Enabling Program Comprehension through a Visual Object-focused Development Environment

Fernando Olivero, Michele Lanza, Marco D'Ambros
REVEAL @ Faculty of Informatics - University of Lugano, Switzerland

Romain Robbes
PLEIAD@DCC - University of Chile

***Abstract*—Integrated development environments (IDEs) include many tools that provide the means to construct programs. Coincidentally, the very same IDEs are a primary vehicle for program comprehension. We claim that IDEs may be an impediment for program comprehension because they treat software elements as text, which may be counterproductive in the context of program understanding—where abstracting from the source text to the level of structural entities and relationships is the key.**

We are currently building *Gaucho*, a visual object-focused environment that allows developers to write programs by creating and manipulating lightweight and intuitive depictions of object-oriented constructs. The research question we investigate here is how such an environment compares with traditional IDEs when it comes to performing program comprehension tasks.

To answer our question, we conducted a preliminary controlled experiment with eight subjects, comparing Gaucho against a traditional IDE. We found that Gaucho outperforms the IDE regarding the correctness of the tasks, while it is slower with respect to the completion time. Our preliminary results suggest that alternative—visual—IDEs may be superior to traditional IDEs as program comprehension aids.

I. Introduction

Nowadays object-oriented developers perform programming tasks aided by IDEs, which feature numerous tools that provide the means to construct programs. Nevertheless, IDEs present difficulties that have been previously stated in the literature [1], [2], [3]; e.g., they introduce accidental complexity due to their file-based dependence, since extra navigation is required to search back and forth for scattered code fragments. While there is certainly room for improvement in the context of forward engineering, the usage of IDEs as program comprehension aids has not been questioned.

According to Chikofsky and Cross program comprehension is focused on *"identifying the system's components and their interrelationships, as well as creating representations of the system in another form or at a higher level of abstraction"* [4]. We argue that IDEs hinder program comprehension because they work on a textual representation of a system, the *source code*, and therefore lack the proper level of abstraction required for understanding the programs. IDEs are specialized for navigating and editing text, thus supporting the notion that programming is writing, as Weinberg [5] argued nearly 40 years ago. Nonetheless, object-oriented programming is better defined as *modeling*, rather than writing: A modeling of the composition and collaborations of the objects under construction. Even though external modeling tools are better suited for program comprehension than IDEs [6], [7], there are several reasons for which developers are reluctant to use anything but the IDE for their activities: One of them is that they are not willing to invest time and effort in learning new tools if they do not perceive a tangible benefit in it [8].

We claim that an environment based on a different metaphor can intrinsically improve the support for program comprehension, and minimize the need to recur to external tools for understanding the programs. The two contrasting metaphors here are the *view-focused* and the *object-focused* metaphors. Mainstream IDEs, such as Eclipse, are view-focused: Objects are secondary to the tools which are the concrete visual elements available for interaction. On the contrary, object-focused environments, such as Self [9], foster the notion that developers are in direct contact with the objects themselves, instead of abstract entities subordinated to the tools. Our goal is to step away from IDEs as text editors and file navigators, and embrace a vision where environments ease the interaction with and crafting of objects, as well as their comprehension. This eliminates the presence of tools that create a barrier between a developer and the program to be developed and understood. We have implemented *Gaucho* [10], a visual object-focused development environment. We argue Gaucho eases the comprehension of a system, through the following features:

- *Abstractions*. The cognitive burden is lessened in Gaucho, because developers interact with graphical elements depicting software artifact as high-level views, as opposed to raw text that the developer must decode into meaningful chunks of information.
- *Relationships*. The graphical elements depicting the objects provide quick access to related entities, favouring incremental exploration of the system, and easing the navigation of the relationships between objects.
- *Unconstrained Layout*. The graphical elements can be freely placed on the interface, making it straightforward to create side by side views of the objects for understanding and comparing them.

To assess our claim, we performed a preliminary experiment, with promising results. The contributions of this paper are (1) an in-depth reflection on the current state-of-the-practice of development environments; (2) a presentation of *Gaucho*, a novel object-focused environment we are building; and (3) a preliminary controlled experiment with eight subjects to compare a traditional IDE and Gaucho with respect to a set of program comprehension tasks extracted from the literature.

II. Background

The word *program* has an ambiguous meaning that can only be disambiguated by situating the reader in a particular programming paradigm. In the functional paradigm a program is a collection of functions; in the imperative paradigm a program is made up from data structures and algorithms [11]; in the object-oriented paradigm—which is our focus—programming revolves around collaborating objects, instantiated from classes, that send each other messages [12].

If we reflect upon the current understanding of what constitutes an object-oriented program, we find that developers perceive programs as a large collection of files that contain text, i.e., the source code. The formal linguistic aspect of programs, the source code, and the tools one uses to construct them, assume a preponderant role and dictate how one perceives the act of programming. Hutchins et al. stated that any user interface provides a frame of reference for reasoning about the problems [13]; therefore, the manner in which IDEs present software systems influences how developers reason about them. Is programming writing? An object-oriented program is more than a conglomeration of code organized into files. Even the vocabulary of modern object-oriented developers has steadily evolved since its inception: Nowadays, developers use words such as building, constructing, architecting, and designing [14].

We believe that the lack of abstraction in the IDEs, due to the textual representation of the software artifacts, hinders program comprehension since the composition and relationships between the objects are difficult to visualize when translated to raw text. Developers recur to external modeling tools or even rough sketches, presenting the conceptual abstractions of the program and their relationships in the form of visual diagrams. Others have questioned the usage of textual representations of code entities for program comprehension, e.g., Wettel et al. have shown that making use of an alternative metaphor for depicting the programs—as 3-D cities—eased reverse engineering [15].

Limitations of Mainstream IDEs. IDEs are view-focused tools specialized for navigating and editing programs, viewed as textual representations of classes, methods and packages. Mainstream IDEs, such as Eclipse and Visual Studio, feature numerous tools providing the means to construct and comprehend programs. An often adopted fashion to arrange the various tools, termed by some as a *bento box approach*, is depicted in Figure 1.

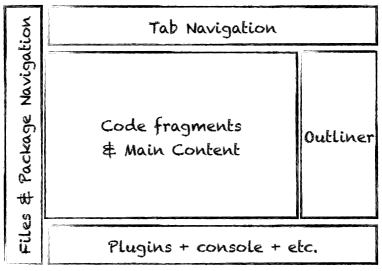


Fig. 1. Sketch of mainstream IDEs

Developers using IDEs struggle with difficulties when navigating software systems. Artifacts related to a single concept are distributed in a huge space—the entire code base—and the relationships between them remain hidden; this forces developers to open numerous views on artifacts to reveal their relationships, leading to a crowded workspace with many opened windows or tabs [3]. In the arrangement depicted in Figure 1, developers are forced to continuously move around the focus of attention by scrolling up and down through the code, and managing several tabs when relating separate entities, as it is hard to create side by side views of the code [1] [2]. Relying on tab-based views depicting files of the system is inadequate since most tasks are not aligned with the structure of the IDE and require navigating to different parts of the system [16]. The single window interface forces plugin developers to compete for the available real estate, making it difficult to extend the IDE. One could argue that power users develop subconscious habits that over time allow them to overcome the limitations of the tools in use. We believe that developers accustomed to mainstream IDEs find it difficult to comprehend that the IDE treats objects as text. Being used to the tools working on a textual representation makes developers mentally visualize the structure of the objects at a higher level, instead of the textual code fragments displayed on the screen. A supporting fact are the several pedagogical environments, such as BlueJ [17] and Alice [18], that present high-level views of the programs to ease the learning of object-oriented programming concepts.

Beyond IDEs: Object-focused Environments. Developers use IDEs to navigate and interact with the source code. Even if developers recur to external tools for sketching and analyzing the conceptual abstractions of the programs [7], the main entry point remains the IDE. We argue that the principal medium of interaction between developers and the software systems under construction—the IDE—should favor modeling over writing, thus improve the support for program comprehension. In our vision, an environment built around the object-focused metaphor would achieve such a goal, by embracing object manipulation instead of text editing.

View-focused environments, like traditional IDEs, support a *conversational* notion: Tools (editor, debugger, etc.) are the medium in which developers converse about the objects. On the contrary, object-focused environments are built on a model-world metaphor, populated by high-level views of the objects, presented as directly manipulable graphical elements that do not need intermediary tools [19], [10]. Traditional IDEs make use of direct manipulation as the means to modify the objects, e.g., methods can be moved using drag & drop, and refactorings are available from a contextual menu that acts upon a code fragment. IDEs provide some form of high level views, such as the outliner depicting the structure of classes and packages. However, the feeling of direct engagement is lessened, as the direct manipulation operations and abstractions are decoupled from the textual representations of the objects. Object-focused environments attain a higher level of direct manipulation and abstraction, by coupling operations and representations into the same graphical element, removing explicit tools.

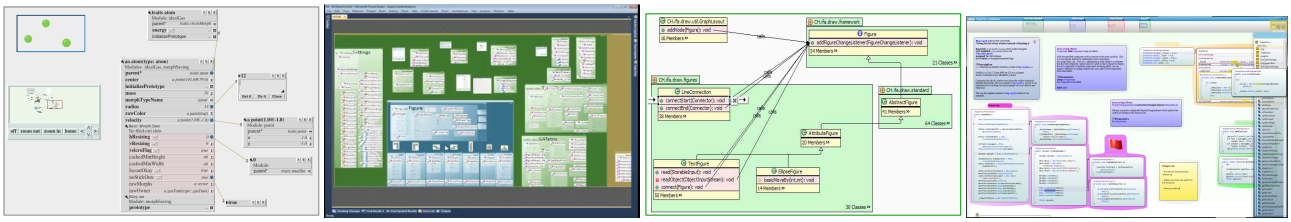


Fig. 2. Environments related to Gaucho: Self, CodeCanvas, Relo, CodeBubbles

III. Related Work

Researchers and practitioners proposed several programming environments that either abstract from the text-based metaphor, or propose an alternative metaphor for manipulating code. Figure 2 depicts the ones most related to Gaucho.

Self [9] is the seminal object-focused environment. Self is both a language and an interface for direct manipulation of uniform graphical objects that populate a malleable world. A Self object has a single outliner tool that reveals the inner structure and provides the means to manipulate itself. In the Self prototype-based language, all the objects have the same structure, hence a single tool can suffice. *Code Canvas* [20] provides an infinite zoomable surface for software development. A canvas both houses editable forms of project documents and allows multiple layers of visualization over those documents. Code Canvas leverages spatial memory to keep developers oriented. *Code Bubbles* [1] is a programming environment that represents code as lightweight, editable fragments called bubbles, forming concurrently visible working sets that avoid the continuous back and forth navigation typical of traditional IDEs. The bubbles exist on a pannable 2-D space. *Relo* [2] supports program comprehension by enabling and facilitating interactive code exploration. The graphical elements depict high level views of software elements presented as UML class diagrams. "Navigation buds" on each graphical element allow developers to build graphs of software elements; Relo automatically places each node according to relationships such as inheritance, or containment.

Inspiration & differences. Gaucho takes inspiration from Self, but provides a richer set of graphical elements because instead of a prototype-based language it is designed for Smalltalk [12], [14], a dynamic object-oriented language: different kinds of objects coexist in the world, such as classes, methods and packages, whereas Self offers a uniform interface. Code Canvas and Code Bubbles address navigation problems of mainstream file-based IDEs, using interfaces based on alternative metaphors, thus making better use of the spatial memory and the available real estate. In Gaucho we adopted a similar approach by providing graphical elements—depicting software entities—that can be freely placed in a 2-D canvas. These graphical elements go beyond text: The developer interacts with different visual representations for each software artifact, stepping away from editing textual representations of code. Code Bubbles instead are defined as interactive views of source code fragments such as methods or collection of member variables. Relo presents high-level views of software elements that provide the means to interactively navigate between their relationships. Gaucho provides the same facilities for interactive and incremental exploration of the code, without constraining the layout of the graphical elements.

IV. Gaucho

Gaucho[1] [10] is an object-focused environment built on top of Pharo [21], a modern open-source Smalltalk IDE. Gaucho is based on a model-world metaphor populated by directly manipulable representations of the artifacts that make up a software system. Figure 3 shows a screenshot of the current version of Gaucho. The two pivotal concepts in the tool are:

1) *Pampas*, a 2-D surface hosting the graphical elements that make up a system (e.g., packages, classes, methods, class references, recent changes) as *shapes*.
2) *Shapes*, manipulable high-level views of software artifacts that populate the pampas.

Gaucho includes a specific shape for each type of software artifact. Such shapes depict every aspect of the underlying artifacts and provide the means to manipulate themselves. Gaucho shapes have the following properties:

a) Direct manipulation: Shapes provide quick access to the most important operations that act upon them in the form of buttons., e.g., a class shape presents the attributes and methods of the class, and provides buttons for adding new methods and attributes; a package shape presents the list of classes of the package, and provides a button for adding new classes.

b) Exploring the system: Shapes provide a customized set of navigation icons for exploring their relationships across the system. For instance, class shapes provide icons for opening the group of class references, the class hierarchy, the group of subclasses and the package of the class. The icons have a prominent placement since direct tool support for interactive exploration helps one to manage the context and perform program comprehension tasks [2].

c) Pampas layout: Shapes can be freely placed within the pampas, thus eliminating the constraints of mainstream file-based IDEs, that force code to be confined into tabbed text areas competing for real estate and focus. The customizable layout and the persistent visual arrangement allows developers to use secondary notations, such as spatial memory, and to form side-by-side views of objects.

[1] Available at http://gaucho.inf.usi.ch

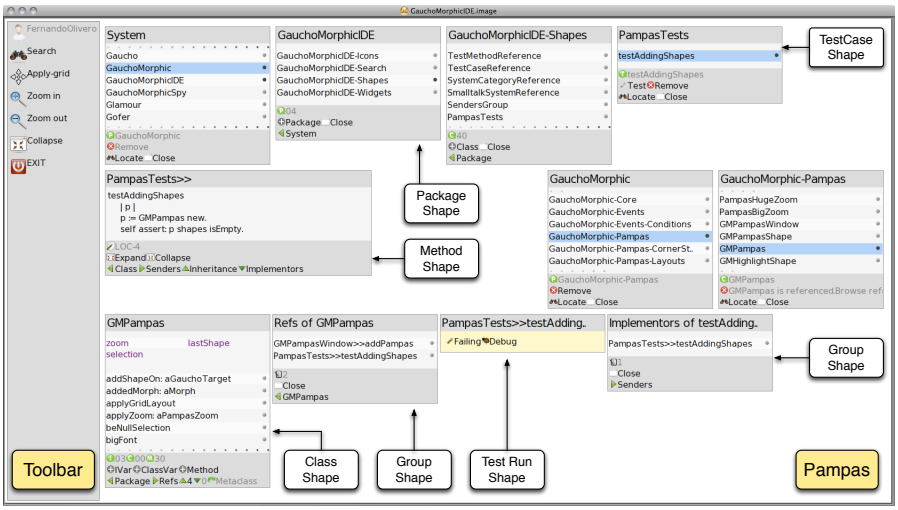


Fig. 3. The Gaucho development environment

d) Abstractions: In Gaucho the burden of the UI is lessened because a developer interacts with a class, a method, a package, a developer and a system shape in a uniform and consistent manner. Gaucho presents software artifacts as high-level views—instead of raw text that the developer must decode into meaningful chunks of information—thus easing the comprehension of the structure and relationships between the system's objects. Text is the means to specify behavior, i.e., to write the statements that make up a method. We avoid visual representations at the behavioral level, because at this level, visual programming makes the understanding of the code a convoluted process; choosing an intuitive visual notation is cumbersome and non-trivial [22].

V. Evaluation

We claim that the use of Gaucho eases the comprehension of the structure and relationships between the entities making up a software system, compared to the use of traditional IDEs. To assess the validity of this claim, we performed a preliminary controlled experiment. This experiment served also to detect usability issues and to collect impressions from developers using Gaucho.

Research Questions. The research questions underlying our experiment are:

- RQ1. Does the use of Gaucho reduce the *time* necessary to perform program comprehension tasks, compared to a traditional IDE?
- RQ2. Does the use of Gaucho increase the *correctness* of the answers to the program comprehension tasks, compared to a traditional IDE?

Variables. The purpose of the experiment is to assess the metaphor in use by Gaucho. Thus, the use of the metaphor is the single independent variable of the experiment. This variable has two levels: the object-focused metaphor and the view-focused metaphor, respectively represented by Gaucho and a baseline. The dependent variables of our experiment are correctness of the task solutions and their completion time.

Baseline. We searched for a baseline within the modern object-oriented IDEs that treat objects as text and settled on the Pharo Smalltalk IDE, a traditional image-based (not depending on files) IDE built around a WIMP (windows, icons, menus, pointing device) metaphor [8]. Since Pharo—as many mainstream IDEs—includes the standard development tools for navigating, inspecting and testing the objects in the system, we found it to be an ideal candidate to compare Gaucho to. While it would be interesting to compare Gaucho against the non-bento box tools presented in III, this is not our goal.

Object system & Treatments. We chose *Lumière* [23] as our object system. *Lumière* is a framework for creating and rendering 3-D scenes in OpenGL, consisting of 10 packages, 139 classes, 1,741 methods, for a total of 9,493 lines of code. The system is large enough that subjects unfamiliar with it have to perform program exploration and program comprehension to complete the tasks they were asked to perform. Subjects were randomly assigned one of the two following treatments:

Tool	Object	Description
Gaucho	Lumiere	Gaucho application with a loaded model of Lumiere
Pharo	Lumiere	Pharo IDE with default development tools and a loaded model of Lumiere

Operation. The experiment was performed at the University Of Chile (with 3 professors, 1 PhD student, 1 software engineer, and 2 Msc students) and at the University Of Lugano (with one MSc student). We presented a video demonstration of Gaucho to each subject of the experimental group. An experimental run consisted in a session of up to one hour, during which the subjects solved the tasks with the assigned treatment.

Data collection. We used an automated experiment runner toolset, called *Biscuit*, that (1) presented the experiment to the subjects, (2) collected subjects data, (3) guided them through the tasks until completion, (4) stored the answers and durations of each task, and (5) issued a post-experiment questionnaire.

TABLE I
TASKS OF THE EXPERIMENT

Id	Goal	Questions
T1	Locate the class that represents a rotation of a scene graph in the Lumiere framework	Q1
T2	Indicate the correct names of all the instance variables of the class LLight	Q6, Q16, Q17
T3.1	Locate the root class of the hierarchy of nodes of a scene graph in Lumiere	Q1, Q8
T3.2	Indicate the correct names of all the (direct) subclasses of LLumiereShape	Q9
T3.3	Indicate all the subclasses (direct or not) of LMiddleNode that override the method #accept:	Q11
T4.1	How many packages make up Lumiere?	Q6
T4.2	Indicate the two packages in Lumiere with the largest number of classes	Q7
T5.1	Create a new layout class named *LStackLayout*, located in the same package and with the same superclass as LPolarLayout	Q7, Q8, Q17
T5.2	Add an instance variable named "translations" to *LStackLayout*, and create the accessors	Q17
T6.1	The base Layout implements the method *#runLayout*. Indicate all the layout classes that implement the same method	Q5, Q11
T6.2	Indicate the name of the instance variable that is assigned (set) by most implementors of the method *#runLayout*	Q10, Q15
T6.3	Define the method $LStackLayout >> runLayout$ (cut and paste the code)	Q6
T7.1	How many test case methods reference the class *LVerticalGridLayout* ?	Q15
T7.2	Indicate the test case class where most of the layout behavior is tested	Q6, Q12

Biscuit provided the means to set up an experiment made up of tasks, each task consisting in a description and goals to be accomplished by the subjects. Using Biscuit, we automatically generated a user interface for each experimental session, via an application running either on top of Gaucho or Pharo. Figure 4 shows Biscuit running on top of Gaucho.

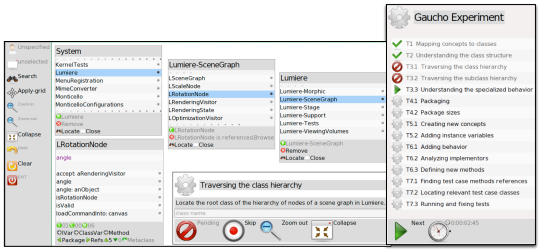


Fig. 4. Biscuit: task list and a running task example

We collected the data produced by each subject in the form of a Biscuit output file. The output file contains the answers and solutions to each task, and some meta-data such as the participant's name, the total completion time, the correctness of the answers, and the post-experimental task evaluations.

Tasks. We designed 3 programming and 11 program comprehension tasks based on the set of questions composed by Sillito et al. [6]. Sillito et al. organized the questions into 4 categories: (1) finding focus points, (2) expanding focus points, (3) understanding a subgraph, and (4) questions over groups of subgraphs. We devised the tasks of the experiment based on the first two categories. We omitted questions from the 3rd and 4th category because both Gaucho and Pharo lack support for maintaining context while answering multiple questions. The rationale supporting the tasks is that each of them relates to one or more questions of Sillito et al., i.e., real questions developers ask during their development activities.

We selected a set of questions pertaining to program comprehension, designed tasks related to them that can be solved using both treatments, and ordered them according to similarity and prerequisites of the necessary data.

TABLE II
REFERENCED QUESTIONS FROM SILITO'S FRAMEWORK

Q1	Which type represents this domain concept/UI element/action?
Q5	Is there an entity named XXX in that project/package/class?
Q6	What are the parts of this type?
Q7	Which types is this type a part of?
Q8	Where does this type fit in the type hierarchy?
Q9	Does this type have any siblings in the type hierarchy?
Q10	Where is this field declared in the type hierarchy?
Q11	Who implements this interface or these abstract methods?
Q12	Where is this method called or type referenced?
Q15	Where is this variable or data structure being accessed?
Q16	What data can we access from this object?
Q17	What does the declaration or definition of this look like?

In Table I we list the tasks, and in Table II the developer questions included in the experiment.

Subject and expertise analysis. At the beginning of each experimental session run by Biscuit, the subjects answer questions related to their expertise.

TABLE III
SUBJECT'S EXPERTISE

	Pharo			Gaucho	
Expertise	Pharo	OOP	Smalltalk	OOP	Smalltalk
None	1	0	0	0	1
Beginner	3	1	3	1	1
Knowledgeable	0	1	1	1	1
Advanced	0	2	0	0	1
Expert	0	0	0	2	1

The results, depicted in Table III, show that the subjects of the control group have little or no experience using the Pharo IDE; the same level of expertise can also be assumed for the subjects of the experimental group, given that their first encounter with Gaucho occurred during this experiment. The expertise of the subjects—regarding Object-oriented concepts in general, and Smalltalk in particular—is also balanced among treatments, as can be seen in Table III. The subjects were unfamiliar with the object system chosen, the *Lumière* toolset.

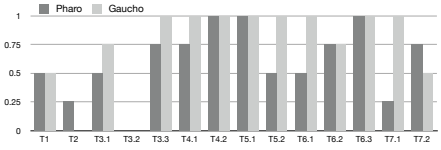
VI. Results

The 14 tasks were automatically graded, yielding a maximum score of 14 points, and the time taken to solve each task was measured by means of the ouput analyzer features of Biscuit.

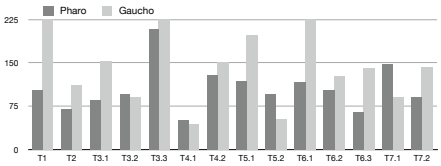
The subjects of the experimental group outperformed the subjects of the control group regarding the correctness of the tasks (cf. Figure 5(a)). The subjects using Gaucho scored, on average, 10.4 points, while the subjects using the Pharo treatment scored, on average, 8.5 points.

On the contrary, regarding the completion time, the subjects of the control group outperformed the experimental group (cf. Figure 5(b)). The total completion time, on average, of the subjects using the Gaucho treatment was 38:08 minutes, while the subjects using the Pharo treatment spent, on average, 28:48 minutes for all the tasks.

Therefore, although the limited number of subjects does not allow us to draw any statistically relevant conclusions, our data gives us strong indications that we can answer the first research question (time) negatively, and the second one (correctness) positively. We now proceed with a qualitative evaluation of the results of each task.



(a) Task correctness



(b) Completion time (seconds)

Fig. 5. Experiment results

Task analysis

To better analyze the obtained results we grouped the tasks according to the similarities between their goals. In the following discussion, for each group of tasks, we present the IDs, a summary of the goals to be accomplished by the subjects, and comment the obtained results.

T1: The goal is to find the class named *LRotationNode*, by searching the class that most resembles a rotation node of a Lumière scene graph. We wanted to assess if the global search widget of Gaucho was accessible and provided the means to effectively perform concept location; it is similar to the lexical or static analysis based search tools of IDEs. The results of the subjects of the experimental group were similar to the control group in terms of correctness. However, they spent—on average—more time than the subjects in the control group, mostly because of usability issues with the global search widget of Gaucho: it currently lacks support for performing concept location via pattern matching.

T2, T4.1, T4.2: These tasks relate to structural comprehension of classes and packages. The goal is to choose the correct answer from a multiple choice scheme of the set of instance variables of the class *LLight* (task T2), the number of packages that make up Lumière (task T4.1), and the two packages in Lumière with the largest number of classes (task T4.2). With these tasks we investigate whether the structure of an object is correctly understood when depicted using a high-level view—the class or package shape—as opposed to lower-level mechanisms such as reading a textual representation of the class definition in the IDE. The results for task T2 show that the subjects of the control group outperformed the subjects using Gaucho. This negative score was a result of a misunderstood scrollable widget of the class shape (cf. Figure 3), which only reveals four instance variables at a time; most subjects were not aware that this list could be scrolled down to reveal the fifth instance variable, thus answering incorrectly. The results for T4.1 and T4.2 show that the experimental group outperformed the control group, but the subjects using Gaucho took more time on T4.2 and less on T4.1 than the subjects using Pharo. In T4.1, a single package shape is involved (the package shape representing Lumière), whereas T4.2 requires interacting with a larger number of package shapes, as they must be placed side by side to compare their sizes; there are still usability issues when positioning shapes within the pampas.

T3.1, T3.2, T6.2, T7.1: With these tasks we wanted to assess the usability and validity of the direct manipulation features available in the shapes, which allow navigating between the relationships of the represented object with the rest of system. The goal is to locate the root class of the hierarchy of nodes of a scene graph in Lumière (T3.1), to choose the correct names of all the (direct) subclasses of *LLumiereShape* amongst 4 choices (T3.2), to indicate all the layout classes that implement the method *#runLayout* (T6.2), and to specify how many test case methods reference the class *LVerticalGridLayout* (T7.1). Tasks T3.1 and T3.2 both involved navigating the class hierarchy from a class shape, the first on super classes and the second on subclasses. Task T6.2 revolved around navigating references to the uses of methods, while Task T7.1 revolved around navigating the references (uses) of a class, and narrowing the results into those who are test case methods. The subjects of the experimental group outperformed the subjects in the control group in the task T3.1, regarding correctness of the answers; but again they did so by taking more time than the subjects using the Pharo treatment , because of the mentioned layout problems when displaying multiple shapes on the pampas. The subjects using the Gaucho treatment were faster and more correct on the task T7.1. The navigation facilities on the shapes allowed quick access to the references of the *LVerticalGridLayout* class, as opposed to the traditional use of a contextual menu—on a selected list item in the browser—for navigating the references of the class. In task T6.2 both groups performed similarly regarding the correctness, but the subjects using the Gaucho treatments were faster because placing side by side shapes depicting the relevant methods eased performing the comparison. Unfortunately, the subjects in both groups

failed task T3.2 because of a misunderstanding in the goals description; the subjects also included indirect subclasses in the answer. Nevertheless, since the subjects answered all the indirect subclasses correctly and given the similarities between the completion time, we can conclude that a class hierarchy can be well understood in both treatments because they share a similar graphical depiction of a class and its subclasses.

T3.3, T6.1: We wanted to observe how well Gaucho behaves on tasks that force the subject to look at multiple classes, methods and relationships at once, in order to assess the supposed benefit of the unconstrained layout of the Pampas and the simple graphical representations of objects, against the rigid and complex tools of the IDE. The goal is to indicate all the subclasses of *LMiddleNode* that override the method *#accept* (T3.3), and indicate the name of the instance variable that is assigned (set) by most implementors of the method *#runLayout* (T6.1). The subjects of the experimental group outperformed the subjects of the control group regarding the correctness, but again spent more time solving the task. The difference was more marked in T6.1, possibly because by then the subjects had already gained experience and made better use of the Pampas and the shapes.

T7.2: The goal is to indicate the test case class where most of the layout behavior is tested. We devised this task because before implementing a test, the developers must locate the proper test case class where the test method should be added; this forces developers to search across the test cases of the system to find the one that references the most the Lumière layout classes. On this task the subjects using Pharo performed better than those using Gaucho. We observed that grouping all the tests into a single tool, called the test runner, allowed the Pharo users to quickly locate the desired test. In Gaucho the subjects could access one test at a time by opening a class shape of a Lumière layout class, and navigate the test case references to find the most suitable one, which is less efficient.

T5.1, T5.2, T6.3: The goal of these tasks is to create a class, an instance variable, and several methods. To evaluate the completeness of Gaucho, we wanted to assess whether novel users of the object-focused environment could create and manipulate new classes and methods, by interacting with the shapes, without intermediaries (the tools). The subjects of the experimental group made use of the direct manipulation facilities of the shapes to correctly create and modify the requested objects; for example by creating the class using the add button of the package shape, instead of editing the class definition in the browser tool of the IDE. The difference in correctness in T5.2—and completion time—is due to some Pharo users failing to create the accessors, while Gaucho creates them automatically; subjects of both groups performed equally well in terms of correctness for the other two tasks—Pharo retains the edge in completion time.

VII. Reflections

The results indicate that Gaucho outperforms the IDE regarding the correctness of the tasks, while it is slower with respect to the completion time. Despite the preliminary nature of the results, mostly due to the low number of subjects, we believe that we can indeed question the usage of traditional IDEs as program comprehension aids.

The positive answer to the second research question, RQ2, regarding the correctness of the tasks, might indicate that an object-focused development environment such as Gaucho provides better support for performing program comprehension tasks. We believe this result derives from the following differences between Gaucho and the baseline:

The high-level views of the graphical elements in Gaucho (the shapes) helped developers to better understand the composition of the objects involved in tasks T3.1, T3.3, T4.1, and T6.1; as opposed to manually decoding the relevant information from textual class definition, and searching lists of classes and methods names from the tools of the IDE.

The direct manipulation of the shapes eased operations such as addition, in task T5.2, or navigation through the relationships between objects in tasks T3.2, T6.1, and T7.1; as opposed to textual edition of the class definitions, and the use of contextual menus acting upon a selected item in the IDE.

The unconstrained layout of the pampas allowed developers to create side by side views of shapes, hence creating their own views of the system, to correctly solve tasks T3.3 and T6.1. Using Pharo, the developer can also create side by side views of one or more tools, using windows; nevertheless in Gaucho the shapes are simpler, smaller, and more intuitive depictions of the objects than the tools of the IDE, making it easier for the developer to take advantage of the available real state, and hence compare two graphical elements.

We believe the negative answer to the first research question, RQ1, regarding the completion time of the tasks, is an indication of the lack of maturity and exposure of the interface of Gaucho compared to a traditional IDE. Gaucho is a novel environment, and the subjects were not only exposed to the tool for the first time, but also had to adapt to the usage of an object-focused metaphor. On the other hand, the traditional tools of the IDE are widely known to developers; and even though most subjects of the control group claimed to have little or no experience with Pharo, they asserted to be at least knowledgeable in object-oriented programming and Smalltalk. We interpret this in a positive light: the Gaucho UI has potential to be matured and made more user-friendly, while the rigid structure of traditional IDEs seems to have slowly exhausted the means to advance. A sign of this stalling is the vast number of Eclipse plugins which have to fight over a limited amount of screen space [24].

The subjects using Gaucho spent more time on most of the tasks because of some usability issues found in Gaucho 1.2; mostly regarding the lack of an automatic layout or non-overlapping scheme that forced developers to spend time and effort re-arranging and closing the shapes in the screen. For example the subjects using Gaucho took more time to solve the tasks T6.1 and T3.3, which involve creating side by side views of shapes. We believe this result does not reveal an inherent problem of the object-focused metaphor but it is accidental complexity introduced by the current implementation of Gaucho. We plan to solve this issue by implementing a better

layout policy scheme in the next version of Gaucho, similar to Code Bubbles [1]. Currently, Gaucho lacks the feature to open the complete system and position the shapes according to a default layout. We plan to add this feature to the next version, and perform more experiments to asses whether developers make effective use of the freedom of placement, and to better understand how does the pampas metaphor scale when viewing large programs, managing a large number of opened shapes.

Threats To Validity. The design of the tasks may have been biased towards Gaucho. To alleviate this threat we based each of the tasks on a subset of the real questions asked by developers during development sessions. This increases the chances that we devised real tasks that developers frequently solve using traditional IDES. The tasks lasted on average 150 seconds, such a short duration can indicate that the tasks were too simplistic. We based the tasks on questions pertaining to the first and second category presented in Silito's work, described as low level questions that can be quickly answered, yet realistic.

We were able to compare such short tasks accurately due to the tracking facilities of Biscuit. Biscuit keeps track of the elapsed time for each task by recording a timestamp whenever the users commences a new task and when he provides the answer (using the Biscuit widgets overlaid on top of the tool). Since the timing is done automatically, we are confident the resolution of the time measurement is precise enough.

To answer both of our research questions, we evaluated the results of the experiments regarding the correctness and the completion time of each task. A possible threat is that we omitted the effort of completing a task from our analysis, i.e., the history of every user interaction would enable us to analyze how the subjects fulfilled each task, by measuring the effort using the GOMS model [25].

Another threat is the small number of subjects who performed the experiment (8), mainly because is too small to formally evaluate the results and claim statistical significance. We plan a larger experiment, once we fix the current usability issues of Gaucho, that will aim to provide statistically significant results. Regarding the subjects, the experiment currently lacks a balanced number of subjects from academia and industry (although the experiment feature one practitioner). A better distribution of the expertise of the subjects would provide the means to analyze beginners and advanced IDE developers.

VIII. Conclusions and Future Work

We started by questioning the use of text as the primary means to present and interact with programs. We reviewed the current state-of-the-practice of integrated development environments, pointing out a number of shortcomings that affect program comprehension. We then presented an object-focused IDE named Gaucho, which aims to alleviate the shortcomings that we identified in the current crop of view-focused IDEs.

We evaluated Gaucho in the context of program comprehension by means of a preliminary controlled experiment with 8 subjects, based on a set of common comprehension tasks. We found that users of Gaucho were on average more correct, but slower, than users of a more conventional IDE; usability issues were identified as a primary factor for slowness. Although based on a preliminary experiment, our findings point out a suboptimal and stalling situation regarding traditional IDEs. While we investigated the context of program comprehension, the discussion is pertinent at all levels. In short, the time may be right for developing new solutions from the ground up.

Acknowledgements. The subjects of the experiment (University Of Chile), the Swiss Science foundation (SNF Project No. 129496, "GSync"), the European Smalltalk User Group (www.esug.org), and the Swiss Group for Object-Oriented Systems and Environments.

References

[1] A. Bragdon, S. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr., "Code Bubbles: Rethinking the user interface paradigm of integrated development environments," in *Proceedings of ICSE 2010*. ACM, 2010, pp. 455–464.

[2] V. Sinha, D. Karger, and R. Miller, "Relo: Helping users manage context during interactive exploratory visualization of large codebases," in *Proceedings of ETX 2005*. ACM, 2005, pp. 21–25.

[3] D. Roethlisberger, O. Nierstrasz, and S. Ducasse, "Autumn leaves: Curing the window plague in ides," in *Reverse Engineering, 2009. WCRE '09. 16th Working Conference on*, 2009, pp. 237 –246.

[4] E. Chikofsky and J. Cross, "Reverse engineering and design recovery: A taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13–17, Jan. 1990.

[5] G. Weinberg, *The Psychology of Computer Programming*, silver anniversary ed. Dorset House Publishing, 1998.

[6] J. Sillito, G. C. Murphy, and K. D. Volder, "Questions programmers ask during software evolution tasks," in *Proceedings of FSE-14*. ACM Press, 2006, pp. 23–34.

[7] C. Myers and E. Baniassad, "Silhouette: visual language for meaningful shape," in *OOPSLA 2009*, ser. OOPSLA '09. ACM, 2009, pp. 917–924.

[8] A. Cooper and R. Reimann, *About Face 2.0 - The Essentials of Interaction Design*. Wiley, 2003.

[9] R. B. Smith, J. Maloney, and D. Ungar., "The self-4.0 user interface," in *OOPSLA '95*, October 1995, pp. 47–60.

[10] F. Olivero, M. Lanza, and M. Lungu, "Gaucho: From integrated development environments to direct manipulation environments," in *Proceedings of FlexiTools 2010 (1st International Workshop on Flexible Modeling Tools)*, 2010.

[11] C. Ghezzi and M. Jazayeri, *Programming Language Concepts*, 3rd ed. Wiley, 1997.

[12] D. H. Ingalls, "Design principles behind smalltalk," *BYTE Magazine*, vol. 6, no. 8, pp. 286–298, 1981.

[13] E. L. Hutchins, J. D. Hollan, and D. A. Norman, "Direct manipulation interfaces," *Human-Computer Interaction*, vol. 1, no. 4, 1985.

[14] O. Nierstrasz and T. Gîrba, "Lessons in software evolution learned by listening to smalltalk," in *Proceedings of SOFSEM 2010*. Springer.

[15] R. Wettel, M. Lanza, and R. Robbes, "Software systems as cities: A controlled experiment," in *ICSE 2011*, 2011, p. to be published.

[16] M. Kersten and G. C. Murphy, "Mylar: a degree-of-interest model for ides," in *Proceedings of AOSD 2005*. ACM, 2005, pp. 159–168.

[17] D. J. Barnes and M. Kölling, *Objects First with Java*, fourth edition ed. Prentice Hall / Pearson Education, 2008.

[18] S. Cooper and W. Dann, "Teaching objects-first in introductory computer science," *SIGCSE 2003*.

[19] B.-W. Chang, D. Ungar, and R. B. Smith, *Getting Close to Objects: Object-Focused Programming Environments*. Prentice-Hall, 1995.

[20] R. DeLine and K. Rowan, "Code canvas: Zooming towards better development environments," in *Proceedings of ICSE NIER 2011*, 2011.

[21] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker, *Pharo by Example*. Square Bracket Associates, 2009.

[22] M. Petre, "Why looking isn't always seeing: Readership skills and graphical programming," *Communications of the ACM*, vol. 38, 1995.

[23] F. Olivero, M. Lanza, and R. Robbes, "Lumiére: A novel framework for rendering 3d graphics in smalltalk." ACM Press, 2009, pp. 20–28.

[24] A. J. Ko, H. Aung, and B. A. Myers, "Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks," in *ICSE 2005*. ACM, 2005, pp. 126–135.

[25] J. Raskin, *The humane interface: new directions for designing interactive systems*. ACM Press/Addison-Wesley Publishing Co., 2000.