Learning principles in program visualizations: a systematic literature review

Jeisson Hidalgo-Céspedes, Gabriela Marín-Raventós, Vladimir Lara-Villagrán
Centro de Investigaciones en Tecnologías de la Información y Comunicación (CITIC)
Universidad de Costa Rica
San José, Costa Rica
{jeisson.hidalgo, gabriela.marin, vladimir.lara}@ucr.ac.cr

***Abstract*—Program visualizations help students understand the runtime behavior of other programs. They are educational tools to complement lectures or replace inefficient static drawings. A recent survey found 46 program visualizations developed from 1979 to 2012 reported that their effectiveness is unclear. They also evaluated learner engagement strategies implemented by visualization systems, but other learning principles were not considered. Learning principles are potential key factors in the success of program visualization as learning tools.**

In this paper, we identified 16 principles that may contribute to the effectiveness of a learning tool based on Vygotsky's learning theory. We hypothesize that some of these principles could be supported by incorporating visual concrete allegories and gamification. We conducted a literature review to know if these principles are supported by existing solutions. We found six new systems between 2012 and 2015. Very few systems consider a learning theory as theoretical framework. Only two out the 16 learning principles are supported by existing visualizations. All systems use unconnected visual metaphors, two use concrete visual metaphors, and one implemented a gamification principle. We expect that using concrete visual allegories and gamification in future program visualizations will significantly improve their effectiveness.

Keywords—systematic literature review; program visualization; learning theory; constructivism; visual metaphor; visual allegory; gamification

I. INTRODUCTION

Programming is one of the most fundamental abilities that Computer Science students must develop [1]. Indeed, a correct understanding of how the machine executes code is mandatory to effectively program a computer [1]. Students develop this understanding through the methods, techniques, and materials used in programming courses [2]. The most used method worldwide to teach the runtime behavior of programs is the lecture [3], [4]. The most used technique in lectures to explain how programs run is a combination of verbal descriptions with abstract drawings of programs' memory distribution in some computer architecture. This technique was called **program memory tracing** by Hertz and Jump [5]. The most used materials to create program memory traces in lectures are static illustrations on blackboards or projected slides [3], [4], [6].

The effectiveness and efficiency of static materials to explain an inherently dynamic process have been questioned [2], and program visualizations have been suggested as a solution [1]. A **program visualization** is a software tool that uses images to represent the dynamic behavior of other programs while they are running in a computer [2], [7]. However, in order to learn programming, it is not mandatory to understand how computers run programs at low level (i.e., using architecture's instruction set). The machine operation can be abstracted using the high level concepts provided by the programming language. These programming concepts form a new machine, that is not made of circuits, but of notions, and as such, it was called **notional machine** [8]. The main goal of program visualizations is helping students understand how programs are run by notional machines.

A recent literature review about program visualizations found 46 software systems created from 1979 through 2012 [2]. That review reported that empirical evaluations of program visualizations showed a mixture of results that make it unfeasible to know whether these systems are effective, or not, as learning tools [2]. Other studies show that program visualizations are rarely used in teaching-learning environments [3], [4].

These results reveal an important problem. Students are required to develop a correct understanding of how computers run code in order to program effectively, but the widely used static instruction materials are inefficient to reach this goal. Moreover, the proposed solution, program visualizations, does not offer convincing results as learning tools. The motivation of this work is to contribute on how to build effective program visualizations as learning tools.

This paper proposes that program visualizations must be analyzed in the light of a learning theory to be effective as learning tools. Section II infers from a learning theory, 16 learning principles stated as software requirements that may contribute in the effectiveness of a learning tool. We hypothesize that some of these principles may be implemented using computing techniques such as concrete allegories and gamification. Then, the paper tries to answer the following research question: to what extent are these 16 learning principles supported by existing program visualizations? The question will be answered by a systematic literature review that extends the one reported in [2], and also evaluates visual metaphors and gamification support, in section III. Section IV presents the methodological details of our review. Section V reports and discusses the results of the systematic literature

review. Finally, section VI concludes and highlights the main contributions of this work: an updated list of program visualizations, the answer to our research question, and several new research lines to increase the effectiveness of program visualization as learning tools.

II. THEORETICAL BACKGROUND

In this section, we first discuss the definition used in this review to consider a tool as a program visualization or not. Then, we derive from a learning theory some guidelines or principles that may impact the effectiveness of software tools as learning tools. Finally, we focus on visual metaphors and gamification, because we hypothesize that these techniques may be used to implement some of the derived learning principles.

A. Program visualizations

Visualization is a representation using images of a non-optical phenomenon. Visualizations are created to help people build mental representations of abstract or complex phenomena, such as the geographical distribution of an epidemic in a map, or the average amplitude of several sound signal frequencies produced by a graphic equalizer. When the phenomenon being represented is software, it is called **software visualization**. Because software can be represented at several levels of detail, there are several types of software visualizations:

1. **Algorithm visualization** represents the software at high level of abstraction independently of the programming language. Usually this type of system is used to explain general algorithms such as quick-sort or binary-search. They are mainly targeted to advanced Computer Science students. [2], [9]
2. **Code visualization** represents static features of the source code of the software being studied, e.g., the interdependence of source files. Code visualizations are dependent of the programming language, and mainly targeted to a professional audience [2], [9].
3. **Program visualization** represents the behavior of the software being executed by a specific architecture. It is dependent on the programming language and the computer architecture. Program visualizations are mainly targeted to programming beginners [9].

There are other related concepts of interest. **Manual visualizations** are produced by humans, whereas **computer visualizations** are produced by software running on a computer. A **dynamic visualization** updates the images along the execution time, but a **static visualization** produces images that are not changed until the visualization is started over. An **interactive visualization** allows users to control some aspects of the visualization, such as selecting elements to show or hide, or setting execution parameters or input data. On the other hand, a **computer animation** is intended for passive viewers of the generated images.

A visualization only represents a subset of the phenomenon. A **specialized visualization** represents only a few features of the phenomenon, whereas a **generic visualization** tries to represent a comprehensive subset. For example, a *specialized program visualization* represents the runtime behavior of a few programming concepts, such as parameter passing or pointers [2]. Specialized visualizations have the advantage of centering attention on a topic, reducing distractions from other concepts [2]. A *generic program visualization* tries to represent the majority of concepts of a programming language [2]. Generic program visualizations have the advantage of providing a big picture of the notional machine, and usually they are able to run students' own programs [2].

In this paper we are interested in general program visualizations targeted to programming students or professors. In another words, in this paper, the term **program visualization** refers to software that generates optical images that help programming students understand the behavior of other programs being executed by a notional machine.

As stated by [2], there is uncertainty about the effectiveness of program visualizations as learning tools. In the next subsection, we analyze a learning theory in order to find principles that program visualization may implement in order to be effective as learning tools.

B. Learning principles

A learning theory may be used to identify potential key factors in the success of learning tools. There are four main learning theories: behaviorism, constructivism, cognitivism, and humanism [10]. We selected constructivism because it has been recommended for Computer Science [11]. There are two major types of constructivism: Piaget's cognitive or individual constructivism, and Vygotsky's social constructivism [12]. We chose the latter because it is not limited to the students' ontogenetic development, and because program visualizations can be represented as sociocultural tools [13].

Learning is an experiential process of neural restructuration that changes the meaning of something and can be verified behaviorally (adapted from [14] and [15]). A process is a series of actions to achieve a result or a series of changes that happen naturally. Sociocultural constructivism authors do not provide an explicit series of actions or changes to explain the learning process. We inferred from [16] seven learning stages as diagrammed in Fig 1. Stages will be explained ahead.

Sociocultural constructivism states that learning is a social-mediated process of association of concepts [12]. A new concept can only be learned if it can be associated with other concepts in the student's own knowledge [16] (center light gray zone in Fig 1). The zone of proximal development is the set of all concepts that a person can directly associate, and therefore, learn (dark gray zone in Fig 1). A mediator contributes in the learning process, by influencing the association of new concepts in the zone of proximal development of the learner with concepts already present in learner's knowledge [16].

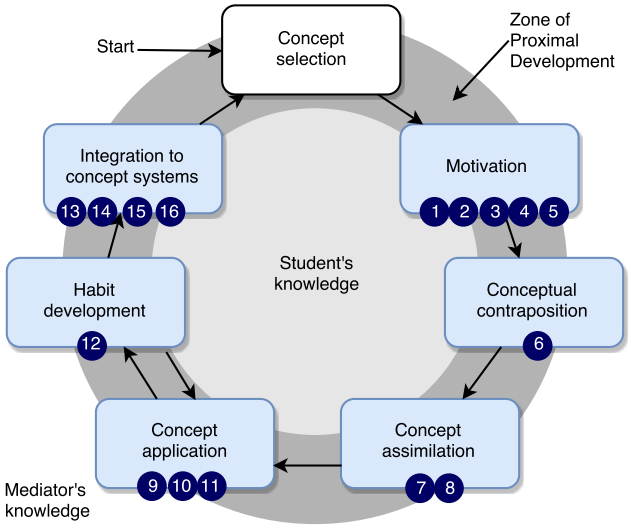


Fig 1. Stages of learning process and identified principles

The mediator is part of the context around the learner, and it is not limited exclusively to other people. Sociocultural tools, such as computers and software, can also mediate in the learning process [13]. We inferred from the mediator's role, 16 learning principles labeled within dark circles in Fig 1. We adapted these 16 principles as software requirements, and they are listed in TABLE I.

TABLE I. CONSTRUCTIVIST PRINCIPLES AS SOFTWARE REQUIREMENTS

#	**Requirement (the tool should…)**
1	Promote an active state of student's mind (e.g: interactivity).
2	Investigate what motivates students.
3	Organize concepts reflecting natural relationships.
4	Evaluate students' previous notions and emotions related to the concept being taught.
5	Arouse students' interest in the academic task.
6	Use the conceptual contraposition technique when students' previous notions do not allow for the construction of new ones.
7	Visualize the new concept associating it with students' previous notions.
8	Encourage students to carefully compare a new concept with previous ones, identifying similarities and differences.
9	Let students exercise the new concept. Exercises must vary in non-essential aspects, but preserve the fundamental principle of the new concept.
10	Give feed back on right or wrong actions, providing explanations when students fail.
11	Let students apply the new concepts to some situations or problems in daily or professional life. Problems should be more complex than exercises solved in principle 9.
12	Foster students to apply the new concept to several problems until they develop a habit.
13	Help students associate the new concept with other related concepts, building concept systems.
14	Focus in and evaluate the learning process rather than the final product before it becomes a habit.
15	Evaluate that the concept systems built by students are right and stable in time.
16	Foster students to apply the concept systems to new and complex situations.

Constructivists state that learning happens only in an active state of the mind [14], [16]. Therefore, program visualizations must not assume learners as passive observers. Conversely, program visualizations must be interactive, e.g., allowing users typing code, modifying it, controlling the execution, or using the visual elements to build or alter code (principle 1 in TABLE I). The previous literature review by Sorva, Karavirta, and Malmi evaluated this principle using a qualitative scale that they called *direct engagement* [2]. Notice this principle is not tied to a particular stage, but to the entire learning process.

Before starting the teaching of a particular concept, the mediator may investigate why students are involved in the educative task (principle 2 in TABLE I) [16]. For example, students may be motived to learn in order to be more competitive than others, earn money, or please parents. Any action executed by the mediator will be amplified or attenuated by the general motivation that the student has about the learning task [16].

It is recommended that the mediator organizes the concepts to be learned, trying to reflect a natural network of relations that are present in real life (principle 3 in TABLE I) [16]. A basic organization is the dependence relation. For example, the concept of pointer requires, at least, knowledge of variables, memory addresses, value of variables, data types, memory segments, and function calls.

Previous principles can be considered as preparation for the learning process. The first step is choosing the concept to be learned by students, based on their interest about a particular problem or any other reason. The mediator should be sure the selected concept is in the learners' zone of proximal development (principle 4 in TABLE I) [16]. For example, the mediator may help students check themselves that they satisfy the concept dependence network in principle 3. Because the new concept will be constructed by associating existing knowledge that students bring from their life experience, old emotions will revive and may affect their motivation towards the new notions [16]. The mediator may encourage students to evaluate their existing emotions about the new concept and change them in case of negativism (principle 4 in TABLE I).

Learning a new concept can be a fluent process if students are motivated. There are some techniques that a mediator may use in order to arouse students' motivation (principle 5 in TABLE I) [16]. For example, by providing context about the importance of mastering the concept or the risks of ignoring it at professional level, or by connecting the concept with the general students' motivation gathered in principle 2.

New concepts in the zone of proximal development are learned by associating them to existing concepts that were constructed previously, even before any formal education [16]. Sometimes, the students' existing concepts do not help to establish a natural association. In that case, the conceptual contraposition technique may be useful (principle 6 in TABLE I) [16]. The **conceptual contraposition** (known in Occident as **cognitive dissonance** [17]), consists in making students realize that their old abilities and notions are insufficient or contradictory to meet the objectives they are motived to achieve [16]. For example, challenging students to resolve a problem that require the new notions, but using their actual

knowledge, such as programming a swap function in C before introducing pointers. When they realize their methods or notions that they own are ineffective or insufficient, they will experience a cognitive uncertainty state, and they will be willing to reorganize old concepts and construct new ones (that is, a willingness to learn) [16]. This is the ideal situation to present new concepts.

When the students are requiring a new concept, the mediator can help them to find information about the new concept or just present it. The new concept must be explained or visualized using other concepts in the apprentices' knowledge (principle 7 in TABLE I) [16]. For example, the pointer concept may be visualized as an integer variable with some abilities to point to some other place in the memory.

When a concept is introduced, wrong associations may be constructed. For example, because all pointers are integer variables, a student may generalize that they can be used interchangeably. The mediator may encourage students to carefully compare the new concept against the associated ones, in order to list their similarities and differences (principle 8 in TABLE I) [16]. For example, both integer variables and pointers store integer values, but pointers are never negative, the value 0 has a different meaning for each one, and we cannot assign them directly.

The mediator may provide a series of short exercises that help students identify the properties of the new concept, the relations, similarities and differences with previous concepts, and their application to several situations (principle 9 in TABLE I) [16]. The task of solving several short exercises that keep the new-concept's fundamental principle and vary only its non-essential aspects (like its application), will help promote the temporary associations from short-term memory to long-term memory [16]. Students require immediate feedback from the social context, or mechanisms that allow them to self evaluate the validity of their notions, in order to progress in the learning process (principle 10 in TABLE I).

Once assimilated in abstract thinking, the new concept should be applied to practical situations, otherwise it will not add any meaningful change to students' life experience [16]. The mediator may provide real life problems that require the application of the new concept in order to be solved (principle 11 in TABLE I). For example, writing a reusable library function that calculates both roots of quadratic function, and returns them as parameters by reference.

Students should apply the new concept to solve different problems until they reach a habit (principle 12 in TABLE I). A habit is developed as consequence of the refinement of a method through its repeated application to several distinct situations sharing the same fundamental principle [16].

After a habit is developed and the concept is established in long-term memory, students should solve more comprehensive problems. These problems require the interaction of the new concept with other concepts learned previously (principle 13 in TABLE I). For example, students may solve problems that require pointers to functions, or pointers to pointers. Comprehensive problems are very important to construct concept systems. Vygotsky states that knowledge is not constructed by isolated concepts, but systems of associated concepts that reflect the relationships to objects and real life phenomena [16].

Evaluations made by the mediator or other students should be used to provide immediate feedback about three aspects of the learning process:

1. Their performance in the problem solving process rather than the final product or solution (principle 14 in TABLE I) [16].
2. Concept systems are correct and stable over time (principle 15 in TABLE I)
3. Students are able to apply concept systems to new situations (principle 16 in TABLE I) [16].

Before investigating the current support to the identified 16 learning principles by active program visualizations, we propose two techniques that may be used to implement some of these principles: visual allegories and gamification.

C. *Visual allegories*

Constructivism emphasizes that learning occurs by association of new concepts with previous knowledge, and it is reflected in principles 7, 8, and 13. We hypothesize that these three principles may be supported by program visualizations using visual allegories, a special type of metaphors.

A **metaphor** is a mapping of a subset of features between two concepts, that allows the representation of a concept, called target, using the other, called source (adapted from [18]–[20]). For example, in the metaphorical phrase "time is money", the target "time" is represented by the source "money" because a mapping is established between features, such as valuable and limited. The source can be elided, which is very common in Computer Science, where metaphors are used to name almost all concepts [20]. A few examples are: mouse, port, shopping cart, mailbox, file, window, tree, pointer, and memory leak [20]. Metaphors are important learning instruments, because they ease the association process of abstract Computer Science concepts with more colloquial concepts [20].

Metaphors are not constrained to the linguistic scope. Metaphors can be visual, audible, tangible, or a combination of these. Visual metaphors are particularly important in this research because program visualizations use visual metaphors to represent programming concepts. A **visual metaphor** maps a subset of common features between a target concept (e.g., programming concept) with an optical image as source [21].

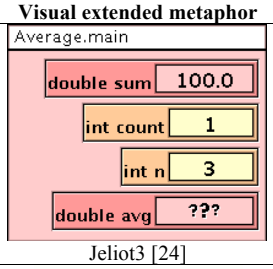
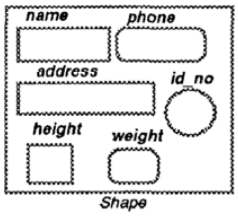
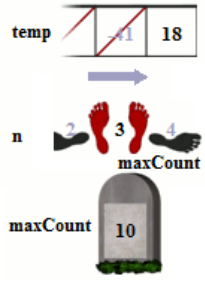
An **extended metaphor** is a set of interrelated metaphors by two explicit discourses: the source and target [22]. An **allegory** is an extended metaphor whose target discourse is elided [22]. Lakoff conceptualizes allegories as *conceptual metaphors* and one of his initial examples represents love as a journey: "*Look how far we've come. It's been a long, bumpy road […] The relationship isn't going anywhere. We're spinning our wheels…*" [18, p. 206]. Understanding an allegory requires a dual mental work: while the source discourse is explicitly perceived, the mind must construct the implicit target discourse. Allegories demand a coherent discourse between the

source metaphors in order to be understandable. Extended metaphors may have a disconnected discourse between source metaphors, because the target discourse is provided.

General program visualizations provide a visual metaphor for each notional machine's concept. These visual metaphors altogether constitute a **visual allegory** if the target discourse is elided because there exists a coherent and parallel discourse between the visual images and the target discourse. For example, in static program visualizations made by Waguespack (TABLE II), rounded rectangles represent integer values, and sharp rectangles real values, because the former can be stored in the latter without loss of information (decimals) [23]. If the source discourse between the used graphical elements is not coherent, the target discourse must be explicated textually, and therefore a **visual extended metaphor** is provided. For example, rectangles in Jeliot3 [24] are used to represent method calls, local variables, and their values; therefore, data types are included as part of the textual discourse (TABLE II).

The goal of program visualizations is to make clear how programs are run by notional machines [8]. Program visualizations designers must choose either abstract images (such as the geometric figures or arrows in first row of TABLE II) or concrete images (such as the objects in second row of TABLE II) to represent programming concepts that are abstract by nature. An analysis of metaphors within a time frame of 300 years, found a tendency of associating abstract concepts with concrete ones, as an attempt to make the abstract world understandable [25]. This tendency is also supported by the sociocultural learning theory, because programming notions must be associated to knowledge that learners have constructed previously in their life experience. We want also know if this tendency is happening in program visualizations.

TABLE II. VISUAL METAPHORS IN PROGRAM VISUALIZATIONS

	Visual extended metaphor	Visual allegory
Abstract	 Jeliot3 [24]	 Waguespack [23]
Concrete	 PlanAni (adapt. from [26])	No visualizations found

D. Gamification

Gamification is a careful and considered application of game elements to other activities with the purpose of increasing people's motivation and engagement, and quality of their results (adapted from [27]). Kapp identifies 12 game elements that may be used in the gamification of serious activities: abstraction, goals, rules, conflict and competition/ cooperation, time, rewards, feedback, levels, storytelling, curve of interest, aesthetics, and replay [27]. He also provides theoretical support and scientific evidence of effectiveness of gamification in learning activities [27].

We hypothesize that program visualizations may support several learning principles by applying game elements. For example, programming concepts may be organized (principle 3) as worlds and levels, and "boss" levels may evaluate the application of concept systems.

III. PREVIOUS WORK

There are some non-comprehensive literature reviews about algorithm visualizations, code visualizations, or software visualizations in general, such as [28] (1990), [29] (1993), [30] (2002), [31] (2009), and [32] (2013). Sorva, Karavirta and Malmi claim their study be the first comprehensive literature review about program visualizations [2] (2013). That study found 46 systems between 1979 and 2012, inclusive. From 2013 to April 2016, no other literature review about program visualizations was found. Therefore, [2] is the most relevant previous work, and this paper updates the program visualization list from 2013 to first quarter 2016.

There is not a consistent nomenclature related to software visualization terms in literature reviews, papers about a specific tool, and even taxonomies. For example, [2] defines the term *program visualization* including *code visualizations* and *visual programming*, but their review's selection criteria matches the definition used in this paper.

In [2] two variables related to motivation, engagement and interaction, were evaluated for each system found. In this paper, our objective is to evaluate the use of the learning principles identified in TABLE I, the type of visual metaphors, and gamification elements.

IV. METHODOLOGY

In order to estimate to what extent learning principles are supported by existing program visualizations, we conducted a systematic literature review, following the methodology recommended in [33]. Because the terminology to refer program visualizations varies according to authors, some related expressions where used in the search string, and they were also limited to the educative (learning or teaching) scope:

```
("program visualization" OR
"program visualisation" OR
"program animation" OR
"software visualization" OR
"software animation" OR
"visual debugger") AND
(learn* OR teach* OR educat*) AND
publication year on or after 2013
```

Papers were searched in four databases: ACM Digital Library, IEEE Xplore, Web of Science, and Scopus. First two databases are very relevant because they published most of our

control papers. Last two databases were considered because they index the most prestigious journals in Computer Science.

All results were reviewed by one researcher. In order to decide if each paper was included or excluded, the criteria in TABLE III was checked in title, keywords and abstract. If these fields were insufficient, the full paper was read. The references section of selected papers were also analyzed in order to detect potential missing papers (snowball process).

Each included paper was read in order to detect new program visualizations from 2013 to early 2016. General details of new visualizations were retrieved, such as, their notional machine, supported platforms, active or available status, type of visual metaphor, and gamification support. These details were also retrieved for active or available program visualizations listed in [2]. A system is considered *active* in this paper if there is recent activity, such as new publications, software releases, or changes in version control systems. A system is *available* if the software can be downloaded and tested.

For each available program visualization the learning principles in TABLE I were subjectively evaluated, assigning a real number between 0 and 1, where 0 means no support at all, and 1 the principle is entirely supported. The averages of these values are used to answer our research question.

TABLE III. SELECTION CRITERIA

Inclusion	11. Paper presents a new program visualization (from 2013 to 2016) 12. Paper uses a program visualization introduced in other papers 13. Paper relates several program visualizations (e.g., a review) 14. Paper discusses program visualizations in general, does not refer particular ones 15. Tool is a specialized program visualization 16. Tool is not a program visualization, but uses gamification or visual allegories
Exclusion	21. Paper is not a full paper (abstract-only, poster, workshop) 22. Paper is not written in English 23. Full text was not available (in snowball process) 24. Paper is duplicated (i.e. present in another database) 25. Paper is already included in previous literature review [2] 26. Tool is an algorithm visualization 27. Tool is a code visualization 28. Tool is not a "program visualization" as defined in this paper 29. Paper is not related to programming education

V. RESULTS AND DISCUSSION

The query string retrieved 169 results from the four databases. We discarded 44 duplicated results. After applying the selection criteria to the remaining 125 results, 36 papers between 2013 and first quarter of 2016 matched the selection criteria for reviewing.

In the period 2013-2016, seven new program visualizations were found, and they are listed in dark background at the bottom of TABLE IV: Jeliot ConAn [34], [35]; BlueJ Novis [6], [36], [37]; SeeC [38], [39]; Virtual-C IDE [40]; PROVIT [41]; FM Visualization [42]; and JavelinaCode [43]. TABLE IV also includes the active or available program visualizations that were initially reviewed in [2]. Program visualizations, in rows, are ordered by date of appearance (column "From"). From column "Notional M", we infer that Java and C notional machines are the most targeted ones by new developments.

From column "Metaphor" in TABLE IV, is clear that all active or available program visualizations use disconnected visual metaphors to represent programming concepts. In another words, visual allegories are not supported at all. According to column "Visual M.", all program visualizations use abstract images to represent abstract programming concepts, except two visualizations: *PlanAni* [26] (TABLE II) and *Metaphor-based OO visualizer* [44]. Both systems use concrete images, the former to represent roles of variables, and the latter to object-oriented concepts.

Column "Gam" shows that only one active or available program visualization supports gamification elements. This tool will be discussed in next section. Because only active or available program visualizations are listed in TABLE IV, at least one of the columns "Active" or "Avail" must have a "Yes" value. New tools (in dark background) were considered active because their recent publications. Inactive tools from [2] were not replicated in TABLE IV.

A. Constructivist principles support

In order to have an approximation to what extent constructivist learning principles are supported by program visualizations, a subjective evaluation is presented in TABLE V. Because a runnable copy of the software was required for evaluating purposes, only available program visualizations were considered.

The program visualization's support for each learning principle (in TABLE I) was graded with a real value between 0 and 1, where 1 indicates there is an evident explicit support, and 0 absence of support. The color of each cell reflects its value in a range from red (0) to green (1).

The row "Average" informs to what extent, in average, each constructivist principle is supported by the available program visualizations. In general, the support for all principles is very low, except for principles 1 and 10.

Constructivist principle 1 is supported because the majority of general program visualizations are interactive. Therefore, they require users to type code, provide execution parameters, or to control the execution of the visualization. *UUhistle*, *ViLLE* and *Jeliot ConAn* have the highest support to constructivist principles. *UUhistle* was designed with the activity constructivist principle in mind: the user is invited to perform the animations instead of the computer [45]. The constructivist principles implemented by *ViLLE* are related mainly to its question system, which encourages students to actively study the code and its visualization in order to answer correctly.

TABLE IV. LIST OF NEW OR AVAILABLE PROGRAM VISUALIZATIONS

#	Name	From	Notional M.	Platform	Metaphor	Visual M.	Gam	Active	Avail	Web site
1	GRASP / jGRASP	1996	Java	Java	Extended	Abstract	No	Yes	Yes	http://www.jgrasp.org/
2	The Teaching Machine	2000	C++, Java	Web	Extended	Abstract	No	Yes	Yes	http://www.theteachingmachine.org/
3	PlanAni	2002	C, Java, Pascal	Win	Extended	Concrete	No	Yes	Yes	http://www.cs.uef.fi/~saja/var_roles/planani/index.html
4	JIVE	2002	Java	Eclipse	Extended	Abstract	No	Yes	Yes	http://www.cse.buffalo.edu/jive/
5	Jeliot 2000 / Jeliot 3	2003	Java	Java	Extended	Abstract	No	Yes	Yes	http://cs.joensuu.fi/jeliot/
6	VIP	2005	C++	Java	Extended	Abstract	No	Yes	Yes	http://www.cs.tut.fi/~vip/en/
7	ViLLE	2005	Java/C++/...	Java, Web	Extended	Abstract	No	Yes	Yes	https://ville.cs.utu.fi/old/
8	Metaphor-based OO visualizer	2007	Pascal	Web (Flash)	Extended	Concrete	No	No	Yes	http://www.cs.uef.fi/~saja/oo_metaphors/index.html
9	UUhistle	2009	Python	Java	Extended	Abstract	No	Yes	Yes	http://www.uuhistle.org/
10	Online Python Tutor	2010	Python/...	Web	Extended	Abstract	No	Yes	Yes	http://www.pythontutor.com/
11	Jeliot ConAn	2013	Java	Java	Extended	Abstract	Yes	Yes	Yes	https://cs.joensuu.fi/jeliot/
12	BlueJ Novis	2013	Java	Win, Mac, Android	Extended	Abstract	No	Yes	Yes	http://bluej.org/novis.zip
13	SeeC	2013	C	Win, Lin, Mac	Extended	Abstract	No	Yes	Yes	http://seec-team.github.io/seec/
14	Virtual-C IDE	2014	C	Win, Mac, Android	Extended	Abstract	No	Yes	Yes	https://sites.google.com/site/virtualcide/
15	PROVIT	2014	C	Web	Extended	Abstract	No	Yes	No	http://provit.u-aizu.ac.jp/
16	FM Visualization	2014	C	Win, Lin, Mac	Extended	Abstract	No	Yes	No	
17	JavelinaCode	2015	Java	Web	Extended	Abstract	No	Yes	No	http://yangtamuk.weebly.com/overview.html

TABLE V. SUBJECTIVE EVALUATION OF LEARNING PRINCIPLES SUPPORT BY PROGRAM VISUALIZATIONS

#	Name	CP01	CP02	CP03	CP04	CP05	CP06	CP07	CP08	CP09	CP10	CP11	CP12	CP13	CP14	CP15	CP16	Avera	Average
1	GRASP / jGRASP	0,5	0	0	0	0	0	0,1	0	0	0,3	0	0	0	0	0	0	**0,06**	**5,6%**
2	The Teaching Machine	0,4	0	0	0	0	0	0,1	0	0	0,4	0	0	0	0	0	0	**0,06**	**5,6%**
3	PlanAni	0,6	0	0,1	0	0	0	0,6	0	0	0,7	0	0	0	0	0	0	**0,13**	**12,5%**
4	JIVE	0,8	0	0	0	0	0	0,1	0	0	0,6	0	0	0	0	0	0	**0,09**	**9,4%**
5	Jeliot 2000 / Jeliot 3	0,7	0	0	0	0	0	0,2	0	0	0,5	0	0	0	0	0	0	**0,09**	**8,8%**
6	VIP	0,4	0	0	0	0	0	0,1	0	0	0,6	0	0	0	0	0	0	**0,07**	**6,9%**
7	ViLLE	1	0	0,5	0	0,3	0,25	0,2	0,2	0	0,7	0	0	0,75	0	0,1	0	**0,25**	**25,0%**
8	Metaphor-based OO visualizer	0,2	0	0	0	0,1	0	0,6	0	0	0,2	0	0	0	0	0	0	**0,07**	**6,9%**
9	UUhistle	1	0	0,5	0	0,3	0,6	0,3	0,2	0	0,8	0	0,2	0,3	0,8	0	0	**0,31**	**31,3%**
10	Online Python Tutor	0,8	0	0,1	0	0	0	0,1	0	0	0,4	0	0	0	0	0	0	**0,09**	**8,8%**
11	Jeliot ConAn	1	0	0,1	0	0,75	0,5	0,2	0,1	0,1	0,5	0,1	0,1	0	0,1	0	0	**0,22**	**22,2%**
12	BlueJ Novis	0,6	0	0	0	0	0	0,2	0	0	0,5	0	0	0	0	0	0	**0,08**	**8,1%**
13	SeeC	0,25	0	0	0	0	0	0,1	0	0	0,7	0	0	0	0	0	0	**0,07**	**6,6%**
14	Virtual-C IDE	0,5	0	0	0	0	0	0,1	0	0	0,5	0	0	0	0	0	0	**0,07**	**6,9%**
	Average	**0,61**	**0,00**	**0,09**	**0,00**	**0,10**	**0,09**	**0,21**	**0,03**	**0,01**	**0,52**	**0,01**	**0,02**	**0,07**	**0,06**	**0,01**	**0,00**	**0,11**	**11,3%**

Jeliot ConAn is a variant of Jeliot 3 that creates conflictive animations (Fig 2). When a program is run in *Jeliot ConAn*, some steps will be automatically visualized incorrectly. The goal of conflictive animations is to arouse student's attention and intrigue [34] (constructivist principle 5). *Jeliot ConAn* is used to set up gamified environments. For example, in order to win, students in teams must detect as many faults as they can, in conflictive programs created by the other teams [34]. When a fault is found, the "Fault" button should be pressed in *Jeliot ConAn* interface (Fig 2).

Constructivist principle 10 encourages immediate feedback in the learning process. All program visualizations provide immediate visual information about the program state while it is running in the notional machine. However, a few tools provide verbal or high-level explanations when a program fails or does a very complex operation, which is a desirable feature (TABLE V).

There are six principles that are slightly supported by a few visualizations. Principle 3 (organize concepts reflecting natural relationships) is mainly supported by visualizations that allow instructors to organize exercises hierarchically: *ViLLE* and *UUhistle*. However, relations between programming concepts resemble a network structure.

Principle 5, about arousing students' interest in the academic task, is mainly supported by *Jeliot ConAn* because it is designed for gamified activities. *ViLLE* and *UUhistle* allow instructors to write an introductory text to each programming exercise. These texts may be used to provide students with a context about the importance of the programming concepts involved in the exercise.

Principle 6, about intriguing students using the conceptual contraposition technique, is partially supported by three visualizations. Students may experience intrigue when they observe an anomalous animation in *Jeliot ConAn*, when they do not know how to animate a step in *UUhistle*, or when they cannot predict a result asked by a question in *ViLLE*.

Principle 7, about associating new concepts with previous notions in students' mind, is mainly supported by the two visualizations that use concrete visual metaphors: *PlanAni* and *Metaphor-based OO visualizer*. However, these metaphors are unconnected and verbal explanations are required as a complement.

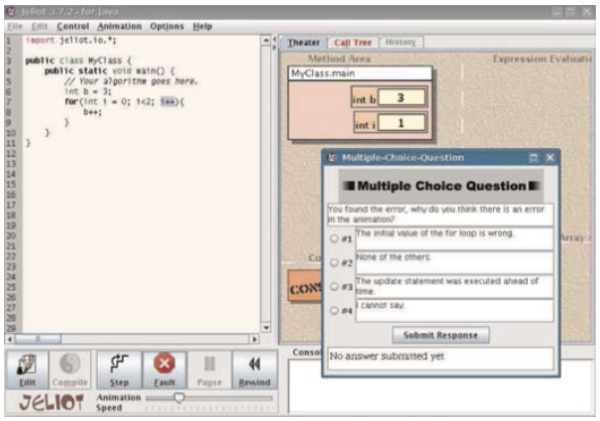


Fig 2. Jeliot ConAn screenshot (source: [34])

Principle 13, about constructing concept systems, is mainly supported by some questions in *ViLLE* that require combining a number of programming concepts. For example, the Java's `String.length()` method is called within several arithmetic expressions, and users must correctly apply operator precedence rules to determine whether integer sums, string concatenations, or both are executed.

Principle 14, about evaluating a process rather than a final product, is mainly supported by *UUhistle*. While students produce an animation, *UUhistle* provides immediate feedback about expected or wrong actions. However, the tool does not track or inform the overall student' progress. This overall evaluation may be used, for example, to avoid students over-animating concepts that they understand correctly.

The remaining eight principles are almost not supported by any existing program visualization. In another words, no program visualization investigates what motivates students (principle 2), evaluates previous notions or emotions (principle 4), compares new against known concepts (principle 8), exercises a new concept by applying it to several contexts (principle 9), solve real problems applying the new concepts (principle 11) until students develop an habit (principle 12), evaluates that students' notions and skills are stable in time (principle 15), and fosters students to transfer this knowledge to new situations (principle 16). This result opens a door for future research.

VI. Conclusions and Future Work

Program visualizations are important educational tools to help learners understand how programs are executed by computers. A previous literature review found 46 generic program visualizations from 1979 to 2012 [2]. We updated the list with seven additional tools found from 2013 to first quarter of 2016.

Because the effectiveness of program visualizations as learning tools is unclear, we inferred from the social constructivism theory, 16 learning principles that may contribute to their effectiveness. We also subjectively evaluated the implementation of these principles in the available program visualizations. We found moderate support for two principles, and almost no support for the rest. This finding is the most important contribution of this paper, because provides a rich horizon for further research.

We hypothesize that many of the identified constructivist principles may be implemented using visual concrete allegories and gamification. Our review found little support for gamification and concrete visual metaphors, but none for visual concrete allegories. Further research is required in order to know if these techniques positively influence the effectiveness of program visualizations as learning tools.

Acknowledgment

This research is supported by the Centro de Investigaciones en Tecnologías de la Información y Comunicación (CITIC), the Escuela de Ciencias de la Computación e Informática (ECCI), both from Universidad de Costa Rica (UCR), and the Ministerio de Ciencia Tecnología y Telecomunicaciones de Costa Rica (MICITT).

References

[1] J. Sorva, "Visual Program Simulation in Introductory Programming Education," Aalto University, 2012.

[2] J. Sorva, V. Karavirta, and L. Malmi, "A Review of Generic Program Visualization Systems for Introductory Programming Education," *ACM Trans. Comput. Educ.*, vol. 13, no. 4, pp. 1–64, Nov. 2013.

[3] T. L. Naps, S. Rodger, J. Á. Velázquez-Iturbide, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, and M. McNally, "Exploring the role of visualization and engagement in computer science education," *ACM SIGCSE Bull.*, vol. 35, no. 2, p. 131, Jun. 2003.

[4] E. Isohanni and H.-M. Järvinen, "Are visualization tools used in programming education?: by whom, how, why, and why not?," in *Proceedings of the 14th Koli Calling International Conference on Computing Education Research - Koli Calling '14*, 2014, pp. 35–40.

[5] M. Hertz and M. Jump, "Trace-based teaching in early programming courses," in *Proceeding of the 44th ACM technical symposium on Computer science education - SIGCSE '13*, 2013, p. 561.

[6] M. Berry and M. Kölling, "The design and implementation of a notional machine for teaching introductory programming," in *Proceedings of the 8th Workshop in Primary and Secondary Computing Education on - WiPSE '13*, 2013, pp. 25–28.

[7] G. Ebel and M. Ben-Ari, "Affective effects of program visualization," in *Proceedings of the 2006 international workshop on Computing education research - ICER '06*, 2006, p. 1.

[8] B. du Boulay, T. O'Shea, and J. Monk, "The black box inside the glass box: presenting computing concepts to novices," *Int. J. Man. Mach. Stud.*, vol. 14, no. 3, pp. 237–249, Apr. 1981.

[9] E. Isohanni, "Visualizations in Learning Programming Building a Theory of Student Engagement," Tampere University of Technology, 2013.

[10] W.-H. Wu, W.-B. Chiou, H.-Y. Kao, C.-H. Alex Hu, and S.-H. Huang, "Re-exploring game-assisted learning research: The perspective of learning theoretical bases," *Comput. Educ.*, vol. 59, no. 4, pp. 1153–1161, Dec. 2012.

[11] T. Greening, "Emerging Constructivist Forces in Computer Science Education: Shaping a New Future?," in *Computer Science Education in the 21st Century*, T. Greening, Ed. New York: Springer New York, 2000, pp. 47–80.

[12] K. C. Powell and C. J. Kalina, "Cognitive and Social Constructivism: Developing Tools for an Effective Classroom," *Education*, vol. 130, no. 2, pp. 241–250, Nov. 2009.

[13] W. Rodríguez-Arocho, "Herramientas culturales y transformaciones mentales: De los jeroglíficos a la internet," *Ciencias la Conduct.*, vol. 17, pp. 12–19, 2002.

[14] M. Tam, "Constructivism, Instructional Design, and Technology: Implications for Transforming Distance Learning," *J. Educ. Technol. Soc.*, vol. 3, no. 2, pp. 50–60, 2000.

[15] S. Francis, *El conocimiento pedagógico del contenido como modelo de mediación docente*. Coordinación Educativa y Cultural, 2012.

[16] Luria, Leontiev, and Vigotsky, *Psicología y pedagogía*, 4th ed. Sevilla, España: Ediciones Akal, 2011.

[17] E. Harmon-Jones and J. Mills, "An introduction to cognitive dissonance theory and an overview of current perspectives on the theory.," in *Cognitive dissonance: Progress on a pivotal theory in social psychology*, E. Harmon-Jones and J. Mills, Eds. Washington, DC: American Psychological Association, 1999, p. 411.

[18] G. Lakoff, "The Contemporary Theory of Metaphor," in *Metaphor and Thought*, 2nd ed., no. c, A. Ortony, Ed. Cambridge University Press, 1993, pp. 202–251.

[19] J. P. Sanford, A. Tietz, S. Farooq, S. Guyer, and R. B. Shapiro, "Metaphors we teach by," in *Proceedings of the 45th ACM technical symposium on Computer science education - SIGCSE '14*, 2014, pp. 585–590.

[20] T. R. Colburn and G. M. Shute, "Metaphor in computer science," *J. Appl. Log.*, vol. 6, no. 4, pp. 526–533, Dec. 2008.

[21] D. Heath, D. Norton, and D. Ventura, "Conveying Semantics through Visual Metaphor," *ACM Trans. Intell. Syst. Technol.*, vol. 5, no. 2, pp. 1–17, Apr. 2014.

[22] P. Crisp, "Between extended metaphor and allegory: is blending enough?," *Lang. Lit.*, vol. 17, no. 4, pp. 291–308, Nov. 2008.

[23] L. J. Waguespack, "Visual metaphors for teaching programming concepts," in *ACM SIGCSE Bulletin*, 1989, vol. 21, no. 1, pp. 141–145.

[24] A. Moreno and N. Myller, "Producing an Educationally Effective and Usable Tool for Learning, The Case of the Jeliot Family," in *Proceedings of International Conference on Networked e-learning for European Universities (EUROPACE'03)*, 2003.

[25] M. K. Smith, H. R. Pollio, and M. K. Pitts, "Metaphor as intellectual history: conceptual categories underlying figurative usage in American English from 1675-1975," *Linguistics*, vol. 19, pp. 911–935, 1981.

[26] J. Sajaniemi and M. Kuittinen, "Program animation based on the roles of variables," in *Proceedings of the 2003 ACM symposium on Software visualization - SoftVis '03*, 2003, p. 7.

[27] K. M. Kapp, *The Gamification of Learning and Instruction: Game-based Methods and Strategies for Training and Education*, 1st ed. Pfeiffer, 2012.

[28] B. A. Myers, "Taxonomies of visual programming and program visualization," *J. Vis. Lang. Comput.*, vol. 1, no. 1, pp. 97–123, Mar. 1990.

[29] B. A. Price, R. M. Baecker, and I. S. Small, "A Principled Taxonomy of Software Visualization," *J. Vis. Lang. Comput.*, vol. 4, no. 3, pp. 211–266, Sep. 1993.

[30] J. I. Maletic, A. Marcus, and M. L. Collard, "A task oriented view of software visualization," in *Proceedings First International Workshop on Visualizing Software for Understanding and Analysis*, 2002, pp. 32–40.

[31] J. Urquiza-Fuentes and J. Á. Velázquez-Iturbide, "A Survey of Successful Evaluations of Program Visualization and Algorithm Animation Systems," *ACM Trans. Comput. Educ.*, vol. 9, no. 2, pp. 1–21, Jun. 2009.

[32] S. Xinogalos, "Using flowchart-based programming environments for simplifying programming and software engineering processes," in *2013 IEEE Global Engineering Education Conference (EDUCON)*, 2013, pp. 1313–1322.

[33] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Springer Berlin Heidelberg, 2012.

[34] A. Moreno, E. Sutinen, and C. I. Sedano, "A game concept using conflictive animations for learning programming," in *2013 IEEE International Games Innovation Conference (IGIC)*, 2013, pp. 175–178.

[35] A. Moreno, E. Sutinen, and M. Joy, "Defining and evaluating conflictive animations for programming education," in *Proceedings of the 45th ACM technical symposium on Computer science education - SIGCSE '14*, 2014, pp. 629–634.

[36] M. Berry and M. Kölling, "The state of play: a notional machine for learning programming," in *Proceedings of the 2014 Conference on Innovation & ...*, 2014, pp. 21–26.

[37] M. Berry and M. Kölling, "Novis: A notional machine implementation for teaching introductory programming," in *Fourth International Conference on Learning and Teaching in Computing and Engineering (LATICE 2016)*, 2016.

[38] C. Egan, Matthew Heinsen; McDonald, "Runtime error checking for novice C programmers," in *4th Annual International Conference on Computer Science Education: Innovation and Technology (CSEIT 2013)*, 2013, pp. 1–9.

[39] M. H. Egan and C. McDonald, "Program visualization and explanation for novice C programmers," in *Proceedings of the Sixteenth Australasian Computing Education Conference - Volume 148*, 2014, pp. 51–57.

[40] D. Pawelczak and A. Baumann, "Virtual-C - a programming environment for teaching C in undergraduate programming courses," in *2014 IEEE Global Engineering Education Conference (EDUCON)*, 2014, pp. 1142–1148.

[41] Y. Yan, N. Hiroto, H. Kohei, S. Shota, and A. He, "A C Programming Learning Support System and Its Subjective Assessment," in *2014 IEEE International Conference on Computer and Information Technology*, 2014, pp. 561–566.

[42] S. Al-Fedaghi and A. Alrashed, "Visualization of Execution of Programming Statements," in *2014 11th International Conference on Information Technology: New Generations*, 2014, pp. 363–370.

[43] J. Yang, Y. Lee, D. Hicks, and K. H. Chang, "Enhancing Object-Oriented Programming Education using Static and Dynamic Visualization," in *2015 IEEE Frontiers in Education Conference*, 2015.

[44] J. Sajaniemi, P. Byckling, and P. Gerdt, "Animation Metaphors for Object-Oriented Concepts," *Electron. Notes Theor. Comput. Sci.*, vol. 178, pp. 15–22, Jul. 2007.

[45] J. Sorva and T. Sirkiä, "UUhistle: a software tool for visual program simulation," in *Proceedings of the 10th Koli Calling International Conference on Computing Education Research - Koli Calling '10*, 2010, pp. 49–54.