Programming Environments for Generative Design

Guilherme Ferreira
guilherme.ferreira@tecnico.ulisboa.pt

Instituto Superior Técnico
Universidade de Lisboa

Abstract. Programs are rarely released with useful documentation. Therefore, when the documentation is poor, obsolete, or absent, the remaining option is to study the program itself. Over the years, many environments have been proposed to write, document, and comprehend a program. This report presents a survey of those environments, and proposes some innovative tools to improve the current state of the art.

1 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 [6–9], and simple tools designed for learning environments [10–15]. 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.

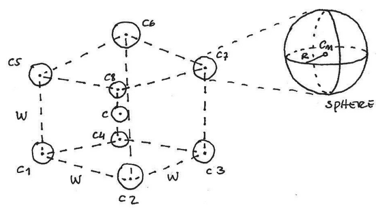


Fig. 1: An architectural sketch.

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.

2 Objectives

This work addresses two challenges:

– minimize the lack of documentation in the programs.
– improve the program comprehension process.

To overcome these challenges, we will investigate better ways to help programmers in their conceptual tasks. The goal is to design and implement innovative tools which support and encourage new ways of thinking, and therefore, enabling programmers to more easily see and understand their programs.

Our approach to achieve this objective is to, 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 will implement two interactive tools tailored for generative design programs:

1. *Sketch-program correlation tool,* which will encourage architects and designers to reuse their conceptual sketches, such as that shown in Figure 1, to visually document their programs. These sketches will be 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 will give a new medium for architects and designers to create new ideas by continuously reacting with changes in their models. This tool will minimize the latency between writing the code and executing it, consequently this will encourage users to experiment ideas quickly, augmenting their comprehension about the program.

This thesis will produce 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.

3 Related Work

A *programming system* has two fundamental parts: the *programming language* that users should know, 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.

In the following sections we divide the programming systems in three categories: (3.1) general-purpose systems; proposed for building complex software, (3.2) teaching systems; proposed to teach programming, and (3.3) empowering systems; proposed to build programs tailored to specific needs. In each category, we focus on the tools provided by the programming environment.

3.1 General-purpose systems

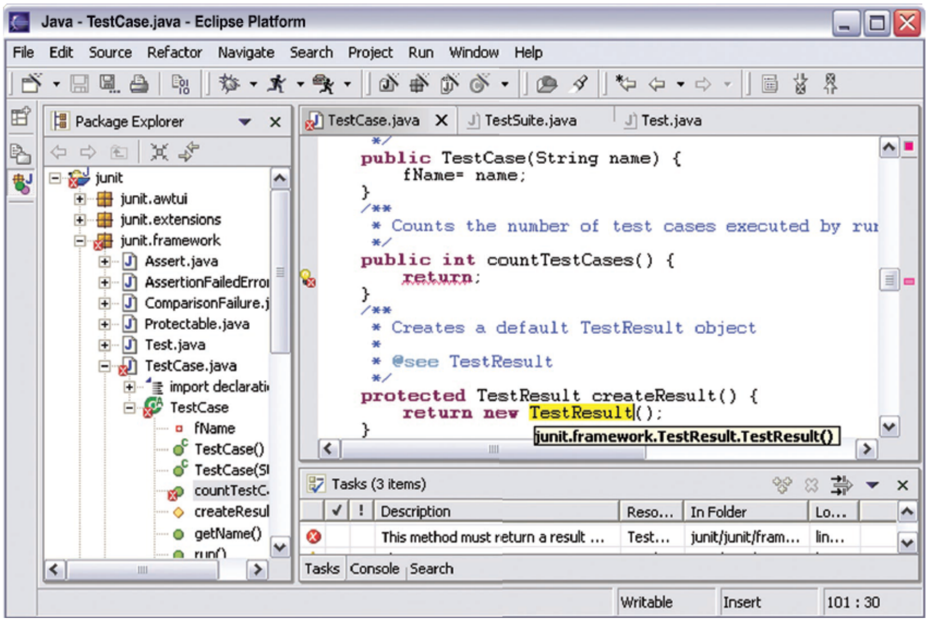
The systems in this category are built to support all, or at least a substantial part, of the software development process. To this end, these systems suggest an integrated development environment (IDE) that aims to support the entire development process by grouping in a single environment all necessary tools.

In this section we describe, in detail, two relevant IDEs: Eclipse [6] and LightTable[1]. For the sake of comparison, we include other IDEs which provide similar features to Eclipse (e.g. NetBeans [7], IntelliJ [8], and Microsoft Visual Studio (MVS) [9]), and similar features to LighTable (e.g. Xcode[2]).

[1] `http://lighttable.com/`
[2] `https://developer.apple.com/xcode/`

Eclipse [6] is a popular IDE used mainly by Java developers, although it supports other programming languages such as C, C++ and JavaScript. As showed in [24], the commonly cited reasons for using Eclipse include rich Java development tools support and a *plugin* architecture, the Eclipse Platform [5], that allows tight integration of third-party functionality.

The Eclipse platform [5] has a common architecture among the IDEs presented in this report. This architecture is characterized by two main components, the *plugin* which is the smallest unit of functionality that can be developed and delivered separately, and the platform runtime which will discover and connect the *plugins* to the platform itself. As a result, the platform integrates several tools that are used in distinct phases of the software development process. Figure 2 shows how the IDE looks like when the user is writing a Java program.



```
    */
    public TestCase(String name) {
        fName= name;
    }
    /**
     * Counts the number of test cases executed by rur
     */
    public int countTestCases() {
        return;
    }
    /**
     * Creates a default TestResult object
     *
     * @see TestResult
     */
    protected TestResult createResult() {
        return new TestResult();
    }
```

junit.framework.TestResult.TestResult()

	✓	!	Description	Reso...	In Folder	Lo...
⊗			This method must return a result ...	Test...	junit/junit/fram...	lin...

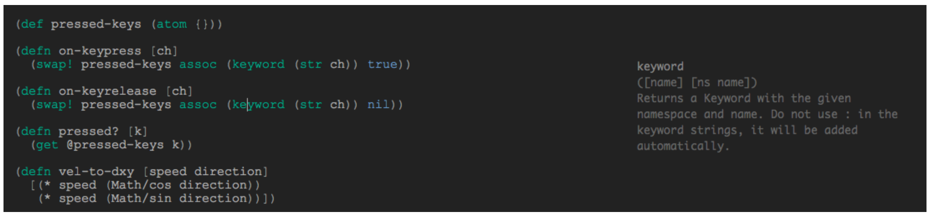
Fig. 2: Eclipse IDE showing Java perspective.

Eclipse has many features that provide feedback to the programmers about what they are constructing, such as syntax highlighting, code completion suggestions, and indications of problems associated with various locations in a source file. In the literature, this concept is also known as *liveness* [25]. It generally refers to the ability to modify a running program, however, as show in recent studies [26], there are several levels of liveness and these tools represent the first ones. While the tools in the first levels respond after a programmer action, the tools in the last levels not only run the program and respond immediately, but also predict the next programmer action. The modern IDEs tools, such as the Eclipse tools, are far from achieving the last liveness levels.

Moreover, both Eclipse and the other similar IDEs also share the following drawbacks:

– They are incidentally complex. There is an immense amount of work to be done in those IDEs that is indirectly related to the real problem itself. For example, until the programmer gets a simple program to run, he needs to install and configure a set of necessary software, and he needs also to configure the development environment. This is a tiresome and time consuming task that adds an extra complexity into programming which is already a complex subject.
– Program execution is difficult to observe, such that the only way to see how the program executes is by a stepwise debugger. This forces the programmer to stop the program and look at a line in a single instant of time. Consequently, the programmer cannot see how his program is executing, nor how his changes affect its execution.

LightTable is a programming environment which aims to turn programming in a observable task. Bret Victor, in his influential work [23, 22], pointed out serious problems with the current environments and showed, using prototypes, how the environment can help to address those problems. LighTable is implemented base on those ideas, and it is designed to build web applications. To support this process, LightTable provides, at least, two useful features: (1) live execution feedback, that executes the program on every change showing the program flow, and (2) the organization of code in tables, enabling quick access to the program documentation.



```
(def pressed-keys (atom {}))

(defn on-keypress [ch]
  (swap! pressed-keys assoc (keyword (str ch)) true))

(defn on-keyrelease [ch]
  (swap! pressed-keys assoc (keyword (str ch)) nil))

(defn pressed? [k]
  (get @pressed-keys k))

(defn vel-to-dxy [speed direction]
  [(* speed (Math/cos direction))
   (* speed (Math/sin direction))])
```

```
keyword
([name] [ns name])
Returns a Keyword with the given
namespace and name. Do not use : in the
keyword strings, it will be added
automatically.
```

Fig. 3: LighTable IDE.

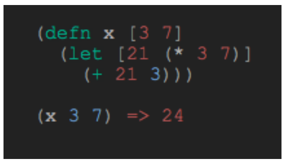
LightTable is implemented in ClojureScript[3] (a Clojure compiler that targets JavaScript). Due to this implementation, in LightTable adding a new user interface (UI) element into the programming environment or changing an existing one is doable in a short amount of time, contrary to other IDEs, such as Eclipse [6], where an equivalent change requires considerable amounts of time.

The program documentation in LightTable is quickly accessed by a lateral tab, where primitive functions of Clojure and ClojureScript can be consulted. This documentation is a textual description of the function parameters, the type of return, and some usage suggestions. While looking at a program, it is helpful

[3] `http://clojure.org/clojurescript`

to have the documentation of strange primitives, such as the `keyword` function shown in Figure 3, however non-primitive functions are still undocumented.

On the other hand, programmers are encouraged to understand new functions by seeing how the values of a function call flow through it. This feature is based on the old idea of Lisp environments: the read-eval-print loop (REPL) which is a prompt used to try out expressions of the language without having to run all the code. This approach goes further by using reflection mechanisms to trace the function call values and shows them filled in the function template (as shown in Figure 4 the flow of values produced by calling `(x 3 7)`).



```
(defn x [3 7]
  (let [21 (* 3 7)]
    (+ 21 3)))

(x 3 7) => 24
```

Fig. 4: LightTable real-time debugger.

The real-timer debugger is an interactive way to debug the code and understand the program flow. Using this feature in an arbitrarily complex program (a program with more than 30 functions) is, however, worthless, because, all the programmer sees is a replica of his functions filled with numbers. It is a poor representation of flow which forces the programmer to spend as much effort with this feature as without it. For this reason, other systems, described in this report, represent the program flow using graphs which are more appropriate in some cases, for example, to show error occurrences in the source code.

Despite providing some tools which are state of the art, LightTable remains in an experimental phase. It has serious limitations to identify and clearly present the errors in the source code. This problem is, mainly, related with the Clojure compiler which loses significant *metadata* between conversions. Consequently, programmers can spend more time and effort to find a bug using LighTable, than using the other IDEs, such as Eclipse.

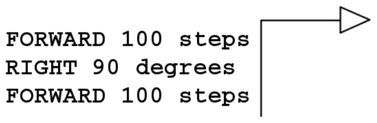
3.2 Teaching systems

Unlike the previous systems, teaching systems are designed with the goal of helping people learning to program. Most of the systems in this category provide simple programming tools that expose the novice programmers some of the fundamental aspects of the programming process. After acquiring experience with a teaching system, students are expected to move to a more general-purpose environment.

LOGO [10] is a programming language and environment intended to allow children to explore a wide variety of topics such as physics and mathematics. The programming language is a dialect of Lisp, with much of the punctuation removed to make the syntax accessible to children, it uses a helpful metaphor which facilitates the introduction of programming concepts.

In Logo, the programmer draws pictures by directing the “turtle”, an on-screen character which leaves a trail as it moves (see Figure 5). The turtle is a metaphor that helps learners to translate their experiences as a person into programming knowledge. That means, to figure out how to make the turtle perform an action, the programmer can ask how he would perform that action himself, as if he were the turtle.

For example, to figure out how to draw a circle, a learner would walk around in circles for a bit, and quickly derive a "circle procedure" of taking a step forward, turning a bit, taking another step forward, turning a bit. After teaching it to himself, the learner can then teach it to the computer.



```
FORWARD 100 steps
RIGHT 90 degrees
FORWARD 100 steps
```

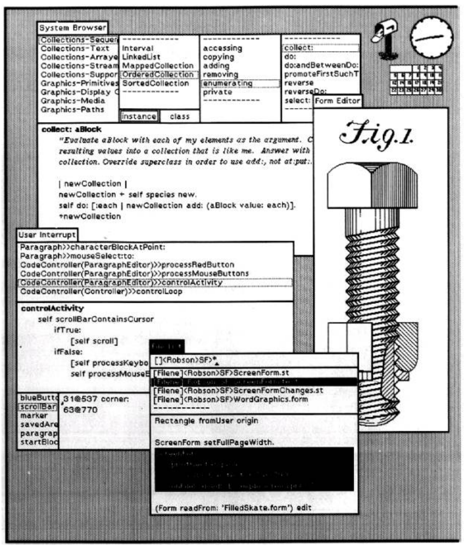
Fig. 5: Directing the "turtle".

LOGO has influenced several systems, and its principles show how a system can be designed around the way people think and learn.

SmallTalk [11] is a programming language and environment to support children in the world of information. The designers of this system, wanted to create a programming language that had a simple model of execution and a programming methodology that could accommodate a wide variety of programming styles. SmallTalk was based around three ideas: (1) everything is an object, (2) objects have memory in the form of other objects, (3) and objects can communicate with each other through messages.

Smalltalk programming environment was a successful achievement with relevant improvements on its successors. The system consisted of about 50 classes described in about 180 pages of source code [11]. This included all of the OS functions, files, printing and other Ethernet services, the window interface, editors, graphics and painting systems, as shown in Figure 6.

In the Smalltalk programming language, the communication through messages has a strong resonant metaphor. To specify the behavior of an object, the programmer casts himself into the role of that object (to the extent of referring to the object as "self") and thinks of himself as carrying on a conversation with other objects. This is a strong metaphor, because role-playing and conversing are innate human facilities.



```
collect: aBlock
    "Evaluate aBlock with each of my elements as the argument. C
    resulting values into a collection that is like me. Answer with
    collection. Override superclass in order to use add:, not at:put:.

    | newCollection |
    newCollection ← self species new.
    self do: [:each | newCollection add: (aBlock value: each)].
    ↑newCollection
```

```
Paragraph>>characterBlockAtPoint:
Paragraph>>mouseSelect:to:
CodeController(ParagraphEditor)>>processRedButton
CodeController(ParagraphEditor)>>processMouseButtons
CodeController(ParagraphEditor)>>controlActivity
CodeController(Controller)>>controlLoop
```

```
controlActivity
    self scrollBarContainsCursor
        ifTrue:
            [self scroll]
        ifFalse:
            [self processKeybo
            self processMouseB
```

```
31@537 corner:
63@770
```

```
[]<Robson>SF>*
[Filene]<Robson>SF>ScreenForm.st
[Filene]<Robson>SF>ScreenFormChanges.st
[Filene]<Robson>SF>WordGraphics.form
------------
Rectangle fromUser origin

ScreenForm setFullPageWidth.

(Form readFrom: 'FilledSkate.form') edit
```

Fig. 6: Smalltalk user interface.

In fact, SmallTalk features are, nowadays, a consistent reference for any programming system.

Processing [12] is a programming language and environment designed to teach programming in a visual 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`. 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. Despite in the default mode the result is presented in a 2D graphical window, the render can be configured to present the result in 3D or in 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 HMTL5 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 fundamental approach to solving a complex problem by breaking it into simpler problems, because drawing and input events are tied to single entry points. Thereby the behavior of submodules must be tangled across these global functions, making it difficult to achieve clean decomposition.
– *Poor recomposition.* Processing discourages combining two programs. The programmer 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 meaning 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 lookup 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 with the language and its implementation.

Fluxus[4] is a programming language and learning environment designed for rapid prototype using 3D graphics and sounds. This emphasis on rapid prototype and quick feedback makes Fluxus a tool for learning computer animation, graphics and programming. However, most users of Fluxus use it for *livecoding*, which is the act of performing coding lively to an audience.

Fluxus is mainly written in C++ and it is statically linked to several shared libraries, specified at compile time. For instance, Fluxus uses `jack-audio`, `ode`, and `fftw` libraries to handle and synchronize the audio, `GLEW` to present graphics, and `racket3m` to embed the Racket run-time system into the application. In this way, Fluxus is an extension of Racket (a descendant of Scheme) with graphical commands.

Like Processing, Fluxus provides simple access to those libraries through single entry points, for instance `start-audio` connects an input audio to the application, `every-frame` registers a function called once per frame, and so on. However, as stated above, it is a barrier for decomposition since the behavior of submodules must be tangled across these global functions.

Fluxus has its own environment specifically tailored for *livecoding*. It is composed by a OpenGL graphical window with a simple text editor. The programmer types his code in the editor and presses a shortcut key each time he wants to run the code. Fluxus evaluates the code through Racket run-time system and shows its result in the same graphical window that the code was written. This mechanism is valuable in an *livecoding* environment, because the performer can be editing the code while the result of the previous computation is maintained in background. However, if the code has any error and the performer execute it, the previous computation disappears and the environment will not help to find it. This is a serious problem, specially for the Racket syntax.

Moreover, Fluxus shares with Processing similar drawbacks to those previously stated. However Fluxus can be used as a module of DrRacket [13] programming environment and, fortunately, in DrRacket the above problem and many others are solved.

DrRacket [13] is a programming environment designed to support the Racket language. DrRacket is one of few programming environments which supports gradual learning in a more general language from the start. Consequently, it has been widely used in introductory programming courses in several universities around the world.

The usual scenario where DrRacket is used is to teach functional programming using Racket. To facilitate this process, DrRacket provides three tools. The first is a symbolic stepper. It models the execution of Racket programs as

[4] `http://www.pawfal.org/fluxus/`

algebraic reductions, since Racket is implemented on top of lambda calculus. The second tool is a syntax checker. It annotates programs with font and color changes based on the syntactic structure of the program. It also permits students to explore the lexical structure of their programs graphically and to α-rename identifiers. The third tool is a static debugger that infers which set of values an expression may produce and how values flow from place to place in the source text, and, upon demand, it explains the reason of errors by drawing value flow graphs over the program text.

Similar to Lisp environments, DrRacket provides a read-eval-print loop (REPL). This is a command prompt intended to quickly evaluate expressions and print their results. Especially in a learning environment, this feature makes an important connection between program execution and algebraic expression evaluation. However, the Lisp-style syntax obscures the effect of this feature. To overcome this limitation, DrRacket provides a `pretty-printer`. A module capable of printing algebraic expressions in a meaningful way, as well as other graphics elements supported by the text editor, such as images, snips, XML boxes, and so on.

From the perspective of professional programmers DrRacket can be a potential target. It is useful for developing complex applications, including DrRacket itself. Moreover it is extensible by the same application programming interface (API) which the above tools implement. Through this API it is also possible to extend the REPL, as in Pict3D[5] (a 3D engine that integrates new graphical elements in the DrRacket environment). On the other hand, for supporting extensions, DrRacket's architecture has become increasingly complex. For instance, to make a simple change in an editor's element the programmer should be able to understand several modules, unrelated with the problem itself. This extra complexity is a negative impact when DrRacket is chosen as basis for new development tools.

Despite of the identified advantages, DrRacket has some barriers that may discourage the learner. For example, the Racket programming language is simple to teach, but its heavy syntax of s-expressions hinders the learner to read the program. Consequently, the learner can spend a huge mental effort to understand insignificant details of the language.

PythonTutor [14] is a web-based program visualization tool, designed to explain how a piece of Python code executes. It has become popular among students from introductory Computer Science courses. Using this tool, teachers and students can write Python programs directly in the web browser and navigate step by step throughout its execution, seeing the run-time state of data structures.

PyhtonTutor has two main modules: the *backend* which implements the tool core functionality, and the *frontend* which presents the visualization of program's data structures. The *backend* executes the input program under supervision of the standard Python debugger module (`bdb`) which stops execution after every

[5] `https://github.com/ntoronto/pict3d`

executed line and records the program's run-time state. After execution terminates, the *backend* encodes the program state in JSON format, serializing Python data types into native JSON types with extra *metadata* tags and sends it to the *frontend*. The frontend renders the objects using standard web technologies: HTML, CSS, and JavaScript. In this way, users can use the tool without installing any extensions or plugins.

A major concern in PythonTutor is security, because the PythonTutor's *backend* executes untrusted Python code from the web. To prevent the execution of dangerous constructs such as `eval`, `exec` and `file I/O`, PythonTutor implements sandboxing. Basically, it denies the use of most module imports, by parsing the user's code importing, a strict approach, but effective in this case.

The PythonTutor tool allows the programmer to follow the program execution over time, but he only sees a single point in time at any instant. There is no visual context at all. The entire program flow is represented by disconnected points in time. For example, the programmer who wants to understand a conditional algorithm, using this tool will not see the pattern of this algorithm neither understand it at a higher level.

YinYang [15] is a prototype of a programming language and environment whose main feature is the live execution feedback. That means it combines editing and debugging, where updated debug results are conveniently visible while editing. YingYang addresses the above issue in two ways. First, just like the previous system, it allows programmers to see single points of execution directly within the code editor (probe; precede expressions with `@` operator). Second, it has a pane aside the editor which traces execution with entries that are navigable (trace; print-like statements). Basically, the trace is an enhanced display function which, combined with "probes", allows the state of previous executions to be restored. So, programmers can take in the entire program flow at a glance and navigate trough it using probes.

YingYang uses an incremental framework as basis of its programming model. This framework decomposes the program execution into a tree of nodes that can be re-executed independently on a code or input change. However, this decomposition cannot be performed transparently. It requires programmers to specify how the program will be decomposed. To perform this task one must deeply understand the granularity and modularity characteristics of the computations being performed by the program. Otherwise changes can sometimes have a huge impact on program re-execution time ($\sim$50ms). Consequently, live programming would actually reduce programmer productivity as programmers wait for slow feedback.

Although YinYang provides usable features for a learning environment, such as the live execution feedback, it does not have a suitable language for beginners. The language is merely experimental and to navigate through the program execution, programmers must include probes in the code. At the end of experimentation, the code is full of useless expressions.

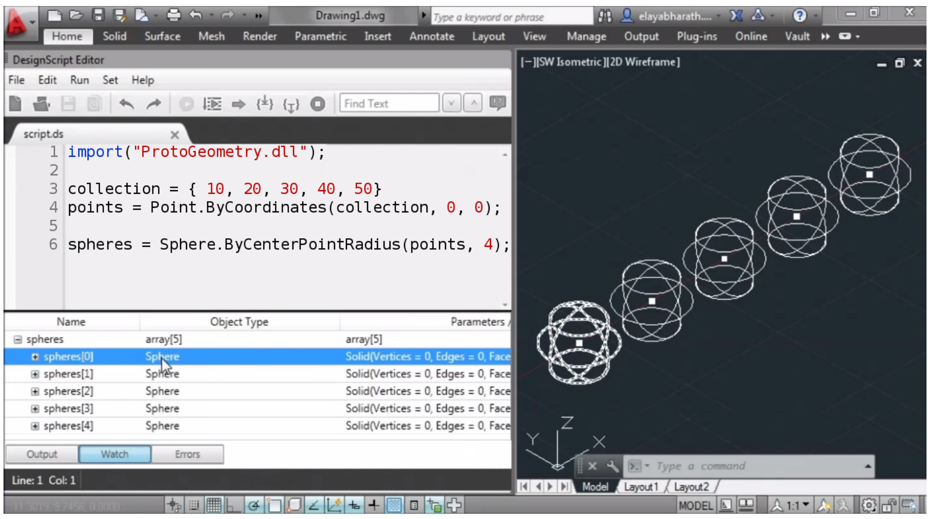
3.3 Empowering Systems

In this category of systems, the most important aspect is to allow people to build programs tailored to their own needs. In this section, we describe how systems from two distinct areas are tailored to achieve their user's needs.

First, we consider Architecture, where new programming languages and environments are being proposed to support the increasing use of generative design (GD) [18]. GD is a design method that uses algorithms to generate architectural models. Usually these models are rendered using a computer-aided design (CAD) tool.

Second, we consider Mathematics, where advanced technologies are used to approximate as much as possible the mathematical models to the ones that we can see and understand.

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. When the script is executed it generates new models in a CAD tool. DesignScript is a AutoDesk[6] product initially proposed to be used within AutoCAD (as shown in Figure 7), 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 covers spatial relationships, properties of building components, such as manufacturers' details.



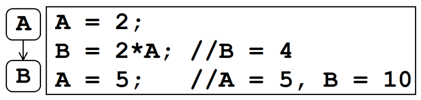
```
import("ProtoGeometry.dll");

collection = { 10, 20, 30, 40, 50}
points = Point.ByCoordinates(collection, 0, 0);

spheres = Sphere.ByCenterPointRadius(points, 4);
```

```
Name              Object Type   Parameters
⊟ spheres         array[5]      array[5]
  ⊞ spheres[0]    Sphere        Solid(Vertices = 0, Edges = 0, Face
  ⊞ spheres[1]    Sphere        Solid(Vertices = 0, Edges = 0, Face
  ⊞ spheres[2]    Sphere        Solid(Vertices = 0, Edges = 0, Face
  ⊞ spheres[3]    Sphere        Solid(Vertices = 0, Edges = 0, Face
  ⊞ spheres[4]    Sphere        Solid(Vertices = 0, Edges = 0, Face
```

Fig. 7: Typical DesignScript programming environment.

[6] `http://www.autodesk.com/products`

The programming language is presented as an associative language. The variables are abstract types that can represent numeric values or geometric entities. These variables are maintained in a graph of dependencies. When a change in a variable occurs it forces the re-evaluation of the graph, as shown in Figure 8, consequently variables has always updated values. This feature is useful, specially in a modeling environment, because it provides continuous feedback to the designer as the model is being modified.



```
A = 2;
B = 2*A; //B = 4
A = 5;   //A = 5, B = 10
```

Fig. 8: Associative interpretation.

The DesignScript's programming environment provides a text editor, an interpreter, and a simple debugger. The language interpreter is invoked 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 mode 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 special tab, as shown in Figure 8. In case the variable represents a geometric model, the respective model will be highlighted in the CAD when the variable is selected. It creates a certain *traceability* between models in the CAD and code in the editor. In this way the user is able to correlate which model a variable corresponds to. However the inverse, starting 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 designer 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 of being presented as pedagogic, has some drawbacks. It does not carry any strong 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.

Monkey[7] is a programming environment designed to support GD. Like DesignScript, Monkey is used to edit, debug and interpreter scripts. However, Monkey

[7] `http://wiki.mcneel.com/developer/monkeyforrhino4`

uses RhinoScript as its programming language and Rhinoceros3D[8] (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 casts variables into inappropriate types. Therefore, it creates errors difficult to find, specially for people which 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, does not provide any well designed feature which helps beginners to start with programming. The provided features are based on general-purpose systems, instead of being tailored for GD.

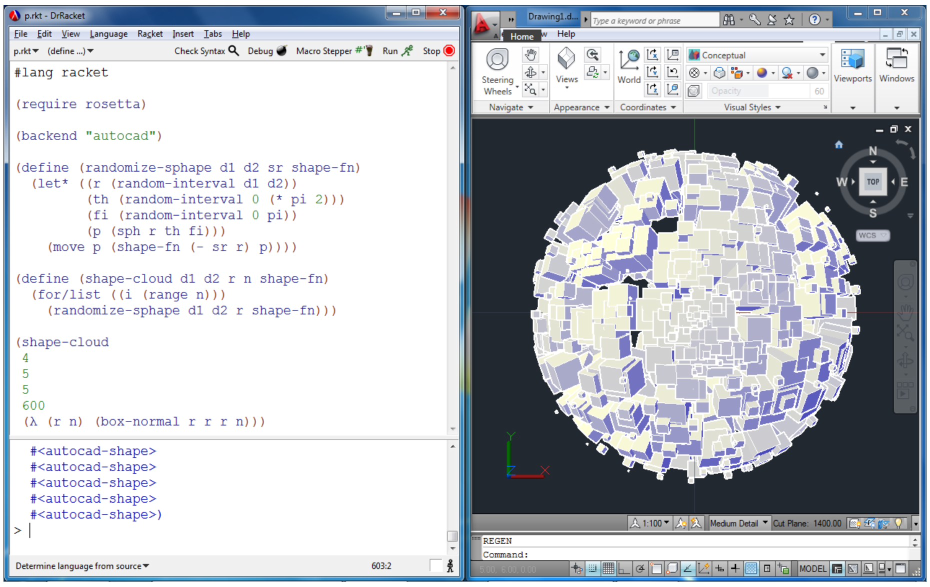
Rosetta [17] is a programming environment designed to support GD that is based on DrRacket [13]. Like Monkey, Rosetta provides its own environment detached from the CAD. Rosetta is a step forward from the previous systems, because it solves the portability problem among CAD tools. In Rosetta a GD program can be written in various programming languages (frontends) and the geometric models can be rendered by various CADs (backends). As a result, designers are free to write their programs in their preferred frontend which, upon execution, will generate the same geometry for the various backends. In Figure 9, a program is written in Racket and its execution produces geometry for AutoCAD.

Rosetta has been used to teach programming in architecture courses. Tailored to this end, Rosetta uses DrRacketas its own programming environment. The DrRacket environment serves a number of functions, but the most important is that the student can start immediately to learn programming. For instance, the environment is set up with just three lines of code. As shown in Figure 9, the `#lang` specifies the frontend language, the `require` imports Rosetta's primitives and finally the `backend` names a possible backend.

The Racket language is also an advantage of Rosetta's environment, because it encourages the use of the mathematical paradigm for writing algorithms. In this way, students that learn simple programming techniques, such as recursion, are able to 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,

[8] `https://www.rhino3d.com`



```
#lang racket

(require rosetta)

(backend "autocad")

(define (randomize-sphape d1 d2 sr shape-fn)
  (let* ((r (random-interval d1 d2))
         (th (random-interval 0 (* pi 2)))
         (fi (random-interval 0 pi))
         (p (sph r th fi)))
    (move p (shape-fn (- sr r) p))))

(define (shape-cloud d1 d2 r n shape-fn)
  (for/list ((i (range n)))
    (randomize-sphape d1 d2 r shape-fn)))

(shape-cloud
 4
 5
 5
 600
 (λ (r n) (box-normal r r r n)))
```

```
  #<autocad-shape>
  #<autocad-shape>
  #<autocad-shape>
  #<autocad-shape>
  #<autocad-shape>)
> |
```

```
REGEN
Command:
```

Fig. 9: Rosetta programming environment.

selecting the model 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 generated. 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.

Grasshopper[9] 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 [27]. For example, the boxes in Figure 10 are components which receive the input (left ports) perform some operations and return the output (right port). The components are linked to other components establishing a *dataflow* paradigm where the input of a component is the output of another.

[9] http://www.grasshopper3d.com/

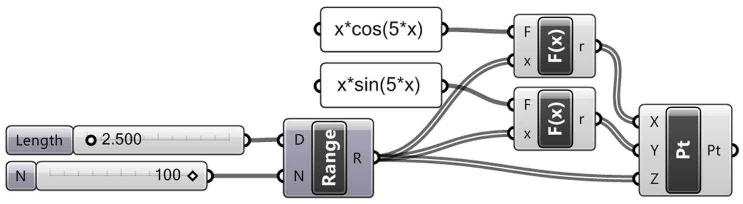


Fig. 10: A program in Grasshopper that computes the 3D coordinates of a conical spiral. Each time the left sliders are dragged a new coordinate is calculated.

Like Monkey, Grasshopper is implemented as a *plugin* for Rhino[8]. 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 are connected 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 better understand a program by figuring out the roles of each component.
– *Show comparisons between models.* Grasshopper provides a special component that, when connected at 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 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 huge 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 such backend.

Moreover, the traceability among components is just in one direction. From the designer perspective, it would be more useful start form 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.

Dynamo[10] 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, as shown in Figure 11, which provides quick access to the primitives of the language, such as the components and widgets. This feature encourages designers to explore the available components and try new components.

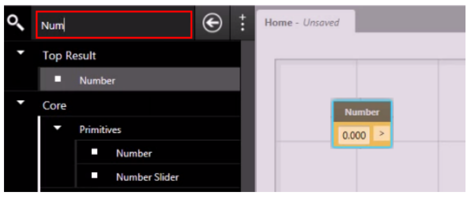
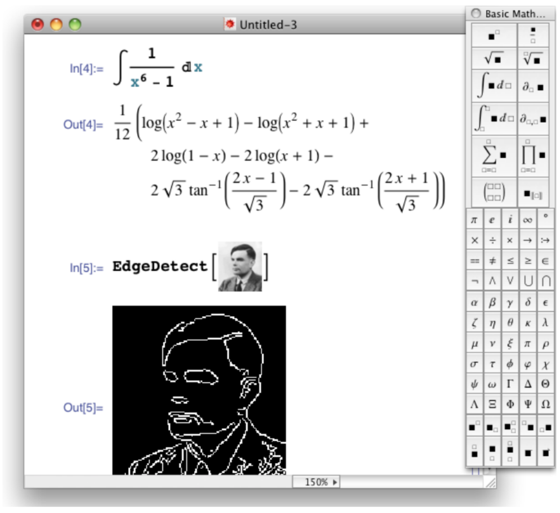


Fig. 11: Dynamo search tab. Searching for a component, highlighted in red.

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 UI elements are attractive for beginners. However as the visual programs become large and complex it requires more time to understand, maintain, and adapt to new requirements, than the textual programs as showed in [28]. 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.

Mathematica [29] is a language and environment built to support scientific calculation. It is widely used in the scientific community, specially by students, because it represents programs using a short and clear artificial language. This language supports not just linear textual input, but also two-dimensional input, like traditional mathematical notation.

The core concepts of Mathematica are based in the paradigm initiated by Turing's work [30]. In this paradigm mathematical processes are systematized as computations. For example, in a typical interaction, the user types a mathematical expression in the Mathematica environment (i.e. notebook), then this expression is evaluated, as shown in Figure 12.



In[4]:= $\int \frac{1}{x^6 - 1}\, dx$

Out[4]= $\frac{1}{12}\left(\log(x^2 - x + 1) - \log(x^2 + x + 1) + 2\log(1 - x) - 2\log(x + 1) - 2\sqrt{3}\tan^{-1}\left(\frac{2x - 1}{\sqrt{3}}\right) - 2\sqrt{3}\tan^{-1}\left(\frac{2x + 1}{\sqrt{3}}\right)\right)$

In[5]:= `EdgeDetect[ ]`

Out[5]=

Fig. 12: Mathematica notebook.

A relevant aspect of Mathematica's notebook is the immediacy that users get a response. Unlike a typical pro-

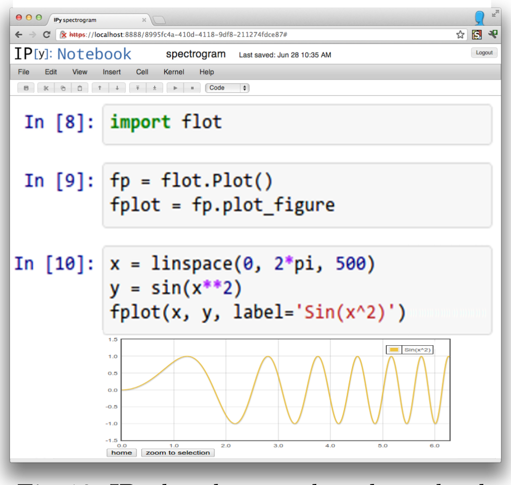
[10] `http://dynamobim.com/`

gram that must be executed
explicitly to get a feedback of an action, in this notebook expressions are evaluated as soon as they are typed. It works like a read-eval-print loop, however it has enhanced mechanisms to present data meaningfully.

The Mathematica features are well designed to present data in a human readable form. It would be useful for an external programming languages, if it could take advantage of these features. Unfortunately, Mathematica is closed for this end.

IPython [31] is a programming environment built to support scientific calculation. Unlike Mathematica [29], IPython is an open platform for extensions, it allows external programming languages (frontends) to use its features which includes an interactive shell, and a browser-based notebook with support for code, text, mathematical expressions, plots, and other rich media.

Like Mathematica, IPython has a notebook where users can try out expressions and immediately see its result, as shown in Figure 13, however this notebook is in a web format. IPython's architecture is a typical client-server, where the frontend is the client (i.e. the notebook), and the server is a language kernel (i.e. the programming language which users interact with). The communication between client and server, is through a strict protocol that the language kernel must implement.



```
In [8]: import flot

In [9]: fp = flot.Plot()
        fplot = fp.plot_figure

In [10]: x = linspace(0, 2*pi, 500)
         y = sin(x**2)
         fplot(x, y, label='Sin(x^2)')
```

Fig. 13: IPython browser-based notebook.

IPython provides a base layer for new programming environments, by exposing the major components of its architecture, consequently its features are available from other systems. For example, IJulia uses IPython interface with Julia language.

MathCAD[11] is a programming environment and language built to support scientific calculation. Like Mathematica, MathCAD aims to present information into a human readable form. However, it generates live calculations with graphical plots, text and images into a single document. This document is the program environment as well as the final product.

The mathematical expressions defined in the MathCAD document act as an associative language. So, when an expression is changed its value is propagated

[11] `http://www.ptc.com/product/mathcad`

through the document. This mechanism is the base of the interactiveness. However, it represents a barrier for program recomposition, because new expressions can change the previous ones. That means that variables has a global scope, the MathCAD document, if in somewhere a variable is changed it will be changed in every occurrence in the document.

3.4 Summary

Table 1 shows the presented systems based on their major design influences. The table is also intended to address the following questions:

Type[(1)]	**System**	**Main feature**[(2)]	**Support to understand programs**[(3)]	**Representation of code**[(4)]
GS	Eclipse NetBeans IntelliJ MVS	support software development life cycle	debugger	
	Xcode LightTable	observable programming	live execution feedback	
	LOGO SmallTalk	understandable language	physical interpretation	text
TS	Processing Fluxus	visual context	instant visualization	
	DrRacket	gradual learning	debugger; stepper	
	PythonTutor YinYang	show program flow	navigate through the program execution	
	DesignScript Monkey Rosetta	support generative design methods	debugger	
ES	Grasshopper Dynamo	alternative way to expressing programs	dataflow paradigm	graphical components
	Mathematica IPython MathCAD	support scientific calculation	present data meaningfully	mathematical forms

Table 1: System attributes.

(1) *What is the purpose of the system?* We categorized three main purposes for a system. General-purpose system (GS) designed for building complex software for the industry; Teaching system (TS) designed to help people learn how to program; Empowering system (ES) designed to help people build things that are tailored to their own needs.

(2) *How does the system support its purpose?* We identified the following strategies: (i) support software development life cycle, (ii) turn programming in something more observable, (iii) create an understandable language, (iv) combine textual programming with a visual context, (v) support gradual learning in a single environment, (vi) show the program flow, (vii) support generative design methods, (viii) find alternative ways for to express programs, and (ix) support scientific calculation.

(3) *Does the programming environment provide additional support to enable users to better understand the behavior of their programs?* Environments in our study used several techniques to help users understand the behavior of their programs. These included (i) a debugger which helps to find bugs in the program, (ii) an enhanced debugger which provides live execution feedback, (iii) languages with strong metaphor allowing physical interpretation, (iv) instant visualization of models, (v) navigation through the program's execution (vi) assembling components in a dataflow paradigm and (vii) present data adequately.

(4) *How does code look in the programming environment or language?* The systems in our study represent programs using text, users can type, graphical components, users can manipulate, and mathematical forms users can fill in.

To conclude, in the surveyed systems the common 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 for simple programs it is easier to read, but it becomes incomprehensible as the program grows.

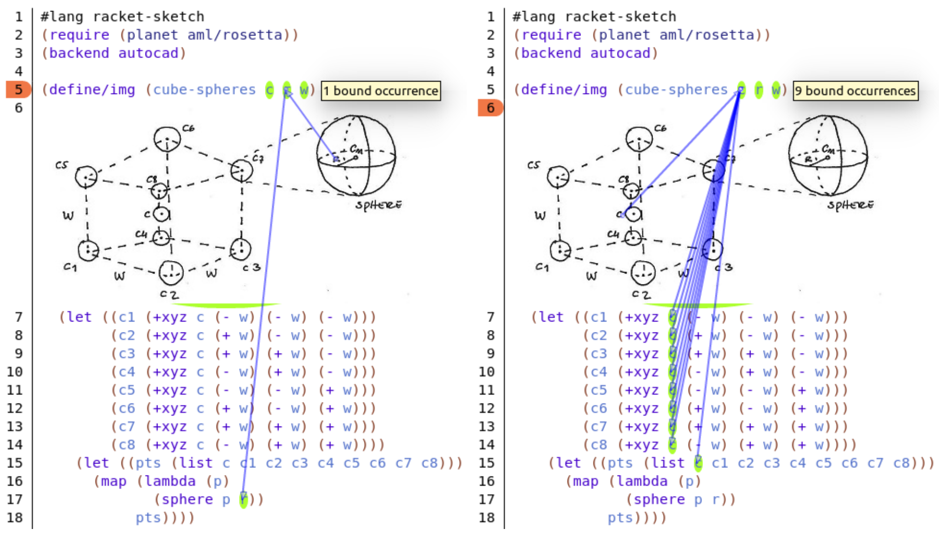
4 Architecture

The problem addressed in this thesis is to design and implement an interactive programming environment for generative design that covers the needs of both beginners and advanced users. Our approach suggests two interactive tools: (1) *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 program comprehension. The next section shows how these features will work.

4.1 Experimental results

An initial prototype was devised to show how code and images can be correlated, as shown in Figure 14. In this prototype, the meaning of the function and its parameters are transparent, because users can move the mouse over an identifier to figure out its meaning. For example in Figure 14a, there are two arrows pointing to the identifier `r`, when we look at the image is easy to see that it represents the radius of the sphere, while the other arrow refers to the location where it is used, i.e. in the `sphere` function. Especially in this example, the sketch illustrates exactly the function output. The function uses a primitive of Rosetta [17] (the `sphere` function) which creates a sphere, given a 3D point and a radius, in the selected *backend*.

Using the immediate feedback tool, programmers can test their ideas by quickly experiment them, as the prototype shown in Figure 15 which supports this process. In this prototype, the function defined in Figure 14, i.e.



```
#lang racket-sketch
(require (planet aml/rosetta))
(backend autocad)

(define/img (cube-spheres c r w) 1 bound occurrence

  (let ((c1 (+xyz c (- w) (- w) (- w)))
        (c2 (+xyz c (+ w) (- w) (- w)))
        (c3 (+xyz c (+ w) (+ w) (- w)))
        (c4 (+xyz c (- w) (+ w) (- w)))
        (c5 (+xyz c (- w) (- w) (+ w)))
        (c6 (+xyz c (+ w) (- w) (+ w)))
        (c7 (+xyz c (+ w) (+ w) (+ w)))
        (c8 (+xyz c (- w) (+ w) (+ w))))
    (let ((pts (list c c1 c2 c3 c4 c5 c6 c7 c8)))
      (map (lambda (p)
             (sphere p r))
           pts))))
```

(a) Searching for "r" meaning.

```
#lang racket-sketch
(require (planet aml/rosetta))
(backend autocad)

(define/img (cube-spheres c r w) 9 bound occurrences

  (let ((c1 (+xyz c (- w) (- w) (- w)))
        (c2 (+xyz c (+ w) (- w) (- w)))
        (c3 (+xyz c (+ w) (+ w) (- w)))
        (c4 (+xyz c (- w) (+ w) (- w)))
        (c5 (+xyz c (- w) (- w) (+ w)))
        (c6 (+xyz c (+ w) (- w) (+ w)))
        (c7 (+xyz c (+ w) (+ w) (+ w)))
        (c8 (+xyz c (- w) (+ w) (+ w))))
    (let ((pts (list c c1 c2 c3 c4 c5 c6 c7 c8)))
      (map (lambda (p)
             (sphere p r))
           pts))))
```

(b) Searching for "c" meaning.

Fig. 14: Contextualizing the code with image.

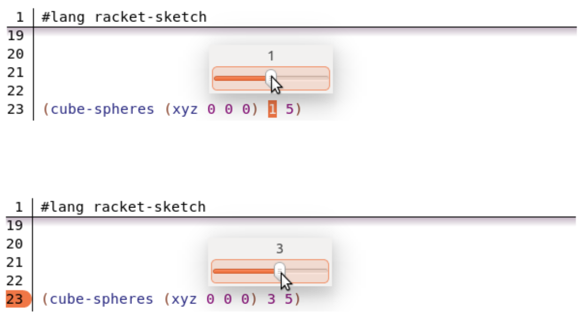
`cube-spheres`, is being interactively tested. Each change in the slider causes an execution of the program with the new slider value, particularly in this example, it will generate a new geometric model (see Figure 15b). As a result, users can confirm that the parameter `r` is, indeed, the radius of the sphere and it also eliminates the cycle edit-compile-run, allowing fast visualization of changes.

These features will be built on top of DrRacket [13]. In the following sections, we will present relevant properties of DrRacket that justify its choice as the basis of this thesis, as well as the proposed architecture to extend the DrRacket environment.

4.2 DrRacket Properties

Like DrRacket, our solution initially will target students, but, eventually, it would be used by anyone who wants to document the code with images or develops a program interactively. Therefore, to implement our solution we choose DrRacket, because it is built in the same principle we search for and has some key qualities:

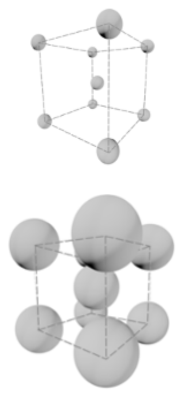
- **Pedagogic.** DrRacket is a popular environment used in introductory courses for programming languages. The environment is designed to guide the student by catching typical mistakes and explaining them in terms that students understand. It is also useful for professional programmers, due to its sophisticated programming tools, such as the static debugger, and its advanced language features, such as *units* and *mixins*.
- **Sophisticated editor.** DrRacket fully integrates a graphics-enriched editor which supports, in addition to plain text, elements such as images and boxes



```
#lang racket-sketch

(cube-spheres (xyz 0 0 0) 1 5)
```

```
#lang racket-sketch

(cube-spheres (xyz 0 0 0) 3 5)
```

(a) Changing the "r" parameter.



(b) The generated models, rendered by AutoCAD.

Fig. 15: Interacting with the `cube-spheres` function.

(with comments, Racket code or XML code). DrRacket also displays these elements appropriately in its read-eval-print loop.

– **Extensible.** The main tools of the DrRacket environment are implemented using the same API that is available for extension. For example, the debugger, the syntax checker and the stepper, despite providing different functionalities, are implemented on top of the same API.

Moreover, DrRacket helps programmers to understand the syntactic and lexical structure of their programs. DrRacket provides a syntax checker that annotates a syntactically correct program in five categories: the primitives, keywords, bound variables, free variables, and constants. When the programmer moves the mouse over an annotated identifier, the syntax checker displays arrows that point from bound identifiers to their binding occurrence, and vice-versa (see Figure 14). However, the syntax checker ignores the category of comments, including its visual elements such as the images, as a result these elements are uncorrelated with the program's structures and behavior.

In the next section, we propose an architecture which aims to address the above issue as well as proposes a solution for the immediate feedback tool.

4.3 Proposed Architecture

Figure 16 presents a publish-subscribe view of the proposed features. 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). From this event bus we subscribe only the UI events which are relevant to our system, defining which components will handle them. It is done at load time when the

event manager reads the *plugin* configuration file (`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. The client-server interaction is only needed to support the correlation between images and code. We assume that the images, inserted in the editor, will be associate to a single function and will contain the same identifiers defined by its associated function parameters (as shown the handmade sketch in Figure 14). Then, the manuscript symbols present in the image (e.g. the parameters "c","r", and "w"), will be parsed using an optical character recognition (OCR) engine. This engine usually gets an image and returns a text file containing a symbol table with the parsed symbols and its respective coordinates. We expect the OCR engine to act as an external service that identifies those symbols, thereby in our architecture the `symbol identifier` component calls this service to handle the recognition of symbols in the image.

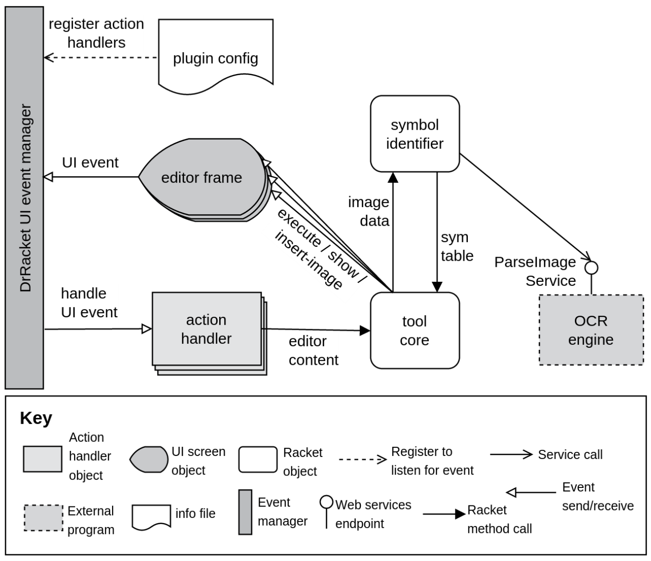


Fig. 16: Diagram for a publish subscribe view of the proposed architecture.

The `tool core` component, in Figure 16, will receive, at least, three kinds of DrRacket events. For each of these events, we will change the programming environment based on the following desired action:

– `on-change:` when DrRacket detects that the editor has been modified, it sends the contents of the editor over to action handlers. The action handler, in this case, it the `online expansion handler` where the code is expanded. *Desired action*: sends an `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 event provides a way to add arbitrary graphics to the editor display. *Desired action*: sends a `show` event to the editor frame, to display a slider widget when the user moves 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. *Desired action*: sends an `insert-image` event to the editor frame, before to get the image and send it to the OCR engine to recognize its symbols and respective coordinates (x, y). Then, it returns a subclass of image snip, containing this extra information.

Finally, to correlate the image-snip, created above, with the function parameters we will use a syntactic transformer, i.e. macro. The macro will add a new syntax form into the language grammar, allowing an image to be used as a comment. Very similar to Lisper's comment, the macro will add a new rule in the grammar, where an image is between a function declaration and a function body (see the macro `define/img` in figure 14). The image will be ignored, but in background, our macro expansion will add into the function body new occurrences of the function parameters. As a result, the DrRacket syntax checker will mark these free variables and will be able to recognize bound occurrences and point to them inside an image.

5 Evaluation

The evaluation of the proposed architecture will be performed experimentally, building a prototype. The prototype will serve to test the proposed ideas and to evaluate them. To evaluate the prototype, we will use the Rosetta [17] generative design tool as a case study. As Rosetta is used by architects, and designer, we will receive real feedback from the target users. In this way, we can evaluate if our programming environment helps their target users to design programs.

Furthermore, to evaluate our proposal we plan to use the following evaluation metrics.

- **Correctness.** To assess the quality of our system we plan to test, individually, each proposed feature with a specific test case scenario, for example using the slider widget to explore the result of a parametric function and inserting different kinds of images and check if the image is well correlated with the function parameters.
- **Security.** Among others qualities, security is an important concern in a live environment where the code is executed instantly. In our case, the code is executed locally, however while the users are using the live code mode they can create dangerous constructs such as `eval`, `exec` and `file I/O` which can damage the operating system. On the other hand, in this mode it is possible to block the environment with a simple "while true" expression. To

avoid these problems, we plan to implement sandboxing, similar to Python-Tutor [14], and design specific tests to test this feature.

- **Performance.** The performance of our system should scale for the generative design programs. The Rosetta tool will give us different *backends* to test the performance of our interactive tool. In fact, to be an interactive tool, the response for an event should be quick (∼50ms). It imposes restrict requirements for the CADs tools, because these tools were designed for the speed of human operation, consequently they are the performance bottleneck. This issue forces us to establish a limit which this tool will be tested, thus we will compare this limit against other similar systems.
- **Comparison with other systems.** We can only claim that our solution is somehow better than the other, if we compare them. Therefore, we plan to compare our system with the existing programming environments in generative design, particularly the visual environments, such as Grasshopper. Between these systems, the performance limit, stated above, will be our reference of comparison.

6 Conclusions

Programs are rarely released with useful documentation. This negatively affects software development including the several areas where it is applied, particularly in generative design, where programs are becoming relatively complex. It is now important to develop good tools for program documentation and program comprehension.

Based on Learnable 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 high level. It also encourages the designer 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 correlates text with sketches, beyond the only way to get immediate feedback is by a stepwise debug which stops the entire program execution, disabling code to be edited.

Using the immediate feedback tool will be possible to edit a program without stopping its execution. 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. On the other hand, the code correlation tool requires that the symbols in the sketch to be strictly identified, i.e. get their coordinates in the image to, eventually, point to them. An OCR engine is designed for this purpose, however, until now, we had bad experiences in using the OCR to recognize handwritten symbols. If necessary, and as a proof-of-concept we will generate the OCR data manually.

A functional prototype of these tools was already implemented. We plan to improve this implementation and, if time permits, explore additional tools, studding possible forms of integrations in the proposed architecture. Table 2, in Appendix A, shows a more detailed plan.

More important than these tools are the underlying design principles that they represent. Understanding how these principles enable people to think is the initial step to evolve the way we build programs.

References

1. R.R. Hamming. *The Art of Doing Science and Engineering: Learning to Learn.* Taylor & Francis, 2003.
2. Frederick P. Brooks. No Silver Bullet – Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19, 1987.
3. Spencer Rugaber. Program comprehension. *Encyclopedia of Computer Science and Technology*, 35(20):341–368, 1995.
4. Ruven Brooks. Towards a theory of the cognitive processes in computer programming. *International Journal of Man-Machine Studies*, 9(6):737–751, 1977.
5. J. DesRivieres and J. Wiegand. Eclipse: A platform for integrating development tools. *IBM Systems Journal*, 43:371–383, 2004.
6. David Carlson. *Eclipse Distilled.* Addison-Wesley Reading, 2005.
7. Tim Boudreau, Jesse Glick, Simeon Greene, Vaughn Spurlin, and Jack J Woehr. *NetBeans: the definitive guide.* " O'Reilly Media, Inc.", 2002.
8. Heiko Böck. IntelliJ IDEA and the NetBeans Platform. In *The Definitive Guide to NetBeans Platform 7*, pages 431–437. Springer, 2011.
9. Sam Guckenheimer and Juan J Perez. *Software Engineering with Microsoft Visual Studio Team System (Microsoft. NET Development Series).* Addison-Wesley Professional, 2006.
10. Seymour Papert. *Mindstorms: Children, computers, and powerful ideas.* Basic Books, Inc., 1980.
11. Alan C. Kay. *The early history of Smalltalk*, volume 28. 1993.
12. Casey Reas and Ben Fry. Processing: Programming for the media arts. *AI and Society*, 20(4):526–538, 2006.
13. Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. Drscheme: A programming environment for scheme. *Journal of functional programming*, 12(02):159–182, 2002.
14. Philip 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*, SIGCSE '13, pages 579–584, New York, NY, USA, 2013. ACM.
15. Sean McDirmid. Usable live programming. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 53–62. ACM, 2013.
16. Robert Aish. Designscript: origins, explanation, illustration. In *Computational Design Modelling*, pages 1–8. Springer, 2012.
17. José Lopes and António Leitão. Portable generative design for cad applications. In *Proceedings of the 31st annual conference of the Association for Computer Aided Design in Architecture*, pages 196–203, 2011.
18. Jon McCormack, Alan Dorin, Troy Innocent, et al. Generative design: a paradigm for design research. *Proceedings of Futureground, Design Research Society, Melbourne*, 2004.

19. Ellen Yi-Luen Do and Mark D Gross. Thinking with diagrams in architectural design. In *Thinking with Diagrams*, pages 135–149. Springer, 2001.
20. Sergio Cozzetti B de Souza, Nicolas Anquetil, and Káthia 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*, pages 68–75. ACM, 2005.
21. Maria João C Sousa and Helena Mendes Moreira. A survey on the software maintenance process. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 265–274. IEEE, 1998.
22. Bret Victor. Learnable Programming – Designing a programming system for understanding programs. Retrieved from `http://worrydream.com/LearnableProgramming`, Jan. 2014.
23. Bret Victor. Inventing on principle. Invited talk at the Canadian University Software Engineering Conference (CUSEC), Jan. 2012.
24. Gail C Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the elipse ide? *Software, IEEE*, 23(4):76–83, 2006.
25. Bowen Alpern and Fred B Schneider. Defining liveness. *Information processing letters*, 21(4):181–185, 1985.
26. Steven L Tanimoto. A perspective on the evolution of live programming. In *Live Programming (LIVE), 2013 1st International Workshop on*, pages 31–34. IEEE, 2013.
27. Brad A Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing*, 1(1):97–123, 1990.
28. 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*, pages 549–557, 2011.
29. Stephen Wolfram. *Mathematica*. Addison-Wesley, 1991.
30. Stephen Wolfram. Wolfram research. *Inc., Champaign, Mathematic*, 3:375–381, 2003.
31. Fernando Pérez and Brian E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007.

A Planning

Tasks	**Details**	**Duration**
Improve the related work research	– Explore additional related systems	July (3 weeks)
Improve the sketch-correlation tool	– Research additional OCR engines – Implement client-server architecture – Test the tool implementation	July (1 week) August (4 weeks) September (1 week)
Improve the immediate feedback tool	– Study DrRacket API – Implement publish-subscribe architecture – Improve the slider mechanism – Test the tool implementation	September (2 weeks) October (4 weeks) November (2 weeks) November (1 weeks)
Implementation of the case study	– Design specific tests using GD programs – Implement the tests – Test the tools – Review the results with the target users	November (2 week) December (4 weeks) January (2 weeks) January (1 weeks)
Evaluation	– Evaluate the solution with the case study results – Perform a comparison with similar systems	February (1 week) February (3 weeks)
Thesis final dissertation writing	– Write thesis dissertation	March (4 weeks) April (2 weeks)
Reviews and Submission	– Dissertation review and deliver MSc dissertation	April - May (3 week)

Table 2: Planning schedule