Science of Computer Programming 98 (2015) 394–407

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Object-focused environments revisited

Fernando Olivero *, Michele Lanza, Marco D'ambros

REVEAL @ Faculty of Informatics — University Of Lugano, Switzerland

ARTICLE INFO

Article history:
Received 27 March 2012
Received in revised form 22 July 2013
Accepted 22 July 2013
Available online 6 August 2013

Keywords:
Object-oriented programming
Development environments

ABSTRACT

Developers write Object-Oriented programs using numerous tools that come as part of integrated development environments (IDEs). We focus on the tool based interfaces of a dynamic class-based language named Smalltalk. Smalltalk IDEs have remained the same for almost 30 years now, despite that they have been found to induce problems related to navigation and the loss of context. The tools work on a textual representation of a program: the source code, which makes it more difficult to comprehend and manipulate the system under construction. In reaction to that, researchers have proposed building IDEs around other metaphors. We explore the desktop metaphor applied to Object-Oriented languages in the form of an *object-focused environment*, and provide a detailed description of our working prototype, named Gaucho. Our goal is to depart from IDEs with tool based interfaces and fancy text editors, towards an environment that eases the interaction and the crafting of objects by providing more concrete means of manipulation within the interface.

1. Introduction

The art of programming was born more than 150 years ago, when Ada Lovelace described how the Analytical Engine could be programmed to solve a mathematical problem. Programming languages evolved from machine-level instructions to more abstract languages, such as Object-Oriented languages. The communication means evolved from using punch cards to input programs into the early digital computers, and later to the use of text editors.

Nowadays developers write, test and debug programs within Integrated Development Environments (IDEs), which include many tools that provide the means to effectively construct programs. We focus on Object-Oriented development environments; we are particularly interested in those which are dynamic and include classes, therefore we chose to conduct our research in Smalltalk [1].

The Smalltalk-80 system introduced the IDE concept almost 30 years ago, coupling the language to a live environment composed entirely by objects which behave by sending each other messages. From the original Smalltalk environments developed at Xerox PARC during the late 1970s, to modern versions such as the open-source Pharo IDE,[1] a Smalltalk IDE includes numerous tools. The environment provides tools such as the inspectors to visualize the state of any object in the environment, and a set of diverse browsers enable navigating and editing the classes, methods and other objects that specify programs in the system; see Fig. 1.

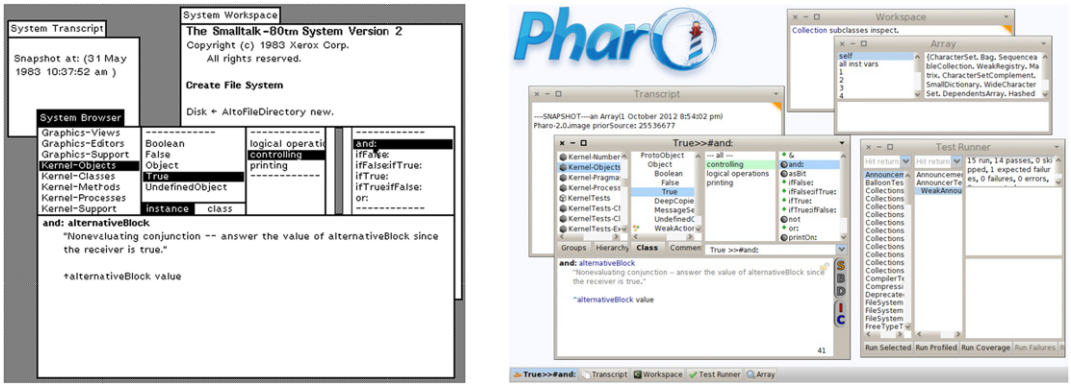
The user interface of the Smalltalk environments use a tools metaphor [2], because developers visualize and interact with tools for editing and manipulating programs and objects. For instance, to rename a class the developer must open a system browser, locate the class, and perform the change using the text editor within the browser.

We claim that the use of a traditional Smalltalk IDE may be counterproductive in the context of OOP, because the objects remain hidden behind the tools, which treat them as text when it comes to performing changes.

* Corresponding author. Tel.: +41 58 666 4758.
E-mail address: fernando.olivero@usi.ch (F. Olivero).

[1] http://www.pharo-project.org/home.

0167-6423/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.scico.2013.07.011



(a) Smalltalk-80, 1980. (b) Pharo, 2012.

Fig. 1. Smalltalk IDEs: the past and the present, from Smalltalk-80 to Pharo.

We are investigating the application of a different metaphor to the construction of user interfaces of class-based Object-Oriented languages, the desktop as opposed to the tools metaphor. An environment that makes use of a desktop metaphor replaces the tools with directly manipulable graphical elements that represent the entities of the system, and provide the means to fully manipulate them. If the latter is specifically designed for an Object-Oriented language, it becomes an *object-focused environment*, which minimizes the usage of tools compared to traditional Smalltalk IDEs.

When it comes to programming, tools only aid the human thought process [3], meaning that no tool will solve the complexities of the overall programming process. Nevertheless, with our work we want to alter how developers program in class-based Object-Oriented languages, to discover new ways of improving the traditional tool based interfaces. In this article we present our vision and its initial embodiment, named *Gaucho*, an object-focused environment for class based Object-Oriented languages.

Structure of the paper. In Section 2 we motivate our work and describe our vision for a next generation environment. In Sections 3–6 we detail the prototype implementation of our vision, named Gaucho. We present a preliminary study we performed in the context of program comprehension in Section 7. We discuss the related work in Section 8, and then conclude the paper in Section 9.

2. Motivation

Smalltalk-80 [2] was the precursor of the modern object-oriented IDE, which introduced many of the tools that are still in use today, such as object inspectors, debuggers, and code browsers. The tools are system windows that populate a 2D surface, and grant access to the objects. Smalltalk IDEs have been found to present problems when navigating the system, mostly because the various windows are distributed in a huge space, and are generic enough to display many different kinds of objects according to its current selection. The resulting workspace often becomes crowded with many opened windows, given the amount and diversity of relevant objects a developer needs to examine while fulfilling a task.

The window plague can be mitigated by an automatic scheme that finds and removes unused (or unwanted) windows [4]. Nevertheless, navigation problems remain because of the generic purpose of the tools: the selection can be modified, thus losing overall contextual information gained through persistent placement, and making it difficult to visualize relationships amongst the displayed objects.

Mainstream IDEs, such as Eclipse and Visual Studio are *file-based*, because they make explicit that source code is stored in operating system files. The file-based view of the system induces a number of problems, for instance that such a view results in navigation overhead when the structure of a task is not aligned with the structure of the system [5]. Furthermore, when researchers analyzed how developers work when solving programming tasks, they found that they work on individual code fragments, namely methods and classes [6], and that a file-based IDE hampers the developers need for manipulating entities at a finer-grained level than files within the IDE [7].

As we previously stated, in this paper we investigate an IDE built around the Desktop metaphor, because we believe it has the potential of overcoming some of the problems present in modern-day IDEs. In the following we describe our inspiration and vision for a next generation environment.

2.1. Object-focused environments

Self [8] was the pioneering Object-Oriented environment to adopt a desktop metaphor. The term *object-focused environment* was coined by Ungar et al., the researchers behind the Self programming language [9]. The environment fosters the notion that developers are in direct contact with the objects themselves, by coupling operations and representations

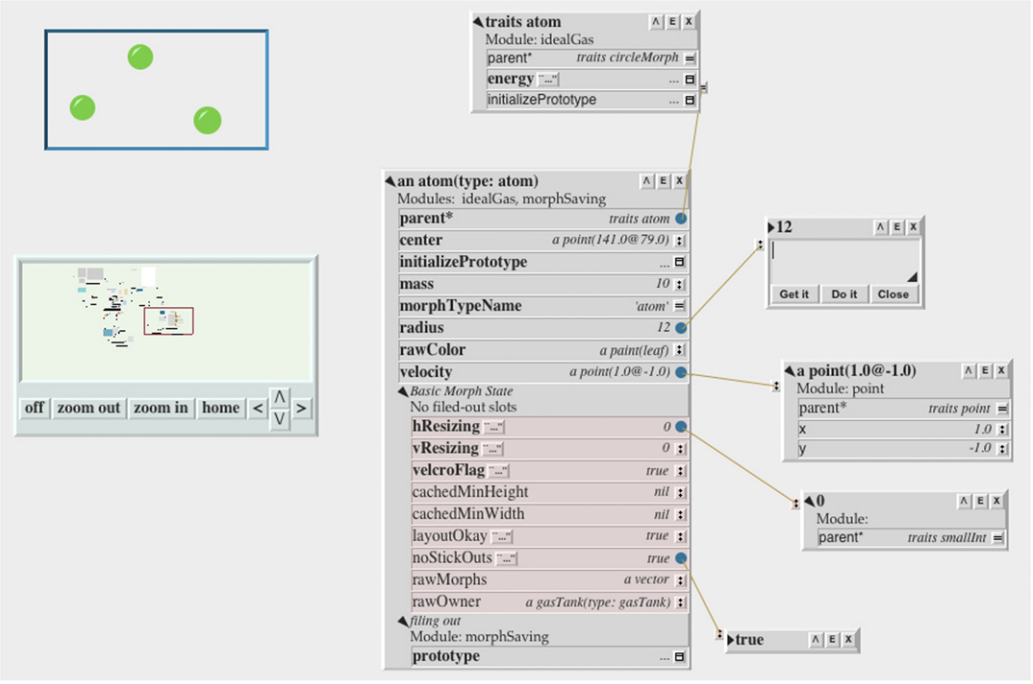


Fig. 2. Self: the seminal object-focused environment.

into the same graphical element, removing the need for tools in the interface. In Self, the programming tasks take place through direct manipulation of graphical elements that represent the objects, see Fig. 2.

Self is an Object-Oriented language based on prototypes instead of classes and instances, the language makes fewer distinctions between all the objects of the system. The uniformity permeates throughout the whole interface, because a Self object has a single tool that reveals its inner structure, i.e., its slots, and provides means to visualize and modify them, the tool depicted at the center of Fig. 2.

2.2. Object-focused environments reloaded

Self is a prototype-based language that deliberately omits the concept of classes and instances. In our work we focus on class-based languages, thus we must augment object-focused environments to include many different kinds of graphical elements: ideally one per each kind of object in the system. The object-focused environment we envision includes graphical elements tailored for diverse objects as classes such as meta-classes, methods, packages, tests, code-critic rules, etc.

Ingalls emphasized the reactive principle as one of the design principles behind Smalltalk [10], which states that *every component accessible to the user should be able to present itself in a meaningful way for observation and manipulation*. The latter is the main reason why traditional Smalltalk environments include inspectors: tools that visualize the state of any object, and enable developers to perform changes by sending messages to it by typing in an embedded code pane. Smalltalk, as opposed to Self, includes more than one kind of tool, in the form of browsers which are specialized inspectors for all the objects that are relevant to programming tasks. The set of diverse browsers are customized for viewing and changing classes, methods and packages. For instance, Fig. 3 depicts both an inspector and a system browser viewing the same object, the class *String*.

We stated previously that browsers provide great functionality at the expense of some problems related to navigation, which can be mitigated by providing finer-grained *tools* to manipulate the objects relevant to programming in Object-Oriented languages. Furthermore, browsers are optimized for writing and browsing textual representations of the objects, namely the source code, a raw text description of the program elements. Using browsers, most of the programming activities are performed by means of textual editions to the source code. The instrumentation of all modification operations relies mostly on textual editions, which might leave the represented objects in an inconsistent state. Although this problem can be amended by using code completion, reading the compiler warnings and hints provided by the tools to fix the syntactic problems, the next-generation environment we envision includes the use of custom widgets which reify diverse operations such as renaming a class, adding or removing methods and variables, changing the superclass of a class, and running tests.

In our environment we want to minimize the presence of intermediaries—the tools—in favor of directly manipulable representations of objects, which react to the developer actions in a uniform manner and provide all the features to successfully modify them. Next, we detail the environment we are building to support our vision: Gaucho.

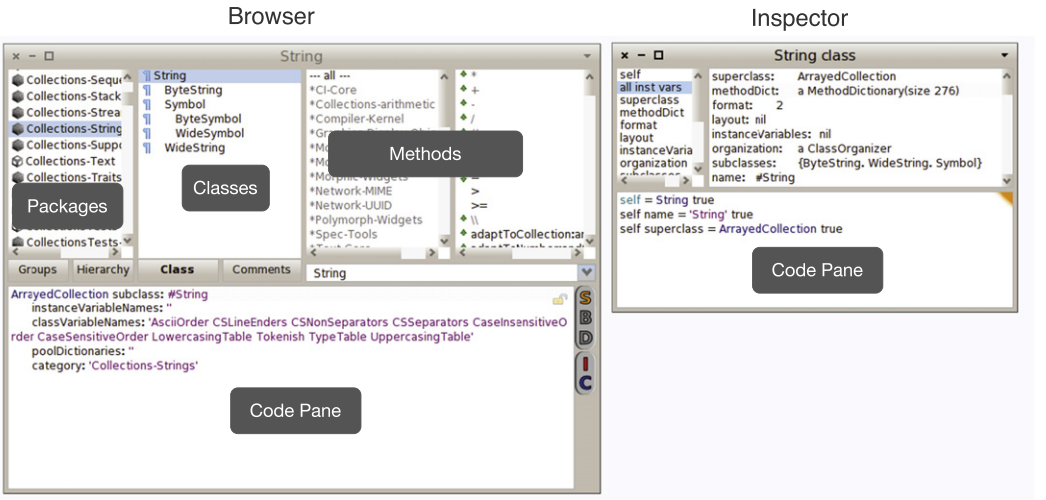


Fig. 3. Smalltalk: inspecting and browsing the class String.

3. Gaucho

Gaucho is the prototype tool we are building to support our ideas and the ongoing research towards an object-focused environment. 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 objects.

3.1. Gaucho in a nutshell

Gaucho [11] is an object-focused programming environment for the crafting of models composed of objects, described in class-based Object-Oriented languages. The three pivotal concepts in Gaucho are:

1. *Pampas*: a 2D surface hosting one or more *v*iews.
2. *Views*: a pannable and zoomable region of the pampas presenting a scene composed of *s*hapes.
3. *Shapes*: high-level graphical elements representing objects that make up the software system (e.g., instances, classes, methods, packages, test cases, and messages).

Gaucho uses a zoomable user interface (ZUI), i.e., the views that populate the pampas can be panned and zoomed, enabling developers to fully customize different perspectives on the system under construction. The top-left view depicted in Fig. 4 includes a pampas menu which enables the creation and deletion of up to four visible views. The objects may be opened (and closed) on demand by simply typing their names in the pampas, enabling programmers to filter those which are relevant to the task at hand.

Gaucho enables the crafting of models composed of many different kinds of objects, such as classes, methods, packages, test cases, meta-classes, instances, messages, etc. Gaucho is an object-focused environment, where every distinct object has its own graphical counterpart, a directly manipulable element that uniquely identifies the object within the interface. Fig. 5 depicts several Gaucho shapes that represent classes, methods, test-cases and packages of the system under construction. For example, the figure includes a *class shape* depicting the structure of the represented class, and a *method shape* presenting the method signature and the statements that make up the body, i.e., the source code.

3.2. Task context support

Developers start developing by finding initial focus points, and then continue expanding those points forming interconnected graphs [14]. We argue that the traditional view-focused tools of mainstream IDEs hinder navigating the system, given their text-, list-, and tab-based nature. For instance, 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 different parts of the system [5], producing back and forth navigation within the tools [7].

In Gaucho, developers create their own view of the system, by opening the shapes relevant to the task at hand. Gaucho does not automatically collect the relevant entities, as in Mylin's degree-of-interest model [5], but enables a developer to incrementally populate an infinite surface, which hosts a subset of the complete system. A view in Gaucho represents an arrangement of shapes which form a scene. Scenes are viewed from a particular viewpoint, named the scene window, which dictates the amount of panning and zooming.

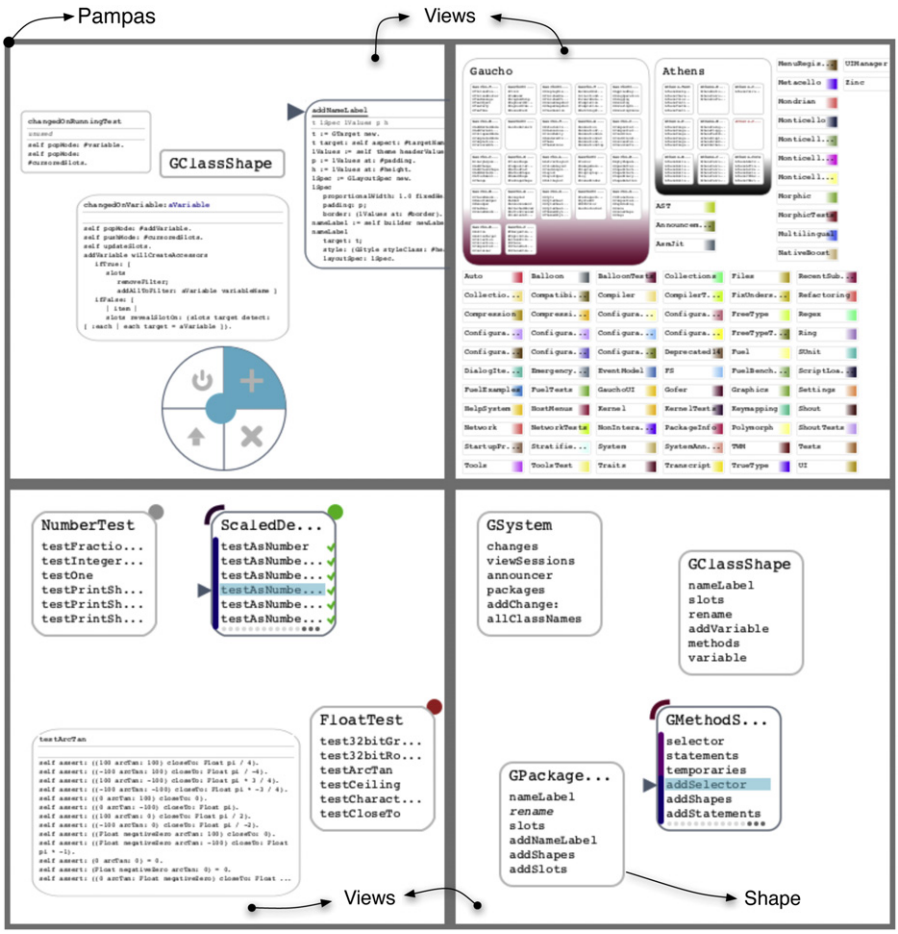


Fig. 4. Gaucho: an object-focused environment.

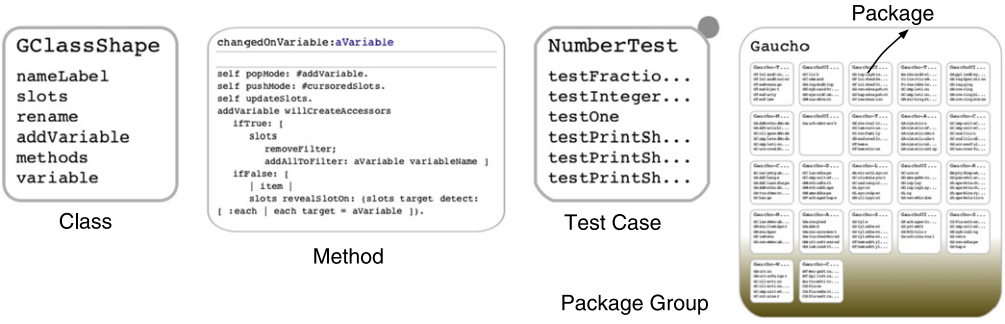
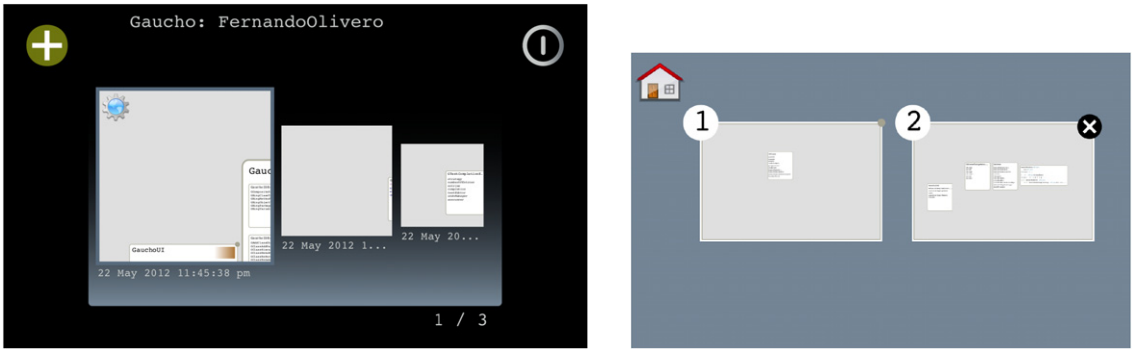


Fig. 5. Distinct Gaucho shapes populating a pampas.

Fig. 6 depicts Gaucho on startup when developers choose to continue working on past development sessions or commence a new one, and the widget that enables managing the created views. The views are persisted throughout the development sessions, thus preserving the task context defined by an arrangement of shapes making use of spatial memory and reasoning. The views are saved to disk and can be re-opened in future development sessions.

4. Programming in Gaucho

Gaucho includes several different shapes to represent the classes, methods, packages, and test cases of the system. In the following, we describe them in more detail, and discuss the interactions that make possible changing the system using Gaucho.



(a) Login by opening a saved session. (b) Manage the views within a session.

Fig. 6. Task context support within Gaucho.

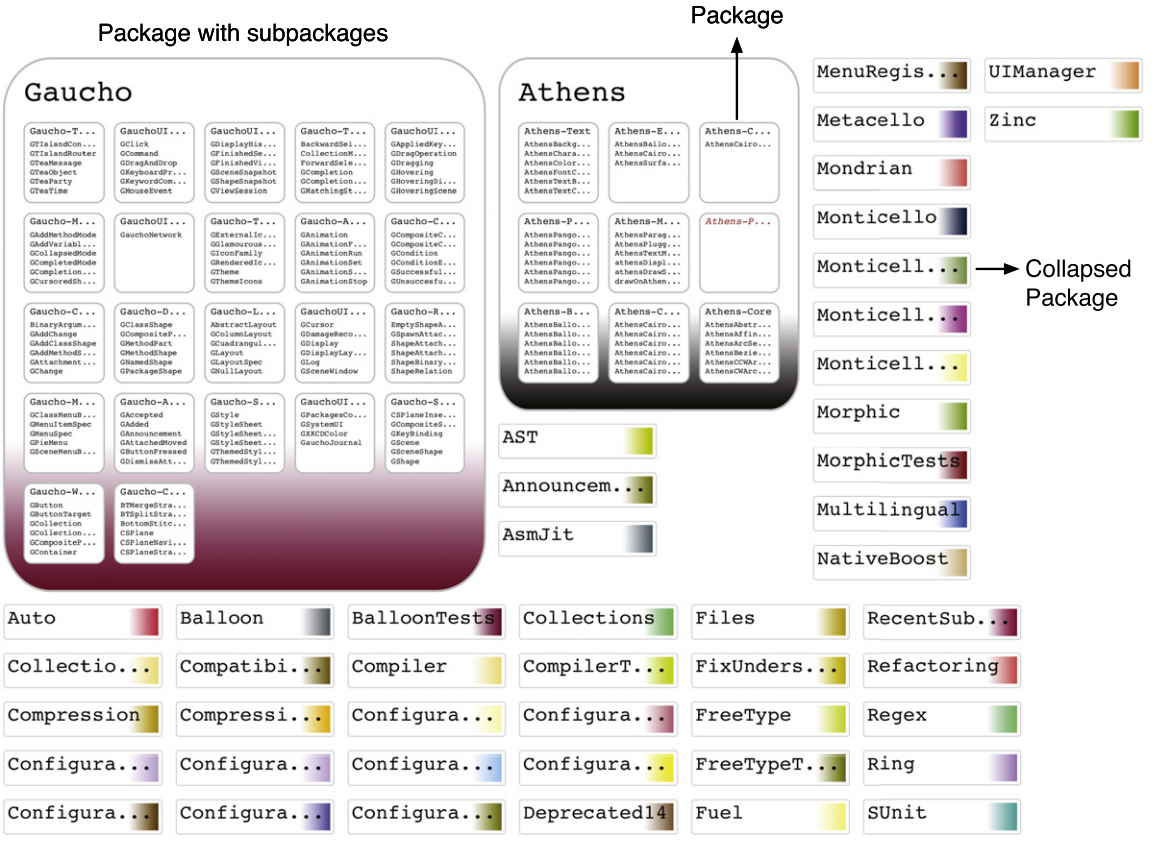


Fig. 7. The system view presents all the available packages.

4.1. Packages

In Gaucho, a system view grants access to all the packages of the system, for performing manipulations such as renaming, adding or removing packages, and the repackaging of classes (Fig. 7).

The system view initially lays out all the package group shapes of the system using a rectangle packing algorithm to optimize the use of real estate, and enables a developer to move, collapse or expand any package, customizing the overall view of the system. In Gaucho, all the packages of the system are uniquely identified by a color used throughout the interface, such as the colored top left visual arc that denotes the package of a class shape (cf. Fig. 8). The coloring scheme and the laying out of all the available packages in a pannable and zoomable surface eases the location of packages and their classes. Although we are aware that such a coloring scheme will inevitably fall short when it comes to dealing with large systems, we are considering other alternatives, based on a more hierarchical scheme.

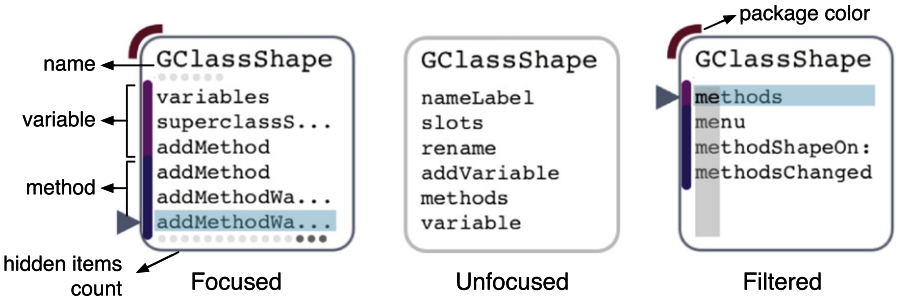


Fig. 8. The shape representing the class shape itself. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

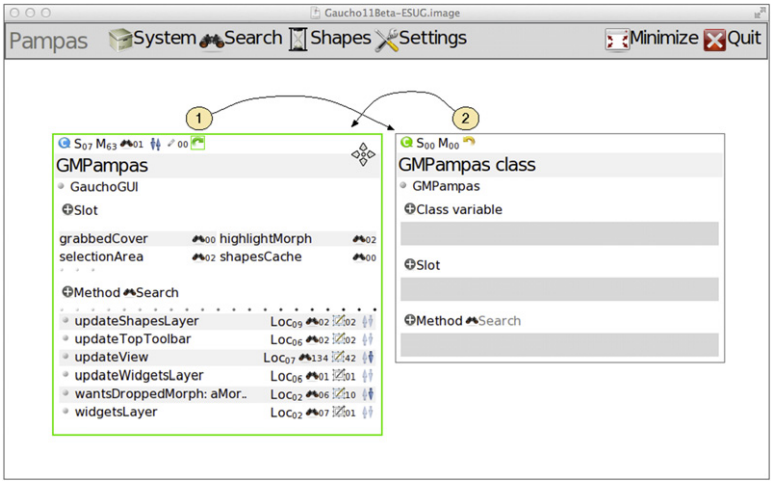


Fig. 9. Class and Meta-Class Toggle.

4.2. Classes

A **Class Shape** is the representation of a class in our object-focused environment, thus the shape must present for manipulation all the elements that make up the underlying class. Class shapes are composed of an editable label widget that displays the class name, and a scrollable list widget that enumerates all the variables and the methods of the class. Fig. 8 depicts a class shape on the class named *GClassShape*. The shape is presented in three different modes: focused, unfocused, and filtered.

The focused and filtered shapes depicted in Fig. 8 include several examples of overlays that convey extra information. The colored left toolbar denotes the type of each list item (i.e., magenta for instance variables, gray for class variables, and dark blue for methods). The arc to the top left indicates the unique color of the package the class belongs to.

Given that a class may contain a large amount of methods and variables, the class shape can be filtered to reveal only those items that match a regular expression, narrowing down the displayed items to ease the location of the class components. For example, Fig. 8 depicts a filtered class shape showing those methods and variables that match the pattern *me**. The filtering mode is also denoted by a vertical grayed bar, which spawns the entire contents of the filtering string.

Differently than the previous version of Gaucho (see Fig. 9), we designed the appearance of a class shape to concisely present a high level view of a class. We strive to present a condensed visualization of class, to ease the comprehension of its composition, therefore the shape only reveals extra information on selection or when it is the focus of attention.

In this version of Gaucho, for the purpose of simplicity we omitted some of the features included in the traditional Smalltalk environments, such as the grouping of the methods of a class into different categories. We plan to augment the class shape to enable method categories, and other properties of the class such as comments and class variables, drawing inspiration from a previous version of Gaucho. The former version enabled toggling between a class and its meta-class shape, to add or remove class side variables and methods. Fig. 9 illustrates how the toggle button of a class shape of the previous version of Gaucho triggered a flip-over animation towards its meta-class shape (1), and also enabled returning to the class shape (2).

4.3. Methods

In class-based Object-Oriented languages, a class defines the behavior of its instances by means of methods. Methods specify how instances of a class answer a request, in the form of a message, made by an object, i.e., the sender of the message.

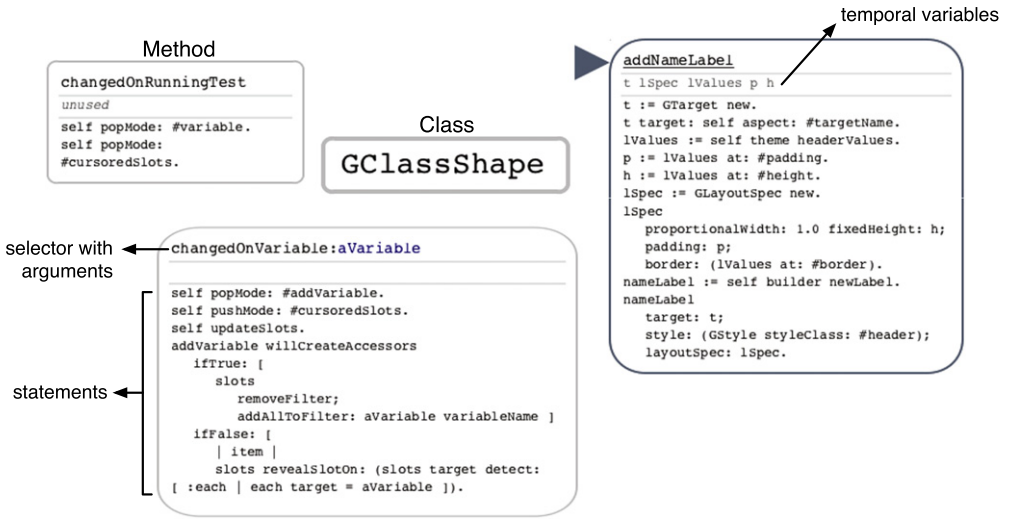


Fig. 10. A collapsed class, with several of its methods surrounding the shape.

Fig. 11. Test Case shapes with different test results.

Methods are composed of a selector that describes the signature of the method, a collection of temporal variables which have a local scope within the method, and a collection of valid expressions of the language, which together form the body of the method.

This composition is recurrent in most Object-Oriented languages, therefore is the definition of a method used in Gaucho. Fig. 10 depicts three method shapes, surrounding the collapsed shape of the class they belong to.

A **Method Shape** includes a custom selector shape, which presents the selector and the arguments of the method, a row with all the temporal variables, and an editable text for editing the body of the method. Methods are spawned from a class shape by creating a selection in the list widget, and pressing a keyboard command. The containment relationship is conveyed by attaching the methods to the class whenever the class shape is dismissed or moved to another location.

4.4. Testing

The SUnit framework introduced testing into the software development process of Object-Oriented languages [12]. Since then it has become common practice to devise unitary tests that assert the correct implementation of a certain functionality, i.e., a collaboration amongst objects.

Class-based Object-Oriented languages generally include a special kind of class, named *TestCase*, which includes test cases in the form of methods whose names start with the word *test*.

Gaucho supports creating and running test cases, by means of manipulating a special kind of shape, a **Test Case Shape**, a specialized class shape augmented with extra visual cues to denote the status of test cases. Fig. 11 depicts three Test Case shapes with different test results: untested, failing, or passing. Typing a custom keyboard command on a focused test case shape runs the test case, and the shape reflects the last run status.

4.5. Navigating the system

Researchers have found that direct tool support for interactive exploration helps to perform program comprehension tasks [13]. According to a study by Sillito et al. on developers' habits [14], developers start programming sessions by finding initial focus points, and then continue expanding those points by exploring relationships between the software artifacts.

Therefore, in Gaucho, the shapes include a customized set of navigation actions, invoked by menu items or keyboard shortcuts that provide quick access to exploring the connected artifacts. For example, a class shape provides means to spawn

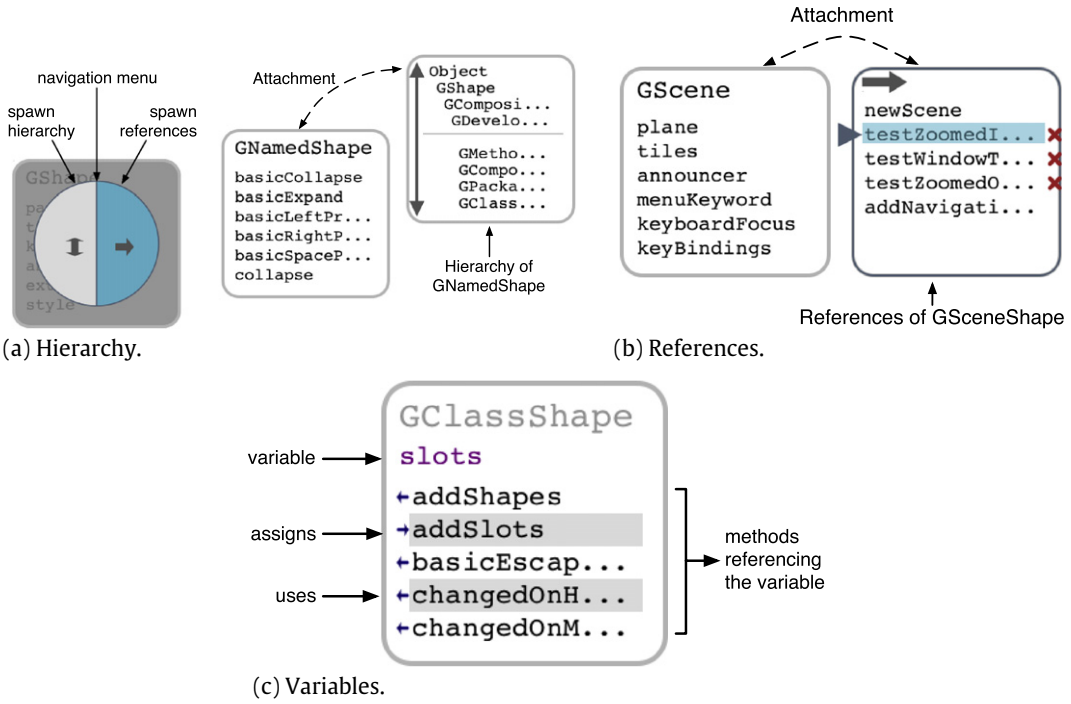


Fig. 12. Navigating the system from a class shape.

both the group of class references and the class hierarchy, which are group shapes that enable further exploration within the system. Fig. 12(a) and (b) depict the result of spawning the hierarchy and the references of a class shape. The leftmost figure presents a class shape with a navigation menu with two options: spawn the references and open the class hierarchy.

IDEs include query mechanisms for exploring the system, which however often open views that clutter the interface and induce context loss. Gaucho favors the preservation of the context because it provides a more focused system exploration: the shapes represent concrete objects as opposed to general purpose tools or tabs.

Moreover, to preserve the relations amongst the shapes in the interface, the spawned shapes in Gaucho are attached to the original shape. The attachment forces any subsequent move or a dismiss of the latter will be applied to the former.

In traditional view-focused tools, most of the information is presented by spawning more tools, distancing the information from the current focus of attention. An object-focused environment enables a more focused query and display of information related to shapes representing concrete objects of the system. For instance, a class shape can be asked to reveal all the methods that assign or use a specific variable (see Fig. 12(c)).

We plan to enhance the navigation features provided by the shapes to automatically present dependences between related objects. A study by Ko et al. found that this missing functionality would reduce the navigation overhead when interacting with the development environment [6].

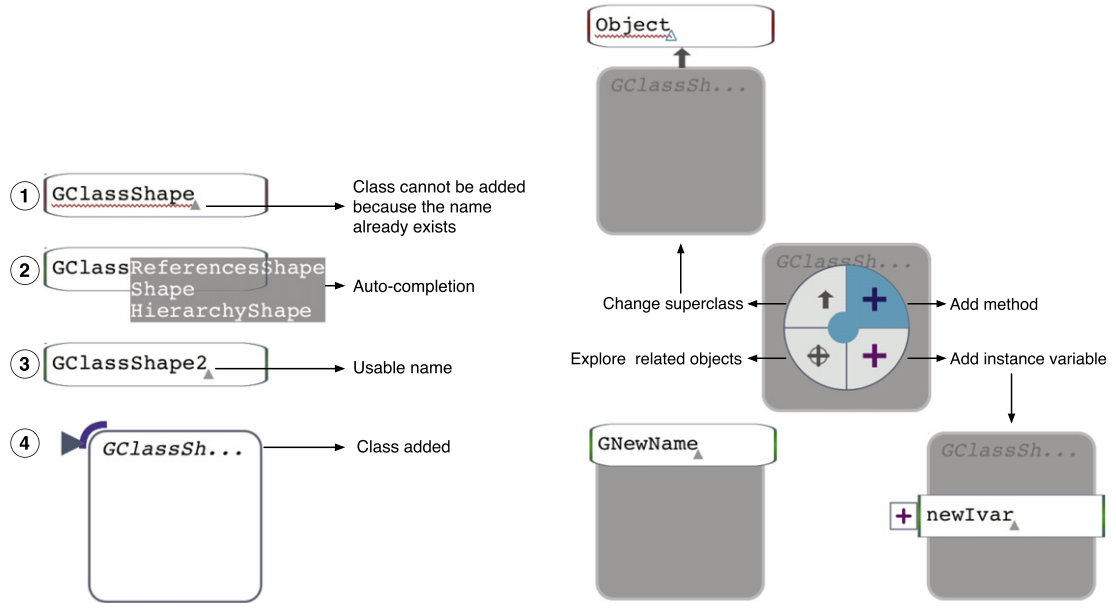
5. Producing system changes with Gaucho

In Gaucho the burden of the UI is lessened because a developer interacts with a class, a test case, a method and a package in a uniform and consistent manner. The shapes behave in a consistent way for adding, removing or modifying the objects, based on a simple set of keyboard commands, and mouse interactions. The use of direct manipulation of graphical elements that represent the objects enables Gaucho to attain a higher-level of interaction, as opposed to the view-focused tools of traditional IDEs where to change the superclass of a class, or add methods, one must locate the proper source code fragment, and then edit the textual representation of the class definition, instead of performing changes with meaningful semantics.

5.1. Classes

In Gaucho, developers interact with a **Class Shape** to rename classes, modify the superclass and adding variables and methods, by opening a custom set of widgets which are easily accessed by keyboard commands or the menu. Fig. 13 depicts a typical scenario of class creation in Gaucho, where the developer engages in collaborations with the pampas and the class shape itself to fully manipulate the newly created class.

Both the keyboard commands and the menu options are a small set of interactions that the user must learn. Gaucho provides a more focused interaction than with general purpose tools, thus the user quickly gets used to the mouse and



(a) Adding a class. (b) Modifying a class.

Fig. 13. Manipulating classes with Gaucho.

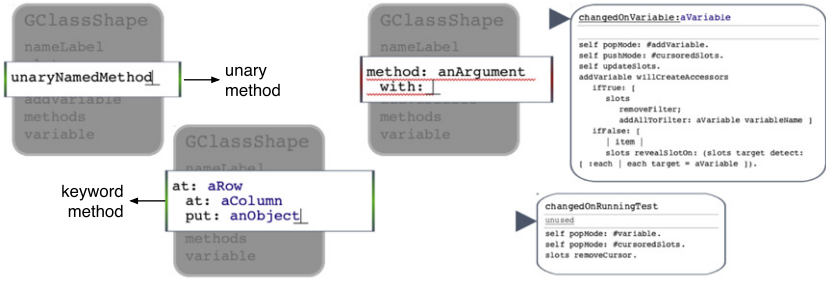


Fig. 14. Adding and modifying methods.

keyboard actions for manipulating the shapes. For the same reason, we chose to use pie menus (see Fig. 13(b)) instead of traditional list-based menus, because when the set of operations is small, they ease pointing and selecting menu options [21].

To add a new class, the developer focuses on the pampas, then presses a keyboard command to (1) open the *add class widget*. The widget helps the developer in typing the new class name (2) by auto-completing on demand (pressing another keyboard command), and updating the widget with visual cues that denote whether the currently typed class could be added. Once the developer types in a valid new class name (3) by pressing enter an empty class is created, and the shape is opened, replacing the widget (4). Fig. 13(a) illustrates the enumerated steps.

5.2. Methods

To add a new method to a class, the developer opens the *add method widget* by pressing a keyboard command or invoking the menu item. The widget automatically re-arranges itself according to the signature of the inputted method, by formatting the code and adding or removing lines. Methods are removed by selecting them in the list of the class shape they belong to, and pressing a keyboard command (i.e., cmd-delete).

To add/remove/modify a temporal variable, an argument or the method body, the developer gives focus to the desired method part by moving the cursor, and then presses enter to enter the edit widget.

Fig. 14 depicts two method shapes with the cursor placed on the selector, to perform a rename method refactoring, and on a temporal variable, to perform a rename/remove refactoring.

5.3. Refactorings

Refactoring is a technique for restructuring an existing body of code, altering its internal structure without changing its external behavior [15]. The concept was introduced by Fowler, and since then many tools have been devised for aiding the developer when performing refactoring tasks [16].

Gaucho enables developers to perform several class and method refactorings, by directly manipulating the relevant shape as in the case of class rename, or by performing visual operations as in the case of a class repackaging or a method move by dragging and dropping a shape. For instance, removing a variable can only be performed when the variable is not referenced anymore, therefore attempts to select and remove a used variable, will focus the class shape on the uses and assignments perspective (cf. Fig. 12).

6. Implementation

Gaucho is written in Smalltalk [1], and implemented on top of Pharo.[2] Gaucho uses Cairo,[3] a vectorial rendering library written in C. We implemented all the infrastructure for handling the mouse and keyboard events, the drawing of the display, the zooming and panning in the interface, a non-overlapping scheme for laying out the shapes, and a complete set of custom widgets. Although building the complete infrastructure from the ground up was a huge undertaking, it empowered us with full control for exploring ideas in the interface, which is not possible when using a standard toolkit or a general purpose user interface framework. Gaucho can be downloaded from the Gaucho web site located at gaucho.inf.usi.ch.

Smalltalk is a highly dynamic and fully reflective language, where everything is an object: the classes, meta-classes, packages, and methods are first class objects. The uniformity of the language enables all the changes to be done by sending messages to the objects via their graphical counterparts, i.e., the shapes. The user interface of Gaucho could be implemented in "non-dynamic languages" such as Java and C#, at the expense of a loss of dynamism when manipulating the shapes. The compile-time and run-time distinction hampers the liveness of an object-focused interface, and results in a more complex implementation of the environment; mostly because the classes, methods, and packages are not *live* objects of the system.

7. Evaluation

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"* [18]. 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 integrated development environments. Modern 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. To assess the validity of this claim, we performed a controlled experiment and present our findings in this section.

Design. We evaluated Gaucho in the context of program comprehension by means of a controlled experiment with 8 subjects, equally divided in two groups. The subjects of the control group used a traditional Smalltalk IDE named Pharo, and the ones belonging to the experimental group used Gaucho. An experiment run consisted in a subject solving 14 tasks based on the set of real questions asked during development activities, extracted in a study performed by Sillito et al. [14]. The tasks were targeted towards *Lumière*, a framework for creating and rendering 3-D scenes, consisting of 10 packages, 139 classes, and 1741 methods, for a total of 9493 lines of code. For example, one of the tasks involved locating the class that represents a rotation of a scene graph in Lumière, and another requested the name of the test case class where most of the layout behavior is tested.

Instrumentation. We used an automated experiment runner toolset, called *Biscuit* [19], 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. Fig. 15 shows Biscuit running on top of Gaucho.

Subject and expertise analysis. During the experimental session, in the context of a Biscuit task, the subjects answered questions related to their expertise. The results evidence that the subjects of the control group have little or no experience using the Pharo IDE. The same level of expertise can be assumed for the subjects of the experimental group, given that their first encounter with Gaucho occurred during this experiment. The subjects also evidence a balanced distribution of their expertise among treatments, regarding OOP and Smalltalk, see Table 1.

Results. The 14 tasks were graded, yielding a maximum score of 14 points, and the time taken to solve each task was measured. The subjects of the experimental group outperformed the subjects of the control group regarding the correctness of the tasks. The experimental group using Gaucho scored, on average, 10.4 points, and the control group using the Pharo treatment scored, on average, 8.5 points. Whilst the subjects of the control group outperformed the experimental group regarding the completion time. The total completion time on average of the subjects using the Gaucho treatment was 38:08 min, while the subjects using the Pharo treatment spent, on average, 28:48 min for all the tasks.

We found that users of Gaucho were on average more correct, but slower, than users of a more conventional IDE; usability issues and a lack of maturity and exposure of the interface of Gaucho were identified as a primary factor for slowness.

2 http://pharo-project.org.

3 http://cairographics.org/.

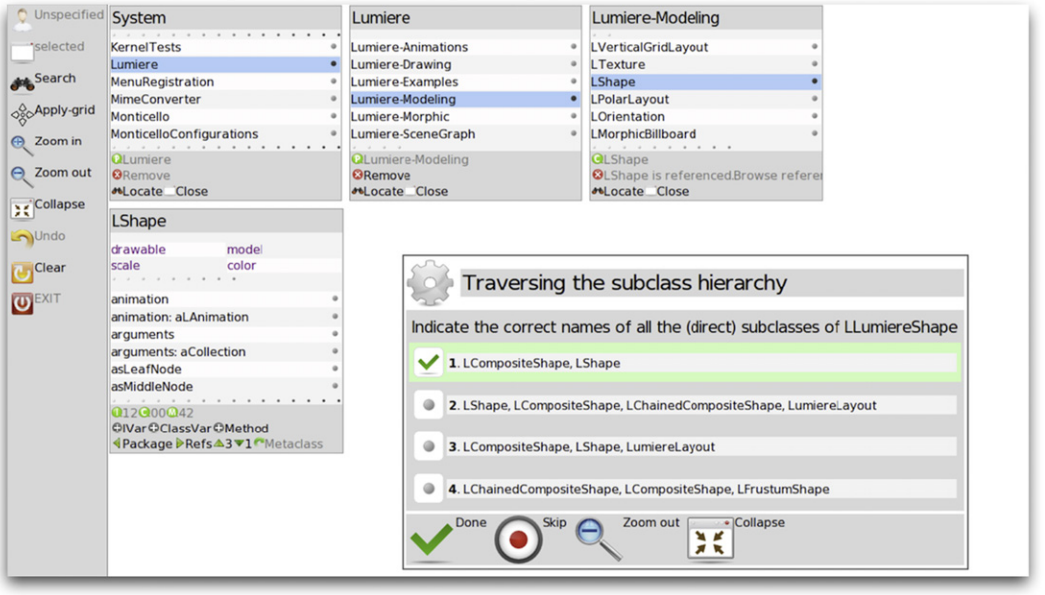


Fig. 15. Biscuit: task list and a running task example.

Table 1
The declared expertise of the subjects

Expertise	Control group			Experimental group	
	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

Reflections. The results 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 shapes helped developers to better understand the composition of the objects; as opposed to manually decoding the relevant information from textual class definition. *The direct manipulation* of the shapes eased operations such as addition or navigation through the relationships between objects. *The unconstrained layout* of the pampas allowed developers to create side by side views of shapes, easing the tasks of comparing two or more objects. Although based on a preliminary experiment, our findings point out a suboptimal and stalling situation regarding traditional IDEs. We published a more detailed analysis of the design, instrumentation, and results of the evaluation in [20].

Although the experiment was conducted with an older version of Gaucho, the tested feature-set remains mostly unchanged in the version presented throughout this paper. The changes mostly consisted in the overall appearance of the shapes, and the addition of a non-overlapping scheme to the pampas. In this evaluation the subjects using Gaucho spent more time on most of the tasks because of some usability issues; 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. We plan to perform more experiments to further analyze the novel features of Gaucho, such as a non-overlapping scheme, a more concise appearance of the shapes, and the reuse of past development sessions.

8. Related work

Researchers and practitioners proposed several programming environments that make use of alternative user interfaces. Fig. 16 depicts the ones most related to Gaucho: Self, Code Canvas, Code Bubbles, and Relo.

Self is the seminal object-focused environment [8]. 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.

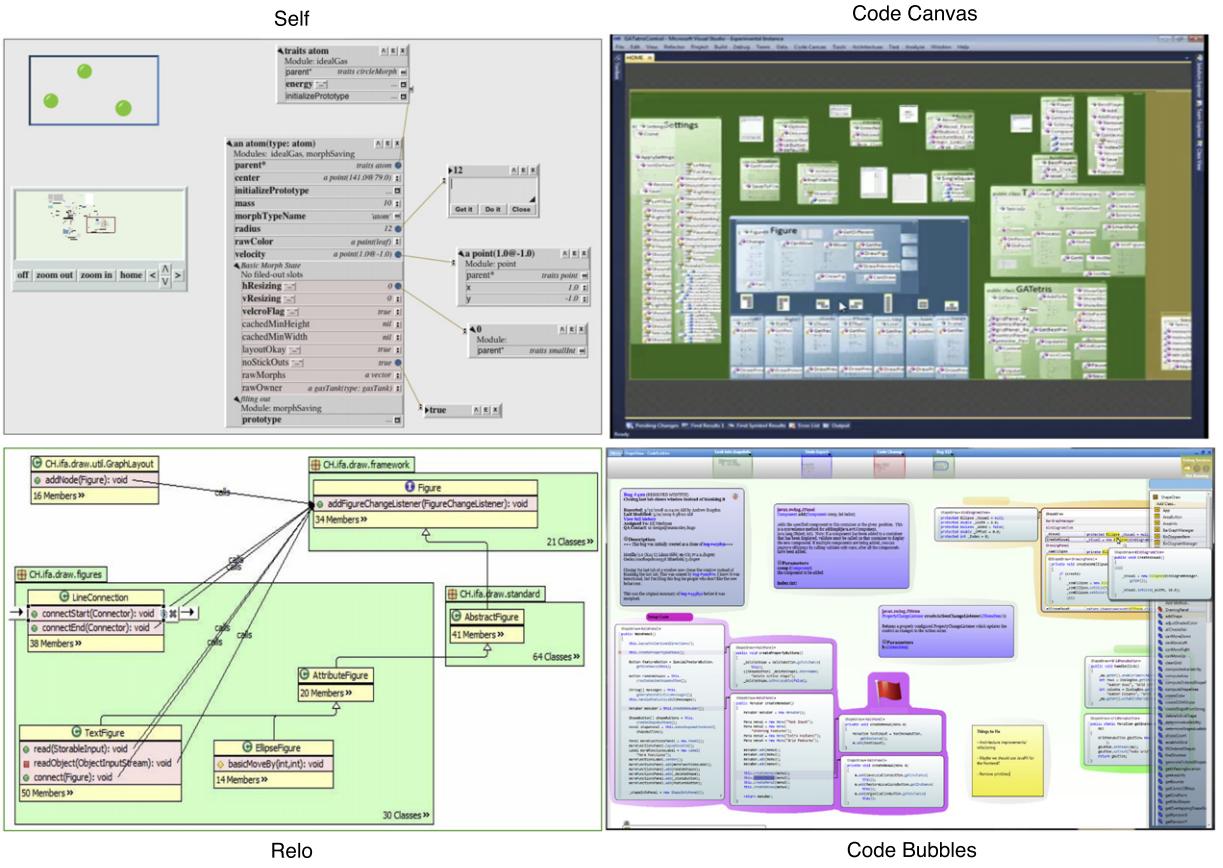


Fig. 16. Screenshots of the most related development environments.

Code Canvas provides a 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 [17].

Code Bubbles 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 [7]. Bubbles exist and are placed on a pannable 2-D space. The bubbles are interactive views of source code fragments such as a method or a collection of member variables.

Relo is a tool that supports program comprehension by enabling interactive exploration of code, interacting with simple visual representations of the software artifacts [13]. The tool is built within the Eclipse IDE. *Relo* supports program comprehension by enabling and facilitating interactive code exploration. The graphical elements depict high level views of software elements presented in a similar manner as the 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.

8.1. Inspiration and differences

Gaucho took inspiration from Self, but provides a richer set of graphical elements because it is designed for Smalltalk [10], a dynamic class-based object oriented language. In Smalltalk, different kinds of objects coexist in the world, such as classes, methods, and packages, whereas Self offers a uniform interface for programming in a prototype-based language.

Relo is designed to be embedded into the IDE, using automatic layout of the visual representations; instead in Gaucho, we provide customizable layouts and a full fledged IDE built upon a single metaphor. 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.

Code Canvas and Code Bubbles address navigation problems of mainstream file-based IDEs, using alternative user interfaces than the traditional tool based IDEs. The interface of both environments makes better use of the spatial memory and the available display 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. In Gaucho these graphical elements provide a more abstract representation than portions of source code, in the form of shapes, which represent the objects themselves.

9. Conclusions

We started by stating that nowadays Object-Oriented developers write programs using numerous tools included in IDEs; then we questioned the use of text as the primary manipulation mechanism. We enumerated some of the problems that traditional IDEs induce due to their dependence on a tool-based view of the system. We continued by describing an alternative user interface for OOP environments, one that gives prominence to objects and provides high-level views of the elements that compose a software system. Our work revolves around the adoption of an object-focused environment for class-based languages, and details how a more focused and concrete interface solves the problems related to the loss of abstraction and navigation overhead found in current tool-focused IDEs. We presented many examples of the application of such an environment, using the prototype system we built, named Gaucho. We reviewed the current state-of-the-art on the environments which are most related to Gaucho.

We concluded with a description of the preliminary controlled experiment we conducted, in the context of program comprehension, to assess the metaphor in use by Gaucho. We found that the users of Gaucho were on average more correct, but slower, than users of a conventional IDE; usability issues were identified as a primary factor for slowness. Despite the preliminary nature of the results, we can indeed question the usage of traditional IDEs, in favor of an object-focused environment.

References

[1] A. Goldberg, D. Robson, Smalltalk-80: The Language and its Implementation, Addison-Wesley, 1983.
[2] J. Johnson, T.L. Roberts, W. Verplank, D.C. Smith, C.H. Irby, M. Beard, K. Mackey, The xerox star: a retrospective, Computer 22 (9) (1989).
[3] E.W. Dijkstra, The humble programmer, Commun. ACM 15 (1972).
[4] D. Roethlisberger, O. Nierstrasz, S. Ducasse, Autumn leaves: curing the window plague in ides, in: Proceedings of WCRE 2009, IEEE Computer Society, 2009, pp. 237–246.
[5] G.C. Murphy, M. Kersten, L. Findlater, How are java software developers using the eclipse ide? IEEE Softw. 23 (2006) 76–83.
[6] A.J. Ko, H. Aung, B.A. Myers, Eliciting design requirements for maintenance-oriented ides, in: Proceedings of the 27th International Conference on Software Engineering, ICSE 2005, ACM, 2005.
[7] A. Bragdon, S. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, 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.
[8] R.B. Smith, J. Maloney, D. Ungar, The self-4.0 user interface: manifesting a system-wide vision of concreteness, uniformity, and flexibility, in: OOPSLA 1995 Conference Proceedings, 1995, pp. 47–60.
[9] B.-W. Chang, D. Ungar, R.B. Smith, Getting Close to Objects: Object-Focused Programming Environments, Prentice-Hall, 1995.
[10] D.H. Ingalls, Design principles behind smalltalk, BYTE Magazine 6 (1981).
[11] F. Olivero, M. Lanza, M. Lungu, Gaucho: from integrated development environments to direct manipulation environments, in: Proceedings of FlexiTools 2010 (1st International Workshop on Flexible Modeling Tools), 2010.
[12] K. Beck, C. Andres, Extreme Programming Explained, second ed., Addison-Wesley, 2005.
[13] V. Sinha, D. Karger, R. Miller, Relo: helping users manage context during interactive exploratory visualization of large codebases, in: Proceedings of OOPSLA Workshop on Eclipse Technology eXchange, ACM, 2005, pp. 21–25.
[14] J. Sillito, G.C. Murphy, K.D. Volder, Questions programmers ask during software evolution tasks, in: Proceedings of FSE-14 (14th International Symposium on Foundations of Software Engineering), ACM Press, 2006, pp. 23–34.
[15] M. Fowler, Refactoring — Improving the Design of Existing Code, Addison-Wesley, 1999.
[16] D. Roberts, J. Brant, R. Johnson, A refactoring tool for smalltalk, Theor. Pract. Object Syst. 3 (1997) 253–263.
[17] R. DeLine, K. Rowan, Code canvas: zooming towards better development environments, in: Proceedings of ICSE 2010, ACM, 2010.
[18] E. Chikofsky, J. Cross, Reverse engineering and design recovery: a taxonomy, IEEE Softw. 7 (1) (1990) 13–17.
[19] F. Olivero, M. Lanza, M. D'Ambros, R. Robbes, Tracking human-centric controlled experiments with biscuit, in: Proceedings of PLATEAU 2012 (4th International Workshop on Evaluation and Usability of Programming Languages and Tools).
[20] F. Olivero, M. Lanza, M. D'Ambros, R. Robbes, Enabling program comprehension through a visual object-focused development environment, in: Proceedings of VL/HCC'11, 2011, pp. 127–134.
[21] G.P. Kurtenbach, A.J. Sellen, W.A.S. Buxton, An empirical evaluation of some articulatory and cognitive aspects of marking menus, Hum. -Comput. Interact. 8 (Mar.) (1993) 1–23.