *Executed action*: sends a `show` event to the editor frame, to display a slider widget when the user presses the mouse over a literal.

- `on-new-image-snip`: this event is sent when an image is inserted in the editor. The default implementation creates an image snip which is an object with the image information, such as path and format. *Executed action*: returns a subclass of image snip, containing an extra meta-data associated with the symbols in the sketch.

### B. Binding association

One of the first problems addressed in this thesis, and perhaps the most challenging one is finding an adequate framework that facilitates the correlation between image resources and code. When we consider DrRacket environment, this problem seems to have a straightforward resolution: DrRacket already provides support for media-rich data, and also a syntax check mechanism that graphically shows arrows from binding variables to their bound occurrences. Thus, the initial idea was to use the power of Syntax Check mechanism to bind variables to image as well. However, implementing this plan was harder than expected.

Following the initial idea, the first step was finding out the internal mechanism used by DrRacket to graphically show the arrows on mouse over. So, on each event, in the text editor, Check Syntax tool expands the program, annotating each identifier with syntax properties. Thereafter, to draw the arrows, it will look at a particular property added previously, named `'sub-range-binders`, which contains the necessary information to draw the arrows, such as the source and target location. This property is added only to syntax identifiers that exist in the program context. Therefore, any literal, comment, image or any other category are ignored.

To overcome this problem the decision made was to assume that the image will be inserted in the function body immediately after its definition. In this way, a syntactic transformer, i.e. macro, was created to handle the image, creating empty identifiers to bind its lexical context. For example, the Listing 1

shows a macro that matches the pattern of line 3, i.e. it assumes that the syntax object `img` is inside the function body, then it calls `lambda/point` to create binding associations between the image and the parameters. At line 5, the two constants 0.5, defines the center position of the arrow in the picture space. The possibility of varying the source position of the arrow indicators was not initially part of DrRacket API. Therefore, an additional implementation was requested to DrRacket authors.

```

1 (lambda/point
2   #,(map (lambda(x) '(,x 0.5 0.5))
3         (syntax-e #'(param ...)))
4   img
5   body ...)
```

Listing 1. Showing the initial solution. The macro `define/img` receives a syntax and returns a modified syntax, where the `img` is treated as an identifier associated to each function param.

Unfortunately, using the previous solution users cannot change the position of the arrows to different zones of the image. For example, the function `(define/img (func foo bar baz) ...)` will expand to the code shown in Listing 2. As we can see, in line 1, each parameter is wrapped in a list with two values that define a coordinate point (0.5, 0.5). This point represents the center of the image, where all parameters are statically associated. Therefore, the parameters of function `func` will point to the center of the image, being impossible to change them.

```

1 (define func
2   (lambda/point ((foo 0.5 0.5)
3                 (bar 0.5 0.5)
4                 (baz 0.5 0.5))
5   img
6   body ... ))
```

Listing 2. Showing the expansion of the macro `lambda/point`.

To solve this problem, the parameter coordinates previously passed as a constant were obtained during the macro expansion. The main challenge of this solution was finding out a way to access this information in the macro context, because the syntactic object, received in the macro, which represents the image, i.e. `img`, does not exist yet. The syntax transformation is done in compile time, and this object is only completely available at run time. Therefore, any attempt to make a function call using this syntactic object will fail. Once the traditional methods used to access an object field does not work, other strategies were considered.

```

1 (let ((metadata
2       (syntax-property #'img
3                         'args)))
```

Listing 3. Showing the expansion of the macro `lambda/point`.

After several attempts to add extra information in a image, such as use an external tool to write in the bitmap fields, up to use libraries that allow rewrite image metadata, the final solution was extend an existing DrRacket class. In this way, the information required to bind code with the picture was stored

in a field of this new class. Using this class, it was possible to have control over the syntax reader process, annotating the image with the information required by the macro at compile time. Thus, to access the picture metadata during the macro expansion, a single line of code, as shown in Listing 3, was added between the line 3 and 4 of Listing 1. Moreover, this class implements several other facilities which are described in the next section.

### C. Image-data-snip

The `image-data-snip%` class, shown in Figure 12, has a fundamental role in the correlation between image and code. This class provides facilities such as create, read, store and access the coordinates associated to each function parameter. Therefore, the original class used in DrRacket to display images, i.e. `image-snip%`, was extended with a hash table field which contains the parameter identifiers, as a key, and their coordinates, in the image space, as a value. Moreover, this class implements its syntax read method to provide the picture metadata through the macro expansion.

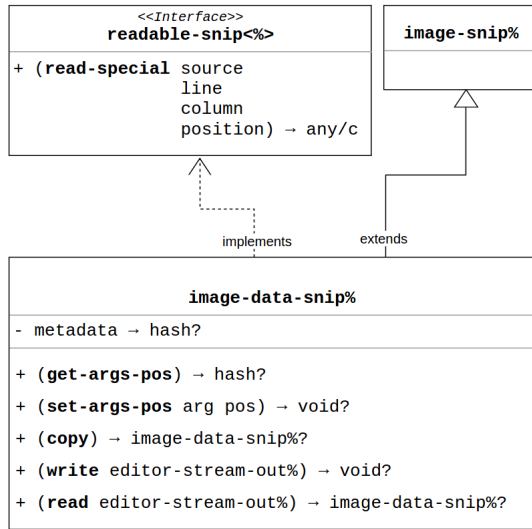


Fig. 12. The UML diagram of the implemented class, `image-data-snip%`. This class extends the original DrRacket `snip` class, and implements its own read syntax method.

Because `image-data-snip%` is a subclass of `image-snip%`, it saves code and its associated metadata in ASCII-encoded binary format. This class uses the superclass method to store the image, but the methods for copying, reading, and storing the metadata field, were implemented from scratch. Unfortunately, the final serialized file does not provide backwards-compatibility among other text editors.

Despite the beginning effort spent to implement this class, other features can benefit from this work. For example, the feature for expanding/collapsing images was applied on top of this class using only a few lines of code. Once the subclass already provides a method for resizing snips, this feature does a simple, super class call to change the size of the `snip` window. Moreover, this class defines a basis for other features that manipulates, or even edits, the image inserted in the text editor.

### D. Auto Run

The core of `AutoRun plugin` is based on the `snip` of code shown in Listing 4. This sample of code registers a pair of procedures with DrRackets online expansion machinery. The first two arguments name a method in a module that is loaded by `dynamic-require`. When DrRacket detects that the editor has been modified, it sends the contents of the editor to that separate place, expands the program there, and then supplies the fully expanded object. Then, this procedure (defined at line 4-10) gets the DrRacket editor frame (at line 6) and executes the code in that frame using the `execute-callback` function (at line 9-10). Therefore, upon a change in the text editor, DrRacket will execute the code immediately.

```

1 (drracket:expansion-handler
2   online-comp.rkt
3   'go
4   (lambda ()
5     (...); collapsed code
6     (define drr-frame
7       (send (send defs-text get-tab)
8             get-frame))
9     (send drr-frame
10          execute-callback)))

```

Listing 4. Executing the program through DrRacket `execute-callback`.

When this *plugin* is installed, it can become quite evasive once it is running code without the user permission. To avoid this situation, the *plugin* is installed as a button in DrRacket's toolbar, having two operating modes. First, it is in disabled mode, which means that users can edit the code as before, and this *plugin* will have no effect in DrRacket. Second, when it is on enable mode, it tries to execute the code at each change. However, to change the *plugin* methods, users must authorize it, by clicking in the *plugin* button.

Finally, to improve this tool an interactive mechanism to give inputs to the program was implemented, using the DrRacket slider widget. In this way, when a program literal is selected, and a keybinding is pressed, a slider appears on top of that literal. So, when the slider is dragged, the editor changes, which causes the `AutoRun plugin` to execute the code.

Unfortunately, adapting the range of values in the slider, interactively as users drag it, becomes difficult or even impossible to implement, using the DrRacket slider. To create a new slider is necessary to define the range and the initial value of that slider. However, to update these initial values, after the slider window is shown, the frame must be closed and opened again, which goes against the purpose of this tool.

## VI. CONCLUSIONS

The representation of source code dramatically affects its comprehensibility and usability [26]. In this thesis, we found that by using image resources, namely sketches and diagrams, applied to generative design programs it is possible to enhance the program visualization. Moreover, accelerating the program execution allows users to test their mental models [4], [27] quickly. In fact program comprehension is improved because

(1) source code becomes documented with media-rich resources, and (2) the effect of changes in the program becomes instantly visible.

Based on Learnable Programming and Literate Programming we propose an interactive environment tailored for generative design. This programming environment helps the designer in establishing a strong correlation between the GD program, and the geometric sketches that it represents, as a result, it eliminates the first barriers to learning, allowing designers to read the code and to understand it at a higher level. It also encourages the developer to test his ideas quickly, by seeing the result of his action.

Among the generative design systems which support programming in a textual form [16], [17], only one [17] supports other elements in the editor, besides plain text. However, none of them associates the source code with images nor even supporting any correlation between them. On the other hand, the only way to get immediate feedback in these systems is by a stepwise debug which stops the entire program execution, disabling code to be edited. So in debug mode users can only change single lines of execution, making it difficult, or even impossible, to have immediate feedback.

Immediate feedback avoids the edit-compile-run cycle, anticipating as much as possible the result of a change in the code. When this mechanism is associated with Rosetta tool, it creates a playground where architects can create models interactively. So, this tool promotes the essential aspect in Architecture: the creativity. However, similar to what happens with all other GD systems, this tool will not scale for all GD programs, due to the render performance of CAD tools. Therefore, using Rosetta architects can use a faster backend to test their ideas, then moving to their preferred backend.

Sketch-program correlation mechanism promotes the documentation in GD programs besides it correlates the source code with images. However, to associate images with code users must intervene in this process by strictly identifying the symbols in the picture with the identities in the program. Integration with OCR engine is part of the system architecture, however due to the bad experiences in using the OCR to recognize handwritten symbols, we skipped this step in the actual implementation.

In this thesis, several features were proposed, the majority of them based on successful research projects. For example, the idea of improving the context view of statement blocks by allowing users to focus on the main fragment of code was proposed in Barista [28] project, and we used it to implement the expand/collapse feature. Like this idea, other helpful features and strategies were discussed, such as how to serialize images with code, how to prevent that images impair in the usability of the code text editor, among others.

The implemented tools just define a first path to follow. Future work, regarding the program-sketch tool would (1) integrate the OCR engine in the current flow, (2) provide facilities to allow users to edit images directly in the code editor, (3) provide flexibility to bind any program fragment to any picture. Regarding the immediate feedback tool, future work would (1) create an adaptive sliders widget, (2) study a better mechanism to predict the slider values, (3) implement

in CAD backends the possibility to show differences between model changes, and (4) implement the autocomplete tool that immediately executes the selected expression in the list of options.

We believe that there is a bright future ahead for innovations in programming environments, and we hope that implementation programming tools like those discussed in this thesis will make such innovations more feasible, more usable, and more useful.

## REFERENCES

- [1] R. W. Hamming, *The Art of Doing Science and Engineering: Learning to Learn*. Boca Raton, Florida, USA: CRC Press, 2003.
- [2] F. P. Brooks, "No Silver Bullet – Essence and Accidents of Software Engineering," *IEEE Computer*, vol. 20, no. 4, pp. 10–19, 1987.
- [3] S. Rugaber, "Program Comprehension," *Encyclopedia of Computer Science and Technology*, vol. 35, no. 20, pp. 341–368, 1995.
- [4] R. Brooks, "Towards a theory of the cognitive processes in computer programming," *International Journal of Man–Machine Studies*, vol. 9, no. 6, pp. 737–751, 1977.
- [5] J. DesRivieres and J. Wiegand, "Eclipse: A platform for integrating development tools," *IBM Systems Journal*, vol. 43, no. 12, pp. 371–383, 2004.
- [6] D. Carlson, *Eclipse Distilled*. Cambridge, Massachusetts, USA: Addison Wesley Reading, Feb. 2005.
- [7] T. Boudreau, J. Glick, S. Greene, V. Spurlin, and J. J. Woehr, *NetBeans: the definitive guide*. Boston, USA: O'Reilly Media, Inc., Nov. 2002.
- [8] H. Böck, "IntelliJ IDEA and the NetBeans Platform," in *The Definitive Guide to NetBeans Platform 7*. New York, NY, USA: Springer, 2011, pp. 431–437.
- [9] S. Guckenheimer and J. J. Perez, *Software Engineering with Microsoft Visual Studio Team System (Microsoft .NET Development Series)*. Addison-Wesley Professional, 2006.
- [10] S. Papert, *Mindstorms: Children, computers, and powerful ideas*. New York, NY, USA: Basic Books, Inc., July 1993.
- [11] A. C. Kay, "The early history of Smalltalk," in *History of programming languages — II*. New York, NY, USA: ACM, 1996, pp. 511–598.
- [12] C. Reas and B. Fry, "Processing: Programming for the media arts," *AI & SOCIETY*, vol. 20, no. 4, pp. 526–538, 2006.
- [13] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen, "DrScheme: A programming environment for Scheme," *Journal of functional programming*, vol. 12, no. 02, pp. 159–182, 2002.
- [14] P. J. Guo, "Online Python Tutor: Embeddable Web-based Program Visualization for CS Education," in *Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '13. New York, NY, USA: ACM, 2013, pp. 579–584. [Online]. Available: <http://doi.acm.org/10.1145/2445196.2445368>
- [15] S. McDirmid, "Usable live programming," in *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. New York, NY, USA: ACM, 2013, pp. 53–62.
- [16] R. Aish, "DesignScript: origins, explanation, illustration," in *Computational Design Modelling*. Springer, 2012, pp. 1–8.
- [17] J. Lopes and A. Leitão, "Portable generative design for cad applications," in *Proceedings of the 31st annual conference of the Association for Computer Aided Design in Architecture*, 2011, pp. 196–203.
- [18] J. McCormack, A. Dorin, T. Innocent et al., "Generative Design: a paradigm for design research," *Proceedings of Futureground, Design Research Society, Melbourne*, 2004.
- [19] E. Y.-L. Do and M. D. Gross, "Thinking with diagrams in architectural design," in *Thinking with Diagrams*. New York, NY, USA: Springer, 2001, pp. 135–149.
- [20] S. C. B. Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, no. 7. New York, NY, USA: ACM, 2005, pp. 68–75.
- [21] M. J. C. Sousa and H. M. Moreira, "A survey on the software maintenance process," in *Software Maintenance, 1998. Proceedings., International Conference on*. IEEE, 1998, pp. 265–274.

- [22] B. Victor, “Learnable Programming – Designing a programming system for understanding programs,” Retrieved from <http://worrydream.com/LearnableProgramming>, Jan. 2014.
- [23] —, “Inventing on principle,” Invited talk at the Canadian University Software Engineering Conference (CUSEC), Montreal, Canada, Jan. 2012.
- [24] B. A. Myers, “Taxonomies of visual programming and program visualization,” *Journal of Visual Languages & Computing*, vol. 1, no. 1, pp. 97–123, 1990.
- [25] A. Leitão and L. Santos, “Programming languages for generative design: Visual or textual?” in *Zupancic, T., Juvancic, M., Verovsek., S. and Jutraz, A., eds., Respecting Fragile Places, 29th eCAADe Conference Proceedings, University of Ljubljana, Faculty of Architecture (Slovenia), Ljubljana*, 2011, pp. 549–557.
- [26] R. Baecker and A. Marcus, “Design principles for the enhanced presentation of computer program source text,” in *ACM SIGCHI Bulletin*, vol. 17, no. 4. ACM, 1986, pp. 51–58.
- [27] A. Robins, J. Rountree, and N. Rountree, “Learning and teaching programming: A review and discussion,” *Computer science education*, vol. 13, no. 2, pp. 137–172, 2003.
- [28] A. J. Ko and B. A. Myers, “Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2006, pp. 387–396.