Object-focused Environments Revisited

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Fernando Olivero

under the supervision of
Prof. Dr. Michele Lanza

April 2013

Dissertation Committee

Prof. Dr. Theo D'Hondt	Vrije Universiteit Brussel, Belgium
Prof. Dr. Stéphane Ducasse	INRIA Nord, France
Prof. Dr. Marc Langheinrich	University Of Lugano, Switzerland
Prof. Dr. Oscar Nierstrasz	University of Bern, Switzerland
Prof. Dr. Cesare Pautasso	University Of Lugano, Switzerland

Dissertation accepted on April 2013

Research Advisor	PhD Program Director
Prof. Dr. Michele Lanza	**Prof. Dr. Antonio Carzaniga**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Fernando Olivero
Lugano, April 2013

Abstract

In the object oriented programming (OOP) paradigm, programs are composed solely of objects. The computational model is based on a world of collaborating objects, where they send each other messages to carry out tasks. The programs are crafted with the aid of tools, which enable to describe their components and behavior in a human readable form. With the advent of the graphical user interface came the pinnacle tool for software development, the integrated development environment (IDE).

IDEs include numerous tools to effectively construct, debug, and test the programs. The tools work on a *static* textual representation of the program – the source code– which conceptually conflicts with the *dynamic* nature of the computational model of OOP. The use of a tool-based interface also produces technical problems, which relate to navigating the system, preserving the task context, and manipulating finer grained entities than the coarse grained perspective offered by the tools.

In this thesis we investigate an alternative interface for OOP environments, which is based solely on direct manipulation of objects. It alleviates the conceptual and technical problems of tool-based IDEs, by giving prominence to the objects themselves within the interface.

We propose an *Object-focused environment*, composed of a 2D surface hosting behaviorally complete graphical representations of the objects. We provide prototype implementations named *Gaucho* and *Ronda*, which illustrate the application of our approach to a broad range of tasks, such as modeling, programming, program comprehension, and collaborative software engineering.

To validate our thesis, we conducted a summative evaluation, instrumented as a controlled experiment where we compared Gaucho with a traditional IDE, finding that it is indeed a viable alternative to the current state of the art.

Acknowledgements

It was four years ago when I crossed the ocean and landed in Switzerland, and since then i've met many people who have greatly helped me along the way into completing my Ph.D studies. I extend my gratitude towards all of them, starting from my advisor Michele Lanza, to former and current REVEALER's: the Doc Romain, Ricky, Mr D, Mircea, Lille, Alberto, Roberto, Remo, Luca, and Tommaso. It was great sharing an office, playing calcio, climbing in Vezia, traveling to conferences, and collecting anecdotes from Il Fornaio.

Michele, thanks for enlightening me with the ways of the word, and for all the support, patience and coolness you emanated during my time in Lugano.

I would also thank the members of my committee, for asking both interesting and tough questions, which greatly improved the contents of this dissertation.

To my former employers, Leandro and Valeria, thank you for teaching me so much in so little time; and to my former professors, Hernan and Carlos, who greatly inspired me to pursue a dissertation on such a interesting topic.

Needless to say, I greatly appreciate the effort and sacrifices that all my family endured during my extended european stay, GRACIAS TOTALES to all of you in the end of the world!

Wioleta, you made everything better, with your love and support. Thank you for an amazing time in Lugano, Kocham Cię.

Fernando Olivero
April 2013

Contents

Figures

Tables

Chapter 1

Introduction

Computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty.

—DONALD E. KNUTH

In his 1974 ACM Turing Award Lecture, Knuth defended his decision to include the word **art** in his renowned series of books entitled: "The art of computer programming" [1]. In the lecture, Knuth related programming to art, because it requires skill and the application of knowledge to master complexity, to ultimately produce beautiful programs [2].

In any form of art, humans playing the role of artists make use of several tools to produce their pieces. For example to produce the Venus of Milo, an ancient greek sculpture, the artist used a hammer and a chisel to carve out the feminine figure from an shapeless piece of marble rock. In this dissertation, we focus on the **tools** that computer programmers use to create **programs**; digital-age artisans crafting their pieces of art.

Tools are the means to an end, which is the crafting of computer programs. Computer programmers create and manipulate the elements that make up the software system, by interacting with several tools. Therefore, the choice computer programmers make on which tools to use is important, because tools have a great influence on how they approach problems and describe solutions. Dijkstra took this argument further by stating that tools enable computer programmers to think and express solutions to problems [3].

1.1 Symbolic Programming

We established that programmers use tools to craft and manipulate programs, without defining what constitutes a **program**. Since the dawn of general purpose mechanical and electronic machines, humans have devised mechanisms to instruct them to perform tasks. We define this series of instructions, in a broad sense, as a program. The purpose of a program is to aid human labour or augment human intellect by instructing a general purpose machine –the computer– to perform a series of operations.

Programs are expressed following an interface mechanism between humans and computers, namely the **language**. Programs represent solutions to problems within the design space delimited by a language. Ingalls stated that the purpose of a language is to provide a framework for communication, that serves as an interface between the models in the human mind and the computer [4].

The framework of communication initially evolved from basic machine instructions to assembly language, easing the writing of programs by abstracting the instructions and memory locations into a human readable form consisting of a combination of symbols and letters. A program written in assembly language consists of a series of operation codes mapping one to one to machine instructions, together with labels referencing memory locations, instead of a plain succession of ones and zeroes closer to the computer hardware.

The *prose* of humans and computers started to diverge with the appearance of **high-level** languages, such as Fortran, Lisp, Simula and Smalltalk. Programs evolved from simple tasks executing machine operations, to more complex tasks involving concepts such as control flow statements, arrays, files, functions, and objects. The latter are not directly available from the machine instruction set, but rather enabled via multiple layers of abstraction.

Programming languages gradually came closer to a human readable form. The framework of communication was based on a symbolic representation of the program, where programmers make use of symbols and rules, to compose and specify programs, and its relation with the other parts of the system.

In our work, we focus on the programming languages that represent programs in a textual form, that is programs that are defined by a piece of text –the **source code**– that specifies the constituents and the behavior of the program, whose content is governed by the syntactic and semantic rules of the language.

1.2 On the writing of programs

From early psychological theories of programming to the present, programming is acknowledged as a form of **writing** [5], thus while fulfilling programming tasks, the editing and reading of a textual representation of the program –the source code– is paramount. The former applies to programming languages that are based on symbolic programming, which are our focus.

Since the dawn of computing, programmers used pencil and paper to write down the programs in a particular language, translating the solutions to problems in a form that computers can understand. At first, the instrumentation of the **human-computer interaction** was based on encoding the symbolic program onto punch cards (see Figure 1.1). The cards represented one statement of the program encoded with the presence or absence of holes, then fed into a card reader machine to ultimately become understandable by the computer.

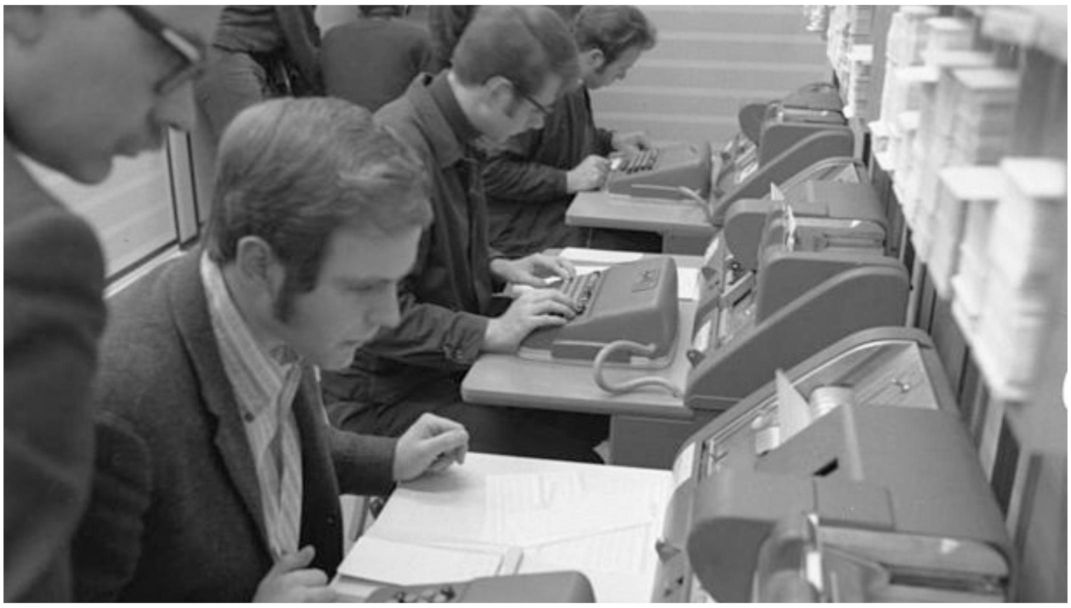


Figure 1.1. University students translating their programs onto punch cards

The removal of the intermediate step of encoding the programs on punched cards, came with the advent of one line editors such as teleprinter machines, and their graphical counterpart included in timesharing systems with terminal displays. Human-computer interaction became more interactive, given that programmers could now directly modify the programs one line at a time, instead of using the batch processing mode of yore [2].

Once the terminal displays and personal computers with graphical user interfaces became widespread, the appearance of software applications known as **word processors** raised the level of interactivity and flexibility of the programming tools. Word processors enabled the viewing and editing of multiple lines of the program at once, as opposed to the single line editing of command line editors. Word processors became an important programming tool, central to the instrumentation of human-computer interaction.

Nowadays, programmers use customized **source code editors**, evolved from general purpose text editors, for manipulating programs written in a particular language. Modern source code editors provide advanced features to aid programmers when writing and reading programs, such as automatically checking the syntactic rules to augment the view with relevant information, and the autocompletion of typed prefixes according to the context of the selected text, *i.e.* classes within a package or methods within a class hierarchy.

1.3 Programming is more than just writing

We established that programming involves the writing and reading of a textual representation of the program, thus programmers make use of some form of word processor to craft programs.

Nevertheless, the whole programming process involves a broad range of other activities, such as the transformation of source code into an executable artifact, the monitoring of running programs to detect and fix problems, and the creation of assertions on the quality and correctness of the programs.

1.3.1 Integrated Development Environments

During the late 70's and the mid 80's, programming became widespread due to the appearance of more interactive programming tools –word processors in terminal displays and personal computers–, and of simpler programming languages such as BASIC. Programmers made use of several different tools to accomplish diverse programming tasks, for example they use debuggers, compilers, linkers, and source-code editors. All the tools were running within the same operating system, yet each one remaining a standalone application.

The lack of a common interface and of interoperability between the tools, favored the appearance of the **integrated development environment** (IDE), a single application that included all the necessary tools.

The use of an IDE eased the undertaking of programming tasks, by providing a uniform interface to all the tools, which became accessible and were managed within one single entry point: the IDE.

One of the first IDEs to be widely adopted was Turbo Pascal, an IDE for the Pascal language that ran on the DOS-based personal computers of the 80's. Turbo Pascal enabled the running, compiling, debugging and editing of programs within a keyboard-based environment.

Nevertheless, the concept of the IDE was conceived a decade before the irruption of the personal computer era in the 80's, in the group working at the XEROX Palo Alto Research lab that created **Smalltalk**, the first dynamic object oriented programming (OOP) language. Figure 1.2 depicts a Smalltalk environment running on one of the first personal computers designed at XEROX Parc during the 70's.

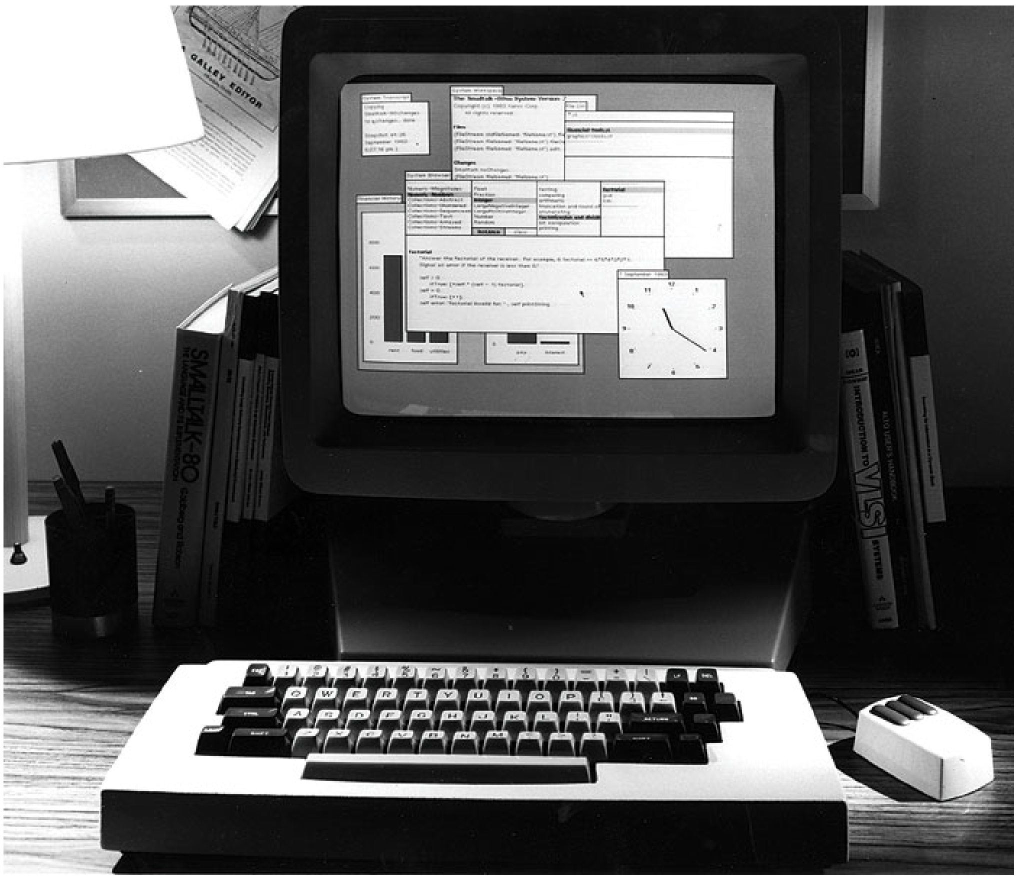


Figure 1.2. An IDE with a graphical user interface for Smalltalk

Nowadays, programmers craft programs with the use of IDEs, and interact with the source code and executables through graphical user interfaces inspired from the early Smalltalk systems, which consisted of a dynamic OOP language and an interactive development environment. Smalltalk was the first language to enable OOP via a graphical user interface that grants access to the underlying objects of the system, solely through graphical elements on the computer display, and the use of pointing devices and keyboards [6].

In this dissertation we focus on IDEs for programing in dynamic OOP languages, and use Smalltalk as the target language for our thesis.

1.4 Object-Oriented Programming

When asked what that means he replies, "Smalltalk programs are just objects." When asked what objects are made of he replies, "objects." When asked again he says "look, it's all objects all the way down. Until you reach turtles."

—Alan Kay

From the first high-level programming languages such as Fortran and Algol, programmers designed the programs around abstract data types and procedures, making a clear distinction between (inert) *data* and the functionality to modify it. Around the late 60's, a different programming language paradigm emerged as a result of the work of Alan Kay, based on the metaphor of communicating objects. Kay designed a system built from the ground up entirely on this metaphor, whose first implementation was the Smalltalk environment developed at Xerox Parc during the 70's [7].

Kay advocated the "disappearing of data" using only methods and objects. Objects subsume the concept of data and procedures into an entity which has state and behavior (methods), and the computation model is entirely based on message passing: the system is a collection of well behaved objects that ask each other to perform computation.

The key concepts of this new approach to computing came from biological cell communications modeled as networked whole computers. Kay was inspired by his biology and mathematics background, his work on the ARPAnet, which became the Internet, and the inheritance mechanism present in the Simula language [7]. Another source of inspiration was the Sketchpad, a computer aided design tool developed by Sutherland as part of his doctoral dissertation [8], which also introduced the concept of classes and instances.

The OOP paradigm proved to simple, flexible, and powerful. Object oriented programming radically changed the computing programming culture. In the present, the OOP paradigm is widely recognized and adopted by academia and industry, in its various flavors: dynamic (Smalltalk,Python), type-based (Java,C++), and prototype-based (Self,Javascript).

1.5 The Problem

We concluded that *programming is a form of writing*, and that the need to edit a textual representation of the program is satisfied by modern source code editors, which go beyond general purpose text edition facilities, by providing more specific features tailored to a programming language.

Nevertheless, programmers spend most of their time analyzing the program [9], performing maintenance tasks that require an understanding of the system. In doing so, programmers seek the aid of several tools because software systems are hard to understand [10]. The tools provide a means to wander around the system, learn about the composition of objects, and explore the relationships between them.

Nowadays, programmers use IDEs that provide numerous tools to effectively construct and maintain object-oriented programs. Besides source code editors, modern IDEs include compilers, debuggers, modeling, and testing tools, amongst many other specific tools. The main tool for programming –the IDE– emphasizes the manipulation of tools for writing and reading the programs [11]. The user interfaces of modern IDEs are built around a *tools metaphor* [12], because developers visualize and interact with tools for editing and manipulating the objects of the system. We expand this discussion in Section 2.4.

The tool-based environments produce both conceptual and technical problems when programming in OOP languages, discussed next.

1.5.1 The Conceptual Problem: Tools vs Objects

In object-oriented languages, the most important concept is that of an **object**. Moreover, in dynamic OOP languages –which is our focus–, the system is composed solely of objects, suppressing the arbitrary distinction between run-time and compile-time, to enhance the liveliness of the system.

The conceptual problem occurs because a tool-based environment treats objects as lifeless entities, that can be only manipulated through a set of tools. For example, to add or remove methods from a class, Smalltalk programmers use System Browsers that provide the means to do so. Nevertheless, method addition or removal might also be achieved by sending messages to the class, because in Smalltalk, classes are just another kind of live object in the system.

This dichotomy, between inert and live *human-object* interactions, results from the former promoting a message-based interface, whereas the latter empowers a tool with the responsibility of modifying inert *data*. In non-dynamic OOP languages, a tool-based IDE is instrumental because in such languages at compile time programmers deal solely with an inert specification of the program: the source code. Therefore, tools are the only possible means of manipulating data. On the other hand, in dynamic OOP languages a tool-based IDE is conceptually detrimental because it diverges from the philosophy behind the language: a computational model based on a world of live objects [4].

Nonetheless, programmers have become efficient on creating and maintaing programs using tool-based IDEs [13], thus we stress that the former is a conceptual problem, which hinders the understanding of the importance of the role of objects in dynamic OOP languages.

1.5.2 Technical Problems

In this section we discuss problems of the tool-based IDEs which are more practical in nature, and result from contrasting developers needs to the interfaces of the tool-based development environments. We categorized the problems into three categories: the first relates to a more *fine-grained manipulation*, the second to the *real estate management* of finite computer displays, and the third to the *representation* of the objects in the interface of the IDE.

Fine-grained Manipulation of Objects

The file-based view of mainstream OOP IDEs prevents the manipulation at a finer-grained level than source code files. In a study on programmers habits while using Eclipse [1], while performing maintenance tasks, Ko *et al.* elicited the requirement that IDEs should provide a fine-grained manipulation of the program entities [14]. Ko advocates for finer-grained representations of the system, because programmers tend to form working sets of classes, methods and packages, pertaining to the task at hand.

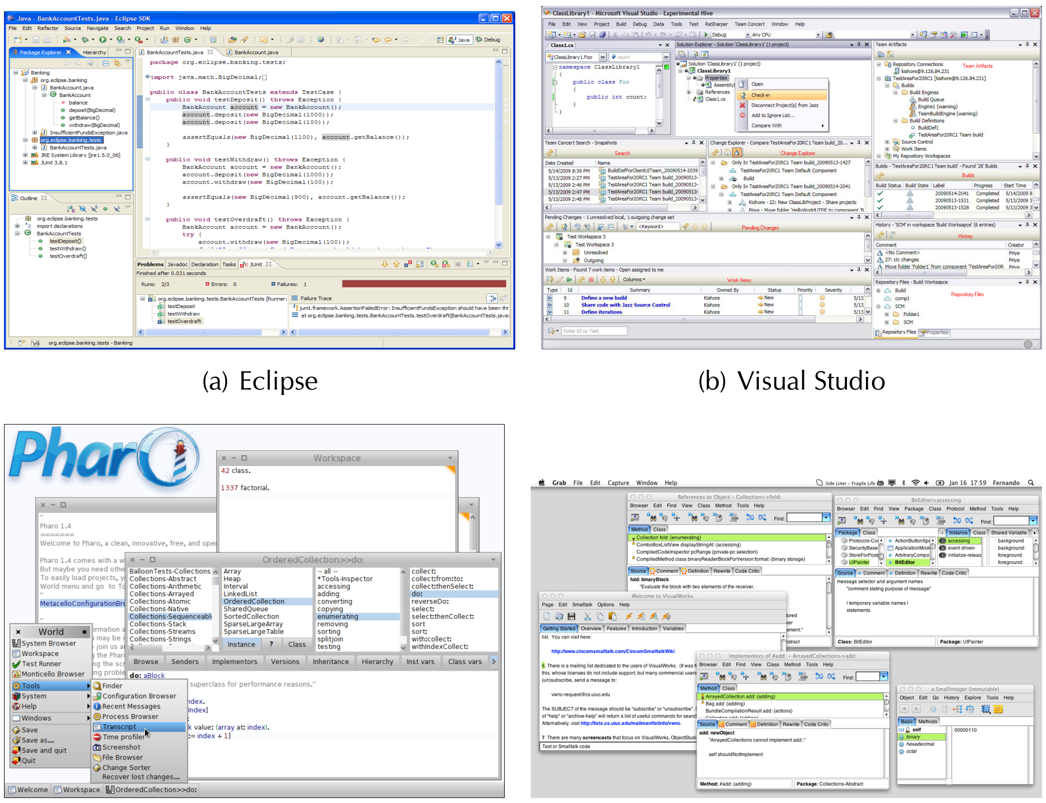
IDEs fail to meet this need, because they provide tools that enable a coarse manipulation of the objects, in the form of whole file navigators and editors as in IDEs for mainstream OOP languages such as JAVA.

[1]An IDE for the Java language, `http://www.eclipse.org`

Similarly, Smalltalk environments, include *system browsers* that grant access to packages, classes, and methods within the same tool. Kersten *et al.* stated that this problem is aggravated when the structure of the system is not aligned with the structure of the task, and propose a degree of interest model that defines a task context, to aid programmers into forming those working sets [15].

Real Estate Management

The user interface of modern IDEs are populated by diverse tools that enable a whole range of programming tasks. In this section we analyze the different mechanisms to arrange the tools within the interface, to detect shortcomings, and find opportunities for improvement. Figure 1.3 depicts several tool-based IDEs, that differ in how they handle the available real estate.



(a) Eclipse (b) Visual Studio

(c) Pharo (d) Visual Works

Figure 1.3. Tool-based IDEs with different real-state management

One possible form of arranging several elements is through a **tab-based interface**. Figure 1.3(a) depicts an Eclipse IDE with the central pane containing two tabs, each one viewing a single file, which in non-dynamic OOP languages represents and stores the classes of the program. Tabs are displayed within a container pane or window, that in Eclipse assumes the main role for programming. The use of tabs forces a single focus of attention: the contents of a single tab can be manipulated at a time [16].

A single focus of attention conflicts with the programmer's needs to create side by side views of the constituents of the program [17]. Sillito *et al.* compiled a series of questions asked during programming tasks [10] that highlights comparing programs elements to gain an understanding of the system.

On the other hand, an interface based on **tiles** allows the arrangement of side by side views of the objects. Tiles are contiguous rectangular panes which are confined within a parent container. Mainstream IDEs allow a tile-based interface, besides a tab-based one. One example is the Visual Studio IDE for the C# language [2], depicted in Figure 1.3(b).

Nevertheless, the use of tiles imposes constraints, because tiles are arranged contiguously in the parent container, according to a fixed layout such as a grid or flow layout [18]. The constrained layout hinders flexibility when creating custom views of the program, while forming the working sets.

Smalltalk environments provide more freedom for laying out the objects of the system. The interface consists of a 2D area spawning the whole display, that includes multiple windows, which in turn each host one tool. The windows may overlap, to make better use of the scarce real estate, confined within the finite dimensions of the computer display.

A **multiple overlapping windows** scheme, can either be implemented within a single operating system (OS) window, as in Pharo an open-source Smalltalk environment [3], or match every smalltalk window to its own OS counterpart, as in Visual Works, a commercial Smalltalk [4], see Figure 1.3(c) and Figure 1.3(d). The overlapping windows share the scarce real estate, enabling programmers to potentially handle more graphical elements than the tab or tile based interfaces. Thus, the workspace often becomes crowded with many opened windows, due to the number of objects and relationships a developer needs to examine.

The *window plague* can be mitigated by an automatic mechanism that finds and removes unused (or unwanted) windows [19]. Nevertheless, problems remain, because of the generic purpose of the tools: the manipulated objects

[2]`http://www.microsoft.com/visualstudio/`

[3]`http://www.pharo.org`

[4]`http://www.cincomsmalltalk.com/main/products/visualworks/`

depend on the current selection of the tools. The selection can be modified, thus the mapping of windows to the objects is not persistent, which complicates the reasoning behind the automatic scheme.

The problem of the explosion of graphical elements within the interface, occurs also when using tabs or tiles. Moreover, all the mentioned schemes can be used in conjunction: tabs and tiles may coexist within the interface, and smalltalk windows can include tabs.

To summarize, tool-based interfaces work on a coarse-grained level, which conflicts with programmers need to form working sets of *objects*, which may spawn across all the modules of the system. Another missing opportunity occurs because the mentioned schemes confine the graphical elements to a finite 2D area, which prevents programmers from using a spatial memory and reasoning for the whole system [20], given that the clutter in the interface is proportional to the number of objects relevant to the task at hand.

Representation of the Objects in the Interface

Software is intangible [21; 22], thus the visual appearance of objects is an extrinsic characteristic. Nevertheless, the objects must be represented in a graphical form throughout the IDEs, because programmers interact with the system using computer displays. Despite the intangible nature of software, objects have a structure and composition that serves as a guideline for designing their visual appearance. For instance, the containment relationship between a class and its methods, or between a package and its classes, serves as a guideline for the representation of the program, either in a textual or a graphical form.

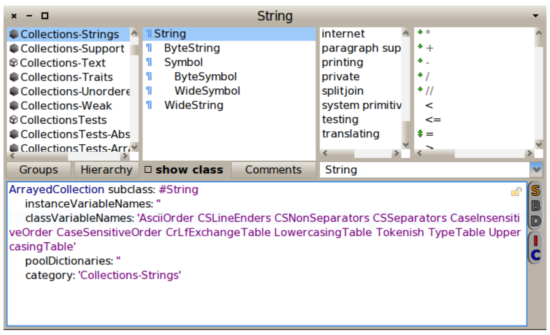
The visual appearance of objects is an important concern when performing programming and program comprehension tasks. For example, the use of software visualization enable programmers to gain an understanding of the system by exploiting metrics or intrinsic properties of the target of study: packages, classes, methods, or the system itself [23]. In this dissertation we focus on tools for programming in OOP languages, thus our concern is the visual representation of objects within the interface.

The IDEs of mainstream non-dynamic OOP languages are file-based, making it explicit that source code is stored in operating system files. File-based IDEs represent the program in a textual form, namely the source code, split into files containing one or more elements of the program. Figure 1.3(a) depicts Eclipse, a mainstream IDE with a standard arrangement of panes in the interface, which includes several sub-panes surrounding a main tabbed pane that presents the source code, following a *bento-box* model [24].

The source code is the complete specification of the program, it defines all the objects, their behavior and composition. When the programmer writes or reads the code, he has to deal with the complexity of the syntactic rules of the language, and process information to effectively modify the program. Therefore, IDEs make use of graphical elements which abstract over the raw code, such as code elision or class overview widgets. Code elision is the folding of meaningful portions of code on demand, to hide irrelevant sections of the objects, to aid programmers when reading complex source code files [25]. Similarly, IDEs provide another means of viewing and navigating the composition of classes and packages, in the form of *overview* widgets, which enable a tree view interaction with the objects, depicted in Figure 1.3(a).

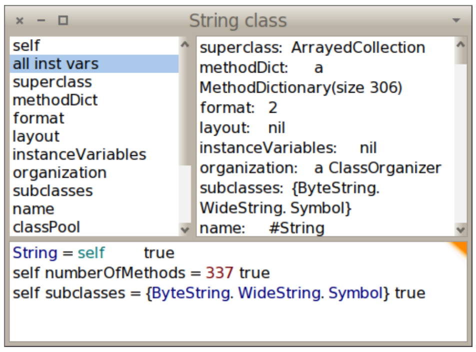
IDEs that provide the latter favor *chunking*. The method of presenting information known as chunking consists in splitting concepts into smaller pieces, which are easier to manage and understand [26]. In this manner, file-based IDEs enhance the usability of the interface at the expense of providing more than one representation of the objects.

On the other hand, Smalltalk environments use a single representation of the objects: a tool-based representation. In Smalltalk the source code is presented in small chunks within the panes of the tools named *browsers*, that allow navigating the system, and editing packages, classes and methods. The tool-based interface of Smalltalk inherently favors chunking. Figure 1.4 depicts the two main tools included in Smalltalk systems: the browser for manipulating the objects that make up the program –classes, methods and packages—, and the inspector for manipulating instances.



```
ArrayedCollection subclass: #String
    instanceVariableNames: ''
    classVariableNames: 'AsciiOrder CSLineEnders CSNonSeparators CSSeparators CaseInsensiti
veOrder CaseSensitiveOrder CrLfExchangeTable LowercasingTable Tokenish TypeTable Upper
casingTable'
    poolDictionaries: ''
    category: 'Collections-Strings'
```

(a) System Browser



```
superclass:  ArrayedCollection
methodDict:     a
MethodDictionary(size 306)
format:  2
layout:   nil
instanceVariables:    nil
organization:   a ClassOrganizer
subclasses:  {ByteString.
WideString. Symbol}
name:    #String
```

```
String = self        true
self numberOfMethods = 337 true
self subclasses = {ByteString. WideString. Symbol} true
```

(b) Inspector

Figure 1.4. The Tools of a Smalltalk Environment

The downside of a tool-based interface is a single object may be presented by more than one tool, thus the identity property of OOP languages is not conveyed in the interface [11]. Another related problem is that tool-based interfaces present the many different objects of the system using the same set of widgets. Classes, methods, and packages appear as a list item in the browsers columns, or in a textual description when the selection of the browser points to a specific object. Figure 1.4 depicts a system browser with a selection on the class *String*, and the code pane presenting a textual representation of the class description. There are missing opportunities in this scheme, because the semantics of the objects may be only conveyed by icons in the list item denoting the name of the objects, as opposed to designing a visual appearance for each kind of object, tailored to reflect its type, structure, behavior, and state.

1.5.3 Summary

We started by discussing the conceptual problem of using a tool-based interface for programming in dynamic OOP language.The problem is that objects remain hidden behind the tools, which a have prominent role in the interface. Thus, objects are perceived as inert data that can only be manipulated by the use of a tool. This conflicts with the essence of OOP, which is based on a computational model of a world of live objects.

Secondly, we discussed the real estate management of current IDEs, for both dynamic and non-dynamic languages, to find out that current interfaces present the objects in tab- or tile-based interfaces, or a multiple overlapping windows scheme, confined within the finite boundaries of the computer display. We detailed how the traditional interfaces hinder the use of spatial memory and reasoning, effectively used for software visualization approaches, such as class blueprints [27] and software cartography [28].

Lastly, we analyzed how IDEs represent the objects that make up the program in the interface, to find missing opportunities of the tool-based interfaces for conveying the semantics of the many different kinds of objects. IDEs present the objects either in a textual form –the source code–, or within one or more generic tools, making it difficult to denote changes to the objects due to a loss of identity, and presenting a uniform appearance regardless of the kind of object being displayed. For instance, a Class is different than a Method, and a Number is different than a Color, nevertheless these objects have the same visual appearance within tool-based interfaces.

1.6 Thesis

Computer programmers craft programs with the aid of tools, which evolved from simple pen and paper to those using a graphical user interface based on windows, icons, menus and a pointing device. Nowadays, programs are created, maintained and executed with Integrated Development Environments (IDEs), which are built on a tool-based interface.

In this dissertation we focus on programs which are described entirely by objects, following a computational model of a world of objects that collaborate by sending each other messages, *i.e.*, we focus on the development environments of Object-Oriented Programming languages.

The dynamic nature of a world of living objects is lessened in the interface of modern IDEs, which revolves around a textual representation of the program, namely the source code. In this sense, programming with traditional IDEs resembles the composing of musical tunes: both activities specify a dynamic artifact with a static one, method statements and class definitions in the former, and pentagrams with notes in the latter.

The tool-based interface of IDEs prevents programmers to directly engage into conversations with the underlying objects of the system. On the other hand, the interfaces built on the desktop metaphor make explicit the presence of the manipulable entities, and provide the means to interact with them.

We investigate the application of the desktop metaphor to an object oriented development environment with an interface solely based on direct manipulation of objects, which we name an **Object-focused environment**.

We formulate our thesis as:

> *An Object-focused environment supports an extensible set of software engineering tasks, eliminates the need for extrinsic tools that work on the source-code, and conveys the dynamic nature of a world of living objects.*

To validate our thesis, we designed and implemented an object-focused environment, and applied the novel interface to several software engineering tasks, ranging from modeling to program comprehension tasks.

1.7 Contributions

In the light of our described thesis, the contributions of this dissertation are:

The definition of an object-focused interface for software development [29]. We describe in Chapter 3 the building blocks of our design of a development environment based on a desktop metaphor, a plausible alternative to the tool-based interfaces of traditional IDEs.

The application of the object-focused interface to modeling and OOP [30]. In Chapter 3 we also describe the shapes of the environment, which are the behaviorally complete graphical counterparts of the objects that form the models (see Section 3.3), and the programs (see Section 3.4).

The implementation of an object-focused environment [31]. We implemented **Gaucho**, an object-focused programming environment for the crafting of models composed of objects, described in class-based object oriented languages. Gaucho is publicly available and has been used in academic research. We present our tool in Section 3.2.

The empirical validation of our approach through a controlled experiment [32]. We performed a controlled experiment to compare traditional IDEs and Gaucho with respect to a set of program comprehension tasks extracted from the literature, such as creating, navigating, refactoring, and understanding an object-oriented system. We present the experiment, and reflect on the findings in Chapter 4.

The design and implementation of a tool to support human-centric controlled experiments [33]. We present **Biscuit** in Section 4.1. Biscuit is an automated experiment runner toolset to support conducting experiments for evaluating software tools with human subjects.

The application of the object-focused interface to collaboration [34]. To demonstrate the extensibility of our approach, we augmented the single developer environment Gaucho into a collaborative development environment. In Chapter 5 we investigate the use of an object-focused environment to support collaboration and real-time awareness of fine-grained changes to the system.

The implementation of a tool to support collaboration [34] In Section 5.2 we describe **Ronda**, an object-focused environment that provides first-class support for collaborative development sessions.

1.8 Structure of the Document

We structured the remainder of this dissertation as follows:

Chapter 2. We start with a historical perspective on our work, by describing the evolution of the tools used by programmers to craft programs. We discuss the tool-based nature of IDEs, describe their problems, and present several alternative user interfaces that alleviate them.

Chapter 3. We continue by describing our approach: the **Object-focused environment**. An extensible re-design of object oriented development environments, with applications to different areas of software engineering.

We present **Gaucho**, the object-focused environment that enables modeling, programming and program comprehension tasks, by interacting with shapes representing the objects.

Chapter 4. Next, we present a controlled experiment we performed to validate our approach. We also describe **Biscuit**, an automated experiment runner toolset we implemented to support researchers when performing controlled experiments with human subjects to evaluate software tools.

Chapter 5. After describing the evaluation and its instrumentation, we present **Ronda**, an enhancement to the single developer environment Gaucho which adds support for the collaborative nature of programming.

We implemented Ronda to demonstrate the extensibility our approach, and investigate the application of object-focused interfaces to collaborative development environments.

Chapter 6. We conclude this dissertation by discussing our approach and summarizing the contributions of this work, and detailing future research directions.

Chapter 2

State of the Art

It has to do with the influence of the tool we are trying to use upon our own thinking habits. I observe a cultural tradition, which in all probability has its roots in the Renaissance, to ignore this influence, to regard the human mind as the supreme and autonomous master of its artefacts.

—EDSGER W. DIJKSTRA

From the beginning of the computer age, computer scientists have underestimated the influence that tools have upon our working habits and the quality of the artifacts we produce [3]. For example, Weinberg recalls the skepticism surrounding the development of Fortran amongst fellow programmers, who challenged the adoption of high level languages on the grounds that programmers would always produce more efficient code [5]. On the other hand, several researchers pondered on the impact of tools. Dijkstra advocated for better tools to struggle against their inadequacies, enabling programmers to focus on solving the problems [3]. Knuth envisioned the use of tools that are a pleasure to use, encouraging programmers to write better programs, by providing more interactivity than the batch processing of early punch card based systems [2].

In this chapter we describe the tool-based development environments for programming in object oriented languages. We start by describing the interfaces that give a prominent role to the tools, which are separate from the objects of the system, whose generic nature enables the manipulation of many elements of the program. We gradually move towards environments that favor fine-grained manipulation of software artifacts –objects, code–, where the tools permeate throughout all the environment, to solve the manipulation, layout and visualization problems discussed in Section 1.5.

Structure of the chapter. We present Smalltalk, from the original versions that introduced the IDE, to more modern implementations in Section 2.1. In Section 2.2 we describe several environments with alternative interfaces for programming in code-centric languages. We detail modeling environments for languages that abstract away from the source code to models, in Section 2.3. We conclude this chapter in Section 2.4 by describing the environments which are closer to our approach, the ones which favor direct manipulation of objects.

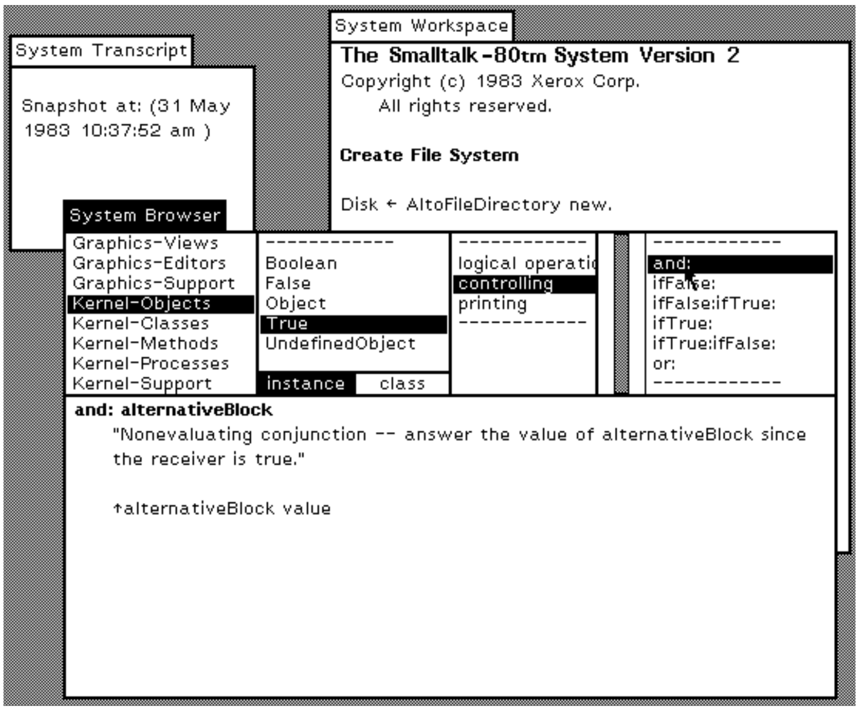
2.1 Smalltalk

The Smalltalk language was the precursor of both object oriented programming, and the integrated development environment. The original environments developed at Xerox PARC during the 70's introduced many of the tools that are still in use today, such as object inspectors, debuggers, and code editors and navigators. Differently than other programming languages that make a clear distinction between the source code and the running programs, in Smalltalk the environment and the language are strongly coupled. This coupling results from the computational model of Smalltalk: a world of *live* objects, that collaborate by sending messages to each other. The graphical user interface to interact with the objects was developed together with the language [7].

2.1.1 Browsers and Inspectors

From the beginning, Smalltalk systems were built around a tools metaphor [12], providing numerous tools for editing and manipulating the objects of the system. The basic tools included in a Smalltalk environment are: *Inspectors* for manipulating instances –objects–, *Browsers* to manipulate the classes, methods, and packages of the system, *Workspaces* for sending messages by evaluating expressions, and the *Transcript* as a global console for the system [35]. Modern Smalltalk systems evolved to include *Test Runner* tools for running tests and refactoring browsers [36].

Figure 2.1 depicts two Smalltalk systems, which resemble one another, despite having been implemented in different decades, including the basic tools of the environment: a browser selected on the True class displaying the method named *and:*, an inspector on the Array object, a workspace with valuable expressions, a transcript and a test runner.

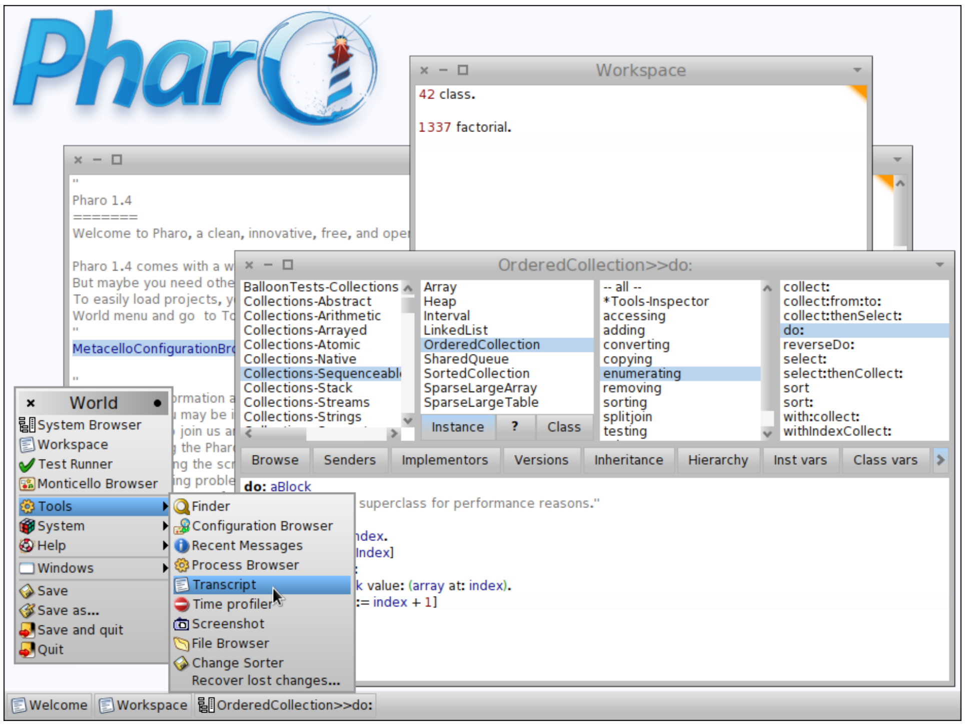


```
Snapshot at: (31 May
1983 10:37:52 am )
```

```
The Smalltalk-80tm System Version 2
Copyright (c) 1983 Xerox Corp.
    All rights reserved.

Create File System

Disk ← AltoFileDirectory new.
```

```
and: alternativeBlock
    "Nonevaluating conjunction -- answer the value of alternativeBlock since
    the receiver is true."

    ↑alternativeBlock value
```

(a) Smalltalk-80, 1980



```
42 class.

1337 factorial.
```

```
"
Pharo 1.4
=======
Welcome to Pharo, a clean, innovative, free, and ope

Pharo 1.4 comes with a w
But maybe you need othe
To easily load projects, y
World menu and go to To
"
MetacelloConfigurationBro

"
ormation
may be i
join us a
the Phar
ng the scr
ng proble
```

```
do: aBlock
superclass for performance reasons."
ndex.
Index]
k value: (array at: index).
:= index + 1]
```

(b) Pharo, 2012

Figure 2.1. Basic tools included in Smalltalk, from the original Smalltalk-80 to the modern Pharo IDE

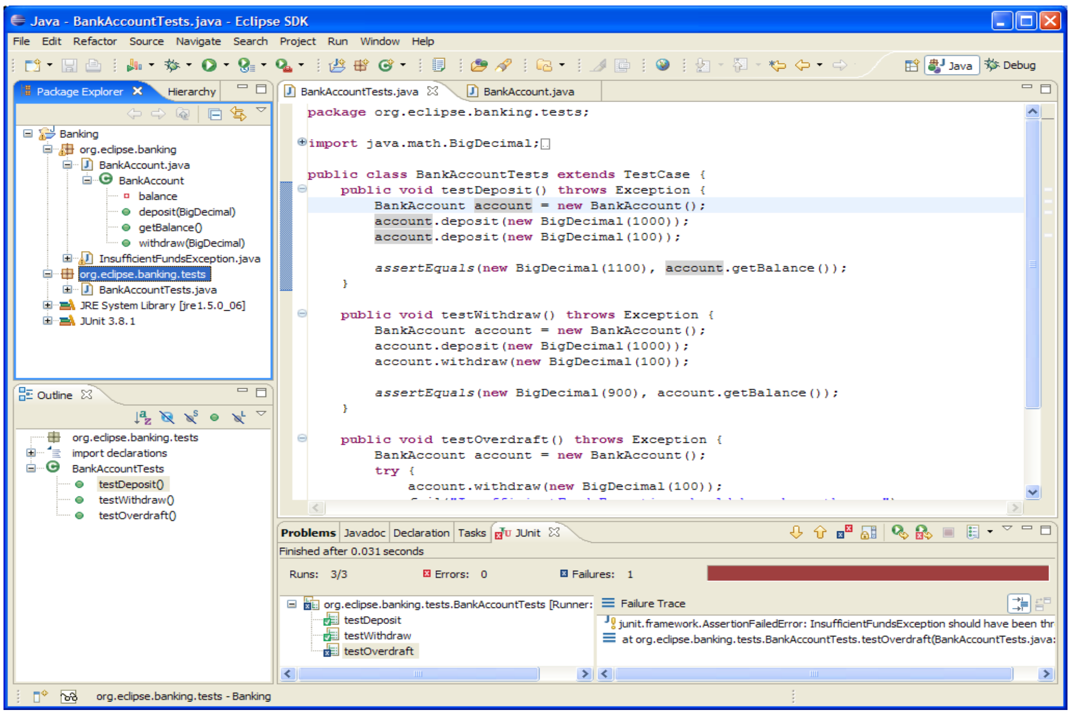
2.2 Code-Centric Environments

We stated previously that nowadays developers write programs, aided by integrated development environments (IDEs); and that IDEs feature numerous tools that provide the means to construct programs.

In this section we describe the mainstream IDEs, that are built around a textual representation of the program, namely the source code. We refer to these IDEs as *code-centric*, because performing textual editions of the source code is the main mechanism for programming.

Moreover, the files which host the code take a primary role within the interface, an often adopted fashion is to arrange the various tools in a *bento box* model, around the source code. Mainstream IDEs are file-based, because they make explicit that source code is stored in operating system files.

We depict Eclipse, a code-centric IDE for JAVA in Figure 2.2.



```
package org.eclipse.banking.tests;

import java.math.BigDecimal;

public class BankAccountTests extends TestCase {
    public void testDeposit() throws Exception {
        BankAccount account = new BankAccount();
        account.deposit(new BigDecimal(1000));
        account.deposit(new BigDecimal(100));

        assertEquals(new BigDecimal(1100), account.getBalance());
    }

    public void testWithdraw() throws Exception {
        BankAccount account = new BankAccount();
        account.deposit(new BigDecimal(1000));
        account.withdraw(new BigDecimal(100));

        assertEquals(new BigDecimal(900), account.getBalance());
    }

    public void testOverdraft() throws Exception {
        BankAccount account = new BankAccount();
        try {
            account.withdraw(new BigDecimal(100));
```

```
Finished after 0.031 seconds
Runs:  3/3          Errors:  0          Failures:  1

org.eclipse.banking.tests.BankAccountTests [Runner:
    testDeposit
    testWithdraw
    testOverdraft

Failure Trace
junit.framework.AssertionFailedError: InsufficientFundsException should have been thr
at org.eclipse.banking.tests.BankAccountTests.testOverdraft(BankAccountTests.java:
```

(a) Eclipse

Figure 2.2. Code-centric mainstream IDEs

The design and usage of Code-centric IDEs produces some problems we discussed in Section 1.5.2, enumerated next:

- The fine-grained **manipulation** of the software artifacts.
- The real estate management that enables or impedes to **layout** the artifacts in an meaningful manner.
- The level of refinement of the **visualization** of the programs elements within the interface.

Next we describe several alternative code-centric environments, designed for manipulating source-code in a non-traditional interface, to alleviate or avoid the enumerated problems at all. The mentioned approaches are either tools that complement the IDE or full-blown development environments.

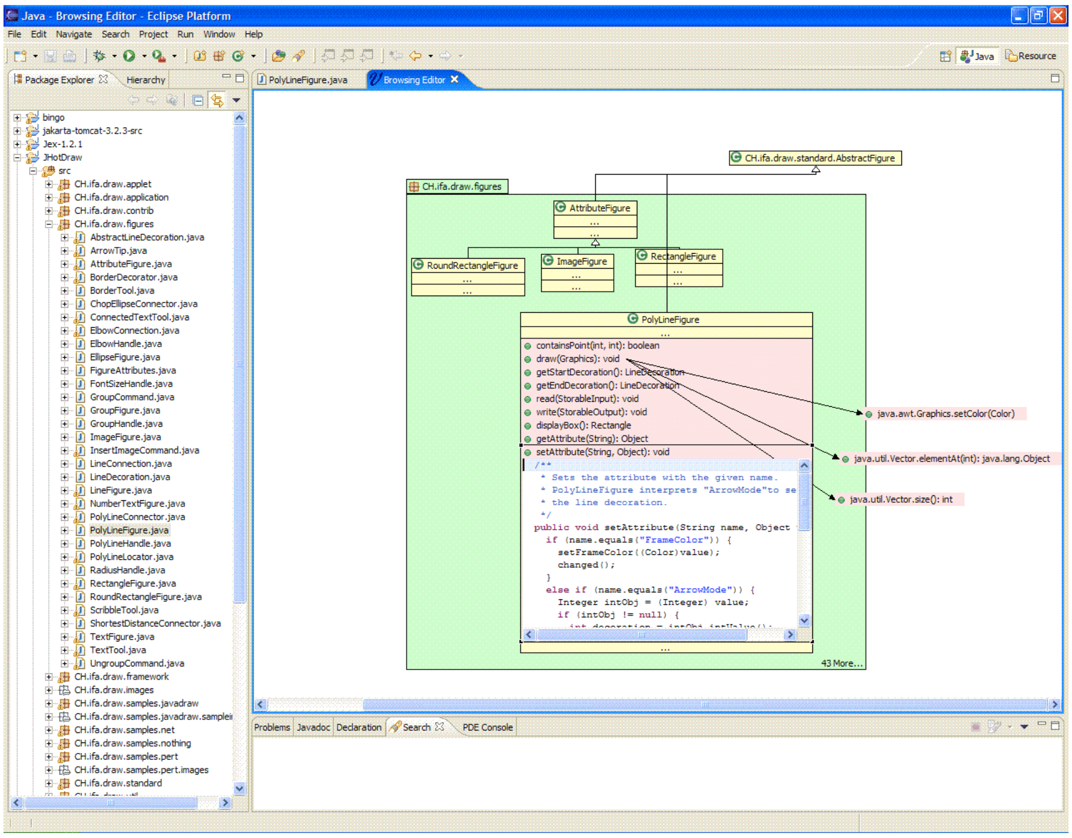
2.2.1 Relo

Relo is a program comprehension tool, that augments the Eclipse IDE with an interactive diagram that includes a graphical representation of the code.

Relo aids programmers to comprehend complex systems with large codebases, by making explicit the context while exploring the code. Figure 2.3 depicts the tool within the Eclipse IDE.

The tool is designed on the premise that when exploring large codebases programmers follow a bottom-up approach, starting from a relevant artifact, and adding the related ones by exploring their relationships. Therefore, RELO starts with a clean diagram, and provides the means to add elements from the system, and to easily navigate to other elements by interacting with the diagram. The tool provides a *fine-grained manipulation* of the classes, methods and packages of the program, thus addressing the first problem.

The nodes of the diagram are graphical elements representing the classes, methods and packages of the system. The source code is abstracted into a visual language resembling the unified modeling language (UML), to ease the comprehension of the objects. The nodes are automatically arranged following topological constraints, for example the inheritance edges between two classes are drawn vertically, while method calls are drawn horizontally. Even though RELO includes some form of real estate management, the tool lacks an overall system layout management, because the diagrams are focused into small portions of the system, providing a graphical context for exploring parts [37].


Java - Browsing Editor - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Package Explorer
Hierarchy
PolyLineFigure.java
Browsing Editor
CH.ifa.draw.standard.AbstractFigure
CH.ifa.draw.figures
AttributeFigure
RoundRectangleFigure
ImageFigure
RectangleFigure
PolyLineFigure
java.awt.Graphics.setColor(Color)
java.util.Vector.elementAt(int): java.lang.Object
java.util.Vector.size(): int
43 More...
Problems
Javadoc
Declaration
Search
PDE Console

```
containsPoint(int, int): boolean
draw(Graphics): void
getStartDecoration(): LineDecoration
getEndDecoration(): LineDecoration
read(StorableInput): void
write(StorableOutput): void
displayBox(): Rectangle
getAttribute(String): Object
setAttribute(String, Object): void
/**
 * Sets the attribute with the given name.
 * PolyLineFigure interprets "ArrowMode"to se
 * the line decoration.
 */
public void setAttribute(String name, Object
  if (name.equals("FrameColor")) {
    setFrameColor((Color)value);
    changed();
  }
  else if (name.equals("ArrowMode")) {
    Integer intObj = (Integer) value;
    if (intObj != null) {
```

Figure 2.3. Relo

2.2.2 Code Canvas

Code Canvas is a zoomable user interface for software development. The project documents are laid out in a 2D surface, to leverage spatial memory and keep developers oriented.

The canvas houses editable thumbnails of the code, which represent a shrunk down version of the source files, with a customizable level of detail through a semantic zooming mechanism (see Figure 2.4). Methods are contained within the classes, which in turn are contained within packages. The inner details of an element is only displayed when zoomed in. For example, the statements of a method are only visible when the containing class has a high level of detail, and is focused on that method.

Code Canvas uses a spatial representation of editable development documents to allow developers to use spatial memory to find them [24], to ease the preservation and communication of task contexts across multiple sessions.

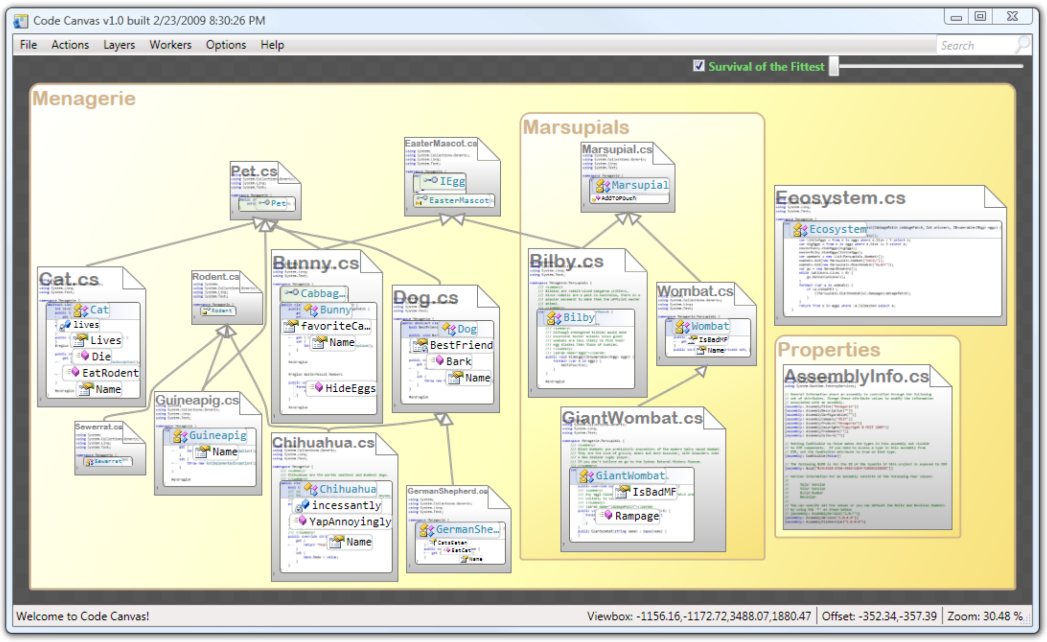


Figure 2.4. Code Canvas

2.2.3 Code Bubbles

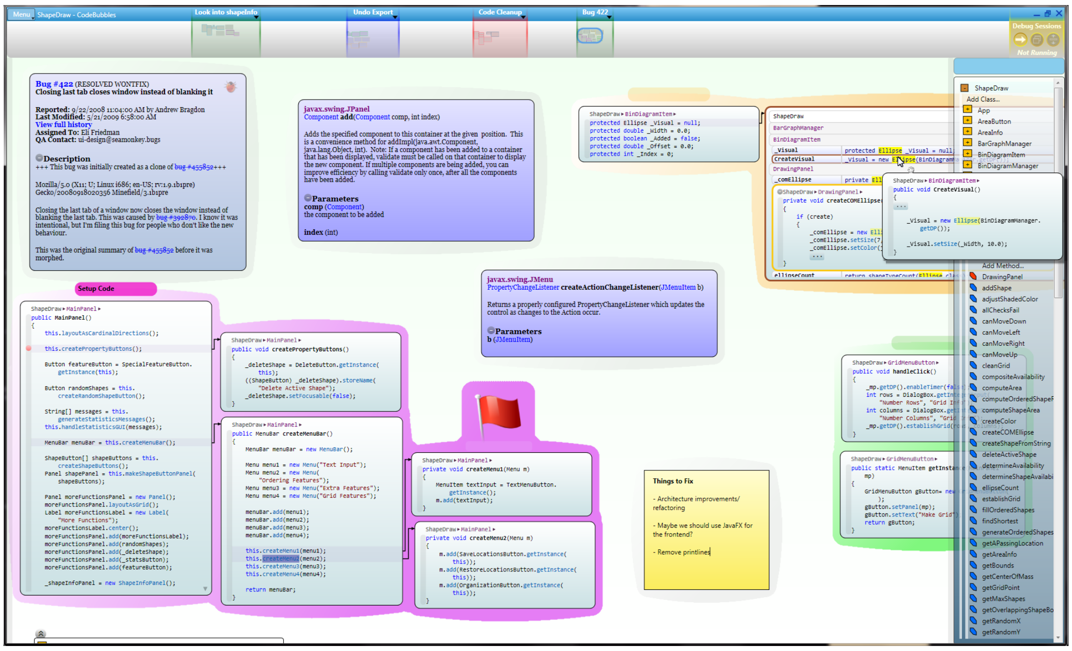
Code Bubbles is a programming environment that represents code as editable fragments called bubbles, forming concurrently visible working sets that avoid the continuous back and forth navigation typical of traditional IDEs [17]. The code bubbles exist and are placed on a non-overlapping 2-D surface [38]. The bubbles are interactive views of source code fragments such as a method or a collection of member variables (see Figure 2.5).

2.2.4 Summary

To ease the comprehension, and to favor chunking, RELO and Code Canvas represent the programs elements in a **visual** manner, either using a language similar to UML that minimizes the details of the code, or by adopting semantic zooming for controlling the level of detail.

The tools enable a **fine-grained manipulation** of those graphical elements, thus satisfying developer needs to create side by side views of individual program elements for comparison, and the forming of working sets that may span the system.

Code Canvas and Code Bubbles both address the real estate management on the large scale, the use of spatial memory and reasoning of a **layout** of the whole program, persisted across multiple development sessions. On the other hand, RELO limits the size of the diagrams to enable exploration of small portions of the system, minimizing the presented information.



Bug #422 (RESOLVED WONTFIX)
Closing last tab closes window instead of blanking it

Reported: 9/22/2008 11:04:00 AM by Andrew Bragdon
Last Modified: 5/21/2009 6:58:00 AM
View full history
Assigned To: Eli Friedman
QA Contact: ui-design@seamonkey.bugs

Description
+++ This bug was initially created as a clone of ~~bug #455852~~+++

Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.1b1pre) Gecko/2008091802356 Minefield/3.1b1pre

Closing the last tab of a window now closes the window instead of blanking the last tab. This was caused by ~~bug #392870~~. I know it was intentional, but I'm filing this bug for people who don't like the new behaviour.

This was the original summary of ~~bug #455852~~ before it was morphed.

javax.swing.JPanel
Component **add**(Component comp, int index)

Adds the specified component to this container at the given position. This is a convenience method for addImpl(java.awt.Component, java.lang.Object, int). Note: If a component has been added to a container that has been displayed, validate must be called on that container to display the new component. If multiple components are being added, you can improve efficiency by calling validate only once, after all the components have been added.

Parameters
comp (Component)
the component to be added

index (int)

javax.swing.JMenu
PropertyChangeListener **createActionChangeListener**(JMenuItem b)

Returns a properly configured PropertyChangeListener which updates the control as changes to the Action occur.

Parameters
b (JMenuItem)

```
ShapeDraw ▸ BinDiagramItem ▸
protected Ellipse _Visual = null;
protected double _Width = 0.0;
protected boolean _Added = false;
protected double _Offset = 0.0;
protected int _Index = 0;
```

```
ShapeDraw
BarGraphManager
BinDiagramItem
_Visual          protected Ellipse _Visual = null;
CreateVisual     _Visual = new Ellipse(BinDiagramManager.getDP());
DrawingPanel
_comEllipse      private Ell
ellipseCount     return shapeTypeCount(Ellipse.class);
```

```
ShapeDraw ▸ DrawingPanel ▸
private void createCOMEllipse(
{
    if (create)
    {
        _comEllipse = new Elli
        _comEllipse.setSize(7,
        _comEllipse.setColor(
        ...
}
```

```
ShapeDraw ▸ BinDiagramItem ▸
public void CreateVisual()
{
...

    _Visual = new Ellipse(BinDiagramManager.
        getDP());

    _Visual.setSize(_Width, 10.0);
}
```

```
ShapeDraw ▸ MainPanel ▸
public MainPanel()
{
    this.layoutAsCardinalDirections();

    this.createPropertyButtons();

    Button featureButton = SpecialFeatureButton.
        getInstance(this);

    Button randomShapes = this.
        createRandomShapeButton();

    String[] messages = this.
        generateStatisticsMessages();
    this.handleStatisticsGUI(messages);

    MenuBar menuBar = this.createMenuBar();

    ShapeButton[] shapeButtons = this.
        createShapeButtons();
    Panel shapePanel = this.makeShapeButtonPanel(
        shapeButtons);

    Panel moreFunctionsPanel = new Panel();
    moreFunctionsPanel.layoutAsGrid();
    Label moreFunctionsLabel = new Label(
        "More Functions");
    moreFunctionsLabel.center();
    moreFunctionsPanel.add(moreFunctionsLabel);
    moreFunctionsPanel.add(randomShapes);
    moreFunctionsPanel.add(_deleteShape);
    moreFunctionsPanel.add(_statsButton);
    moreFunctionsPanel.add(featureButton);

    _shapeInfoPanel = new ShapeInfoPanel();
```

```
ShapeDraw ▸ MainPanel ▸
public void createPropertyButtons()
{
    _deleteShape = DeleteButton.getInstance(
        this);
    ((ShapeButton) _deleteShape).storeName(
        "Delete Active Shape");
    _deleteShape.setFocusable(false);
}
```

```
ShapeDraw ▸ MainPanel ▸
public MenuBar createMenuBar()
{
    MenuBar menuBar = new MenuBar();

    Menu menu1 = new Menu("Text Input");
    Menu menu2 = new Menu(
        "Ordering Features");
    Menu menu3 = new Menu("Extra Features");
    Menu menu4 = new Menu("Grid Features");

    menuBar.add(menu1);
    menuBar.add(menu2);
    menuBar.add(menu3);
    menuBar.add(menu4);

    this.createMenu1(menu1);
    this.createMenu2(menu2);
    this.createMenu3(menu3);
    this.createMenu4(menu4);

    return menuBar;
}
```

```
ShapeDraw ▸ MainPanel ▸
private void createMenu1(Menu m)
{
    MenuItem textInput = TextMenuButton.
        getInstance();
    m.add(textInput);
}
```

```
ShapeDraw ▸ MainPanel ▸
private void createMenu2(Menu m)
{
    m.add(SaveLocationsButton.getInstance(
        this));
    m.add(RestoreLocationsButton.getInstance(
        this));
    m.add(OrganizationButton.getInstance(
        this));
}
```

Things to Fix

- Architecture improvements/ refactoring

- Maybe we should use JavaFX for the frontend?

- Remove printlines

```
ShapeDraw ▸ GridMenuButton ▸
public void handleClick()
{
    _mp.getDP().enableTimer(false);
    int rows = DialogBox.getIntegerInput(
        "Number Rows", "Grid Info");
    int columns = DialogBox.getIntegerInput(
        "Number Columns", "Grid Info");
    _mp.getDP().establishGrid(rows, columns);
}
```

```
ShapeDraw ▸ GridMenuButton ▸
public static MenuItem getInstance(MainPanel
    mp)
{
    GridMenuButton gButton= new GridMenuButton(
        );
    gButton.setPanel(mp);
    gButton.setText("Make Grid");
    return gButton;
}
```

Figure 2.5. Code Bubbles

2.3 Modeling Environments

In this section we analyze the tools that support both *formal* and *informal* modeling practices. The design of modeling tools relates to our discussion on the interfaces of development environments, because they address similar concerns such as the visual representation and placement of the elements that make up the models, which are generally mapped to classes, methods, and packages.

Modeling is the use of something in place of something else for some cognitive purpose. A model is an abstraction, in the sense that it cannot represent all aspects of the real world [39]. We discuss further the differences and similarities of modeling and programming practices in Chapter 3.

Informal Modeling

Modeling its simplest form is accomplished with the whiteboard, or other kind of free-form sketching mechanism, to draft the concepts and overall architecture of a software system. The use of digital tools designed to leverage the flexibility of such approaches, result in tool-based informal modeling practices, which enable persisting and sharing the created models, that are described using simple visual notations. One exemplar is CEL[1], a multitouch tablet application that provides the essential means to model software systems, based on a minimalistic set of elements that represent: concepts, generic relations, and containment [40]. CEL uses a matrix-based visual metaphor, and semantic zooming to address the problems of the small screen size of tablet computers (see Figure 2.6).

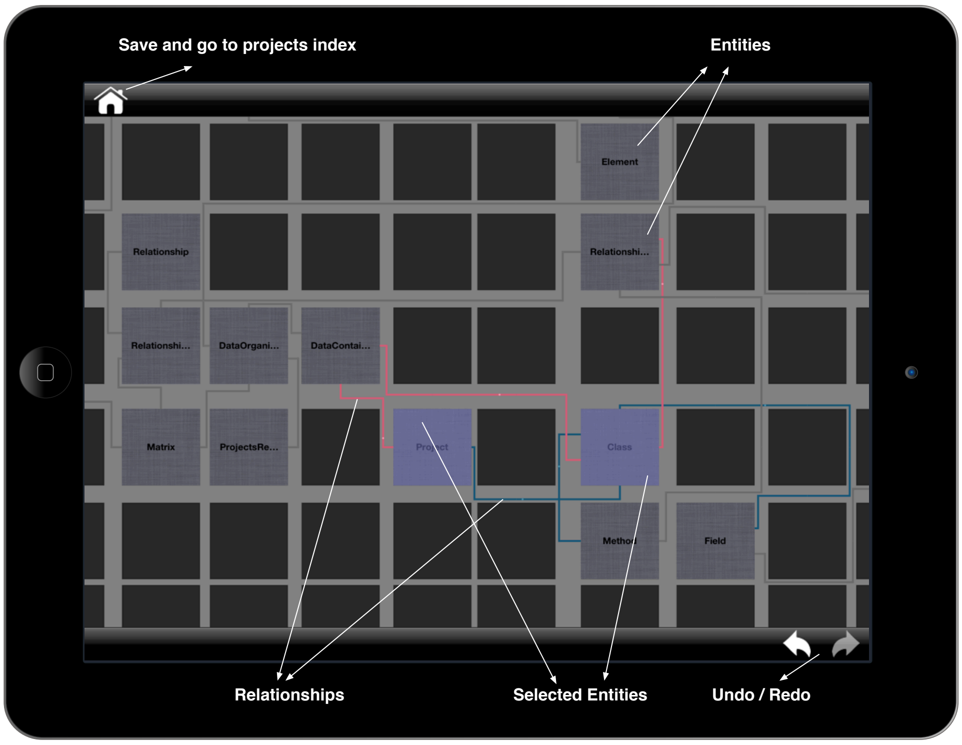
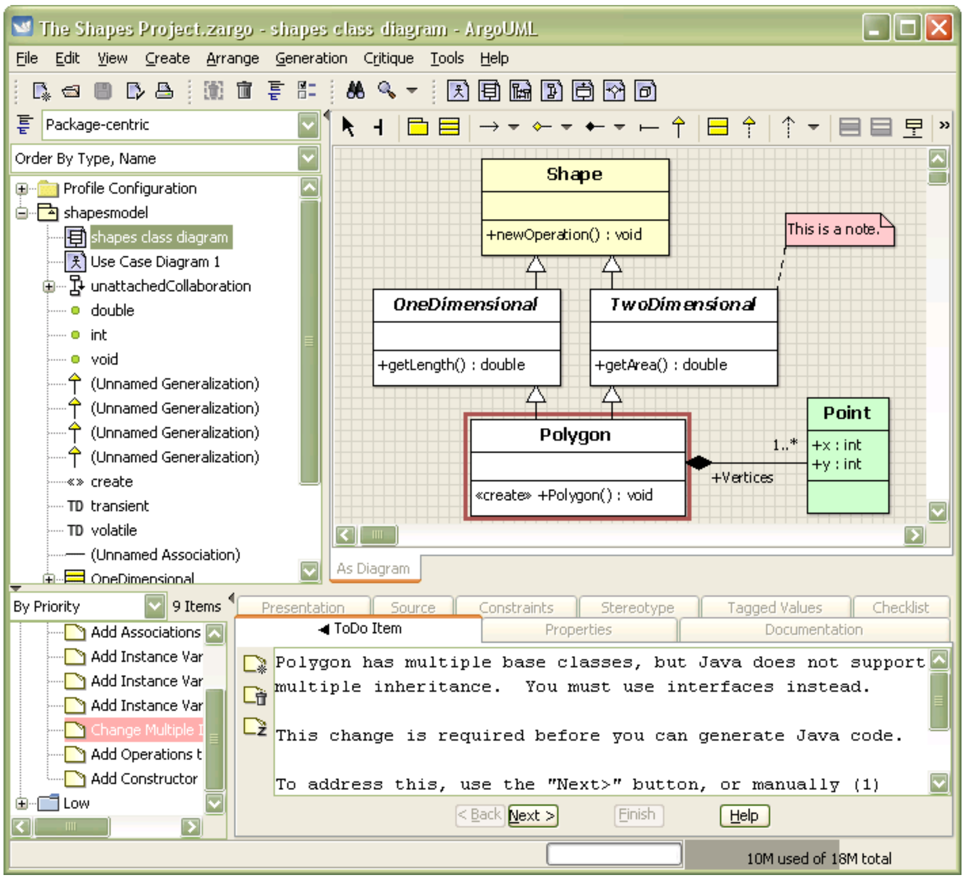


Figure 2.6. Cell: a simple modeling tool

[1]`http://cel.inf.usi.ch`

Formal Modeling

In formal modeling practices, the models are described using a more complex and rigorous language, generally expressed through the use of diagrams and visual notations. The unified modeling language (UML) is the de-facto standard for describing models [41]. Many tools support the creation and edition of UML diagrams. Figure 2.7 depicts ArgoUML, an open source UML modeling tool [2]. ArgoUML is an interactive, graphical software design environment that supports the design, development and documentation of object-oriented software applications.



```
Polygon has multiple base classes, but Java does not support
multiple inheritance.  You must use interfaces instead.

This change is required before you can generate Java code.

To address this, use the "Next>" button, or manually (1)
```

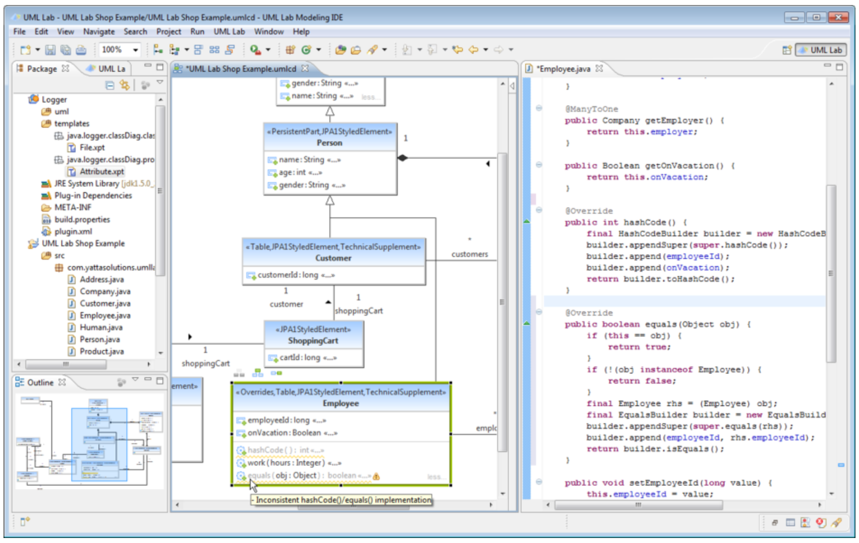
Figure 2.7. Argo UML: a complete UML environment

[2]http://argouml.tigris.org

Summary

Similarly to the alternative code-centric environments, modeling tools make use of fine-grained manipulation of the concepts, arranged within diagrams to leverage the use of spatial memory, and presented as editable graphical elements which abstract from an otherwise cumbersome textual representation.

The limitations of the UML-based tools are associated to the laying out of the elements in the diagrams, because choosing an arrangement is a non-trivial task when depicting large scale diagrams [42]. The visual presentation can be as disorganized and confusing as the original source code. There are some UML tools that automatically transform diagrams into source code, by generating templates of code from the diagrams. The tools can also infer the diagrams from the source code, thus they are deemed round-trip engineering tools. For instance, UML Lab [3] is a modeling IDE where design and implementation of software remain consistent at all times, see Figure 2.8. Even though external modeling tools are better suited for program comprehension than IDEs, 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 [43]. Moreover, the developer has to be aware of two artifacts which stem from different stages of the development process, increasing the cognitive overhead.


UML Lab - UML Lab Shop Example/UML Lab Shop Example.umlcd - UML Lab Modeling IDE
File Edit View Navigate Search Project Run UML Lab Window Help
UML Lab
Package
*UML Lab Shop Example.umlcd
*Employee.java
Outline
Person
Customer
ShoppingCart
Employee
customers
customer
shoppingCart
Inconsistent hashCode()/equals() implementation

```
    }

    @ManyToOne
    public Company getEmployer() {
        return this.employer;
    }

    public Boolean getOnVacation() {
        return this.onVacation;
    }

    @Override
    public int hashCode() {
        final HashCodeBuilder builder = new HashCodeB
        builder.appendSuper(super.hashCode());
        builder.append(employeeId);
        builder.append(onVacation);
        return builder.toHashCode();
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (!(obj instanceof Employee)) {
            return false;
        }
        final Employee rhs = (Employee) obj;
        final EqualsBuilder builder = new EqualsBuild
        builder.appendSuper(super.equals(rhs));
        builder.append(employeeId, rhs.employeeId);
        return builder.isEquals();
    }

    public void setEmployeeId(long value) {
        this.employeeId = value;
```

Figure 2.8. Modeling tools

[3] http://www.uml-lab.com/en/uml-lab/features/roundtrip/

2.4 Direct-Manipulation of Objects

In this section we describe an alternative metaphor for building development environments, based on *direct manipulation* of objects.

Hutchins *et al.* state that direct manipulation systems continuously present the objects of interest, replace the use of complex syntax by the use of labeled buttons (amongst other widgets), and provide reversible operations specified in a higher level language interface, whose impact is shown immediately. The effect of direct manipulation in the interfaces reduces the distance between the user's intentions and the facilities provided by the system, therefore reducing the effort of the user to accomplish goals [44].

The alternative code-centric environments and modeling tools we reviewed in the previous section, use some form of direct manipulation of either of portions of source-code defining classes, methods and packages, or the components of a model. We discuss next the two contrasting metaphors for building interfaces, and describe the environments that were designed to minimize the presence of tools, in favor of direct manipulation of objects.

2.4.1 The Tools and Desktop Metaphor

The mission of the XEROX Palo Alto Research Center (PARC) was to leverage the potential of computers in the late 60's. The team led by Alan Kay sought to accomplish that goal by making computers accessible to everyone, not just technical people. Kay and his fellow scientists conceived the nowadays omnipresent concept of a *personal computer*, with a graphical user interface (GUI) based on overlapping windows, menus, and icons [7]. Kay was building on the previous work of researchers such as Douglas Engelbart, head of the Stanford Augmentation Research Center Lab, who designed the mouse and the first GUI.

The Smalltalk-72 system, implemented in 1972, was the first environment to include a GUI, which included *tools* contained within the windows, that presented the content of the system: the objects. The objects were modified by performing textual edits with the keyboard, and pointing to actions in the contextual menus with the mouse. Nowadays, many environments use a similar interface to the Smalltalk systems, giving prominence to the tools.

On one hand, we have the *Tools metaphor*, and on the other hand we have the *Desktop metaphor*. The interfaces of both metaphors consisted on a graphical user interface including windows, menus, icons and a mouse. They differ in the target of the actions, and the primary role within the interface.

In tool-based interfaces, the users first open the tool, and then select the content to work upon, *i.e.*, which may be data file or an object. Whereas, in interfaces using a desktop-metaphor the users do not invoke the tools, but rather interact with the objects themselves. The desktop is a working environment, which resembles the top of an office desk were various pictures of familiar objects reside: documents, folders, file drawers, *etc.*. The pictures are named icons, and are the visible concrete embodiments of the corresponding physical objects. Icons represent the objects in the form of a picture that react to the actions of the user. Editing the objects involves invoking the associated tool by pointing and clicking the icon with the mouse, or on a menu [12].

The desktop metaphor pushes users toward their data rather than toward the tools, employing analogies with the physical world, at the expense of coupling the objects to assigned tool. In this dissertation, we explore a system that merges the icon with the associated tool into a single graphical representation of the objects populating an infinite desktop, an **Object-Focused Environment**.

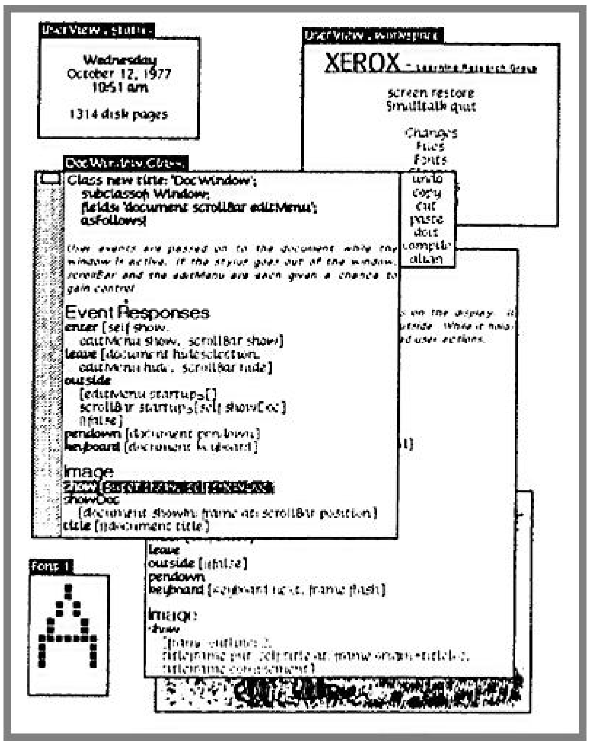
The ubiquitous association between an object and a single tool, strengthens the notion that the manipulable "icon" is the object, which provides all the necessary features to change the object. For example, the tool representing a class of the system must provide features for renaming, adding or removing methods and variables, navigating to related classes and to the containing package. We describe the genesis of object-focused environments in the following sections.

2.4.2 From Structured Text to Graphical Objects

The first Smalltalk systems presented the objects as structured text contained in the windows of the system. Figure 2.9(a) depicts a Smalltalk 76 system, the central window is presenting the class named *DocWindow*, subclass of Window, displaying several methods of the Event Responses and Image protocols.

In Smalltalk 76, the tool support for modifying the classes and methods of the system was based on editing a textual representation of the objects [45]. Modern IDEs for mainstream OOP languages still rely on this interface, a code-centric interface with augmented text editors for changing the system. We described the mainstream IDEs in Chapter 2.2.

The following versions of Smalltalk replaced the structured text representation and manipulation for a tool-based interface, that abstracted away from the source-code. Figure 2.9(b) depicts a Smalltalk 80 system with several opened windows containing the tools, the most important being a *system browser*, designed by Larry Tesler: a multi-paned class "browser" which allowed one to quickly and easily traverse all the classes and methods in the system [7].



Wednesday
October 12, 1977
10:51 am

1314 disk pages

screen restore
Smalltalk quit

Changes
Files
Fonts

```
Class new title: 'DocWindow';
   subclassof Window;
   fields: 'document scrollBar editMenu';
   asFollows!
```

User events are passed on to the document while the window is active. If the stylus goes out of the window, scrollBar and the editMenu are each given a chance to gain control

Event Responses

```
enter [self show.
   editMenu show.  scrollBar show]
leave [document hideselection.
   editMenu hide.  scrollBar hide]
outside
   [editMenu startup⇒[]
   scrollBar startup⇒[self showDoc]
   ⇑false]
pendown [document pendown]
keyboard [document keyboard]
```

Image

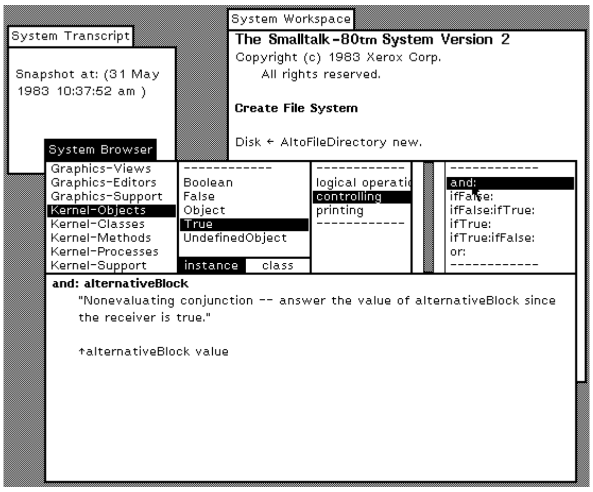
```
show [illegible]
showDoc
   [document showin: frame at: scrollBar position]
title [⇑document title]
```

```
leave
outside [⇑false]
pendown
keyboard [keyboard next. frame flash]
```

Image

```
show
   [illegible]
```

(a) Smalltalk 76: Structured Text



Snapshot at: (31 May 1983 10:37:52 am)

The Smalltalk-80tm System Version 2
Copyright (c) 1983 Xerox Corp.
All rights reserved.

Create File System

Disk ← AltoFileDirectory new.

```
and: alternativeBlock
    "Nonevaluating conjunction -- answer the value of alternativeBlock since
    the receiver is true."

    ↑alternativeBlock value
```

(b) Smalltalk-80: The Browser

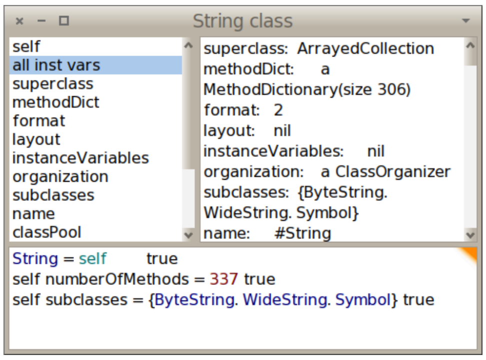
Figure 2.9. The evolution of the Smalltalk programming tools

Smalltalk systems include the *Inspector* tool, for visualizing the state of any object, enabling change by sending messages in an embedded code pane, to satisfy a fundamental design principle of Smalltalk which states that: "every object should be able to present itself in a meaningful way for observation and manipulation" [4]. Figure 2.10 depicts a standard Smalltalk object inspector tool, and a modern version part of the Glamour framework, inspired from the Mac Finder where every object can have multiple presentations through which it can be interacted with [4].

The inspector presents any object, including the ones used to create programs: classes, methods, and packages. A Smalltalk programmer can modify the programs by inspecting classes and sending messages in the code pane.

On one hand, many different types of objects make up the program in a class-based OOP language. On the other hand, prototype-based OOP languages omit the class and instance dichotomy, in favor of a uniform object model with parent delegation. The environments of the latter do not distinguish between *browsers* and *inspectors*, the uniformity of the language enabling the use of a single tool for manipulating all the objects.

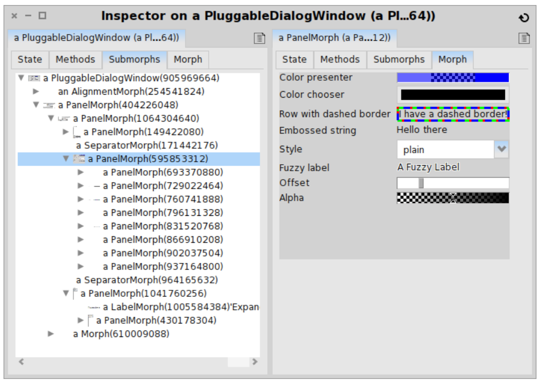
Next, we discuss Self, the seminal OOP language based on prototypes, adopting an object-focused environment based on the desktop metaphor.



```
self
all inst vars
superclass
methodDict
format
layout
instanceVariables
organization
subclasses
name
classPool
```

```
superclass:  ArrayedCollection
methodDict:    a
MethodDictionary(size 306)
format:  2
layout:   nil
instanceVariables:    nil
organization:   a ClassOrganizer
subclasses:  {ByteString.
WideString. Symbol}
name:    #String
```

```
String = self        true
self numberOfMethods = 337 true
self subclasses = {ByteString. WideString. Symbol} true
```


Inspector on a PluggableDialogWindow (a Pl...64))
a PluggableDialogWindow (a Pl...64))
State
Methods
Submorphs
Morph
a PanelMorph (a Pa...12))
State
Methods
Submorphs
Morph

```
▼ a PluggableDialogWindow(905969664)
    ▶ an AlignmentMorph(254541824)
    ▼ a PanelMorph(404226048)
        ▼ a PanelMorph(1064304640)
            ▶ a PanelMorph(149422080)
            a SeparatorMorph(171442176)
            ▼ a PanelMorph(595853312)
                ▶ a PanelMorph(693370880)
                ▶ a PanelMorph(729022464)
                ▶ a PanelMorph(760741888)
                ▶ a PanelMorph(796131328)
                ▶ a PanelMorph(831520768)
                ▶ a PanelMorph(866910208)
                ▶ a PanelMorph(902037504)
                ▶ a PanelMorph(937164800)
            a SeparatorMorph(964165632)
            ▼ a PanelMorph(1041760256)
                a LabelMorph(1005584384)'Expan
                ▶ a PanelMorph(430178304)
        ▶ a Morph(610009088)
```

Color presenter	
Color chooser	
Row with dashed border	I have a dashed border!
Embossed string	Hello there
Style	plain
Fuzzy label	A Fuzzy Label
Offset	
Alpha	

(a) The Inspector (b) The Glamourous Inspector

Figure 2.10. The inspector: a generic tool for manipulating objects

[4]http://www.moosetechnology.org/tools/glamoroustoolkit

2.4.3 Self

Self is a seminal object-focused environment [46]. Self is both a language and an interface for direct manipulation of uniform graphical objects that populate a malleable world. Figure 2.11 depicts the Self environment.

The term object-focused environment was coined by Ungar *et al.* [11], because Self fosters the notion that developers are in direct contact with the objects themselves, by coupling operations and representations 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 Figure 2.11. Self is based on prototypes instead of classes and instances: the language includes a single kind of object. The uniformity permeates throughout the whole interface, where any Self object has a single outliner tool that reveals the inner structure and provides the means to manipulate itself [46].

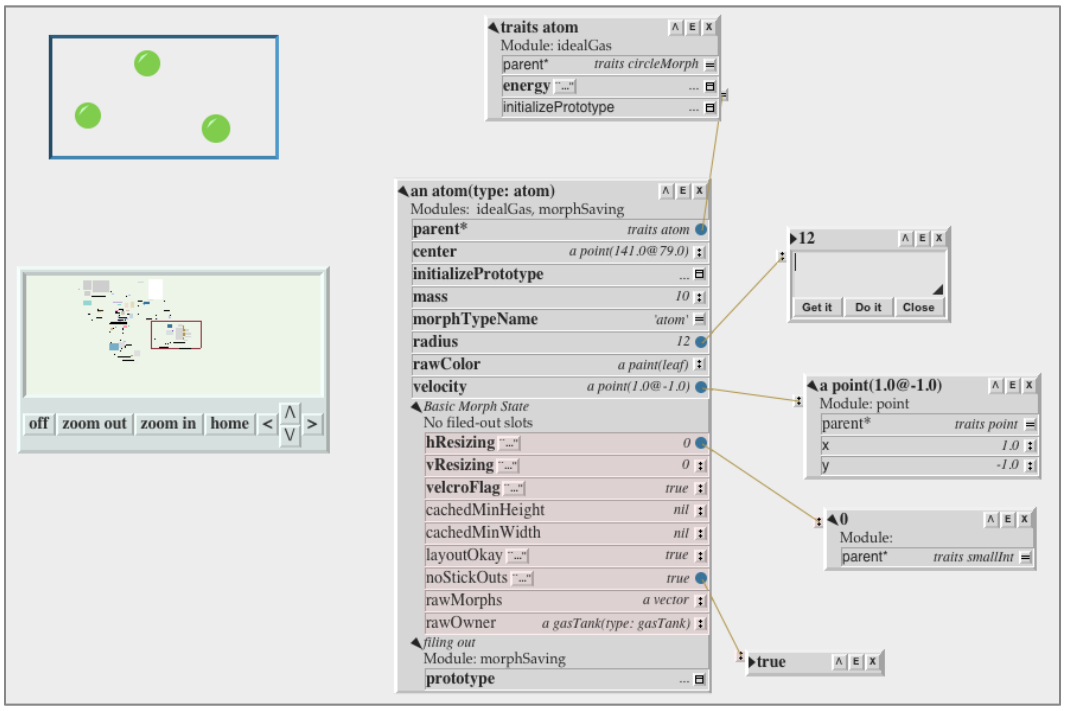


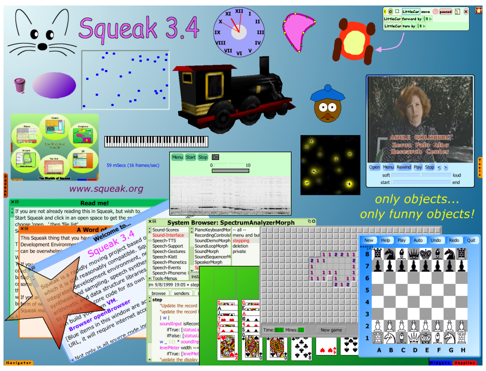
Figure 2.11. Self: the seminal object-focused environment

2.4.4 Morphic

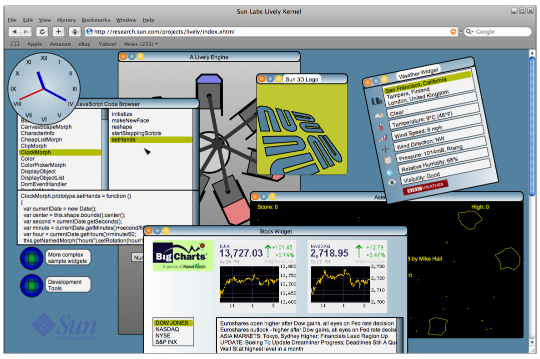
Morphic is a user interface framework designed to support direct-manipulation of graphical objects named Morphs. Morphic first appeared in the SELF environment [47], then in the Squeak Smalltalk environment [48], and lastly in the Lively Kernel environment for web development [49] (see Figure 2.12).

Directness and liveness are the two design principles behind Morphic, consisting in a simple architecture based on composable morphs, which define a visual appearance, a set of children in the form of sub-morphs, and an event handler for the mouse and keyboard events [49].

While Morphic allows one to easily navigate between live objects to its source-code, using the standard tools of the environment, the framework itself was not designed for the manipulation of the objects that make up the program, *i.e.* classes, methods and packages. Nevertheless, the framework provides the means to construct higher-level composite morphs from a set of basic ones, that can better accommodate for custom operations related to performing system changes.



(a) Squeak



(b) Lively Kernel

Figure 2.12. The Morphic UI framework implemented in Squeak Smalltalk and the Lively Kernel for Javascript

2.4.5 Naked Objects

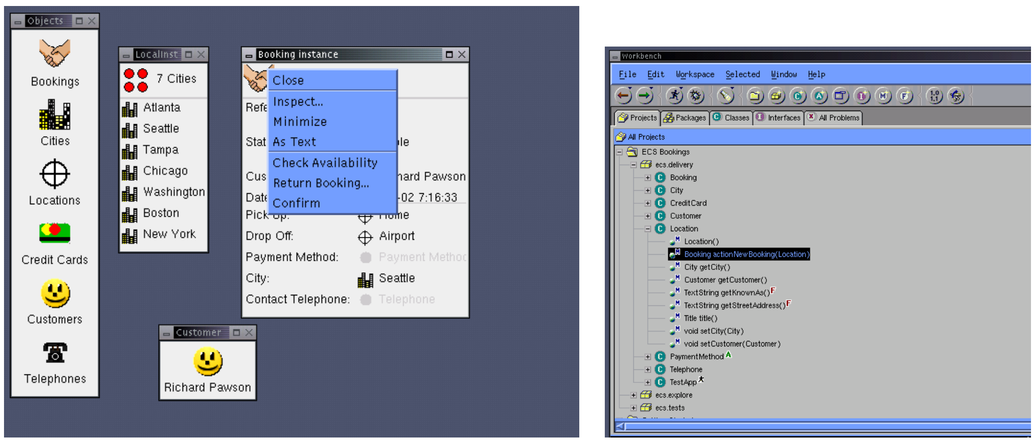
The Naked Objects framework was designed to support the Ph.D. thesis of Pawson [50], aimed at solving the problem of separation of procedure and data in OOP, resulting from the implementation of user interfaces.

Pawson remarks that the objects representing the business entities, Customer, Product or Order, are often behaviorally weak, and that the missing functionality resides in procedures external to the object, which hinders behavioral completeness in object designs.

Thus, in naked objects all the domain objects are exposed explicitly, such that all user actions consist of viewing objects, and invoking behaviors on them by using contextual menus [50].

As opposed to traditional user interface frameworks, the presentation layer is provided automatically. Naked objects involves auto-generating a user interface from the business model definitions. Figure 2.13(a) depicts a user interface for a booking application, which includes objects such as customers, cities, bookings, and locations.

The *objects* that make up the program are manipulated with a custom IDE, in the spirit of traditional code-centric IDEs. A naked objects environment enables direct-manipulation of instances of a running program, whereas packages, classes, methods and unit tests are edited in a separate manner. Figure 2.13(b) depicts the developer's view of the project, the IDE of the framework.



(a) The User Interface (b) The Developer's View

Figure 2.13. The Naked Objects Framework

2.5 Conclusions

We described the genesis of our approach, starting from the early Smalltalk systems with tool-based interfaces, to direct manipulation environments of uniform objects in the case of Self, and of instances of the running programs as in the Naked Objects framework.

In this dissertation we investigate a development environment for class-based OOP languages, that embraces the simplicity and directness of the interfaces of Self, of Naked Objects, modeling environments, and the alternative code-centric IDEs. The directness results from designing the interface on the desktop metaphor, as opposed to the tools metaphor. The reviewed interfaces alleviate many of the problems discussed in Section 1.5 related the real state management, the visualization and the need for a fine-grained manipulation.

The IDE of a Naked Objects implementation enables modifying the programs –the packages, classes, methods and unit tests– by turning to the developer's view of the project, which uses the standard interface of code-centric IDEs. We advocate for an environment that applies the same directness to the programming interface. On the other hand, the alternative code-centric IDEs provide direct manipulation of the program, but do not provide for the manipulation of the instances.

In the Self programming language and environment both the instances and the objects of the program co-exist in the interface, similarly to the Smalltalk systems, and the uniform object model based on prototypes enables the use of a single tool for manipulating them, minimizing the presence of tools into an *object-focused environment*.

Self is a prototype-based language that deliberately omits the concept of classes and instances. In our work we focus on class-based OOP languages, thus we investigate how to augment an object-focused environment 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 manipulating diverse objects such as classes, meta-classes, methods, packages, tests, and code-critic rules.

Chapter 3

Object-Focused Environments

In the previous chapter of this dissertation we focused on the tools that enable programming, from the early punch card machines to the graphical interfaces of personal computers, including the main tool used for the crafting of computer systems: the integrated development environment.

Drawing inspiration from several modeling tools, alternative code-centric environments, Self, and the Naked Objects framework, we advocate for development environments that make use of direct manipulation of objects. The directness of the interface, and an unconstrained layout of fine grained graphical elements helps to alleviate the conceptual and technical problems of IDEs designed around a tool-based interface, described in Section 1.5.

In this chapter we motivate the need for an object-focused environment that minimizes the presence of tools in favor of the manipulation of objects, and present our implementation of such an environment, named Gaucho. The interface of Gaucho is built on a desktop metaphor that can accommodate an extensible set of software engineering tasks.

Structure of the chapter. We start by relating object oriented programming to modeling in Section 3.1, by discussing several scenarios depicting how different software engineering practices solve a programming task. In Section 3.2 we present an overview of the main concepts in Gaucho. We present our vision for an integral modeling environment in Section 3.3. We describe the support in Gaucho for OOP in Section 3.4, and conclude this chapter by reflecting on the advantages of an object-focused environment in Section 3.5.

3.1 Motivation

OOP was conceived as a vehicle to craft models of reality, by focusing on the essence of the design. However, mainstream OOP languages obfuscate the modeling step with their static textual focus, leading to a loss in flexibility and added accidental complexity. On the other hand, model driven development focuses on the high level views of the system, based on rigid notations that hinder exploratory programming. We advocate an integral object modeling environment, which enables the building of software systems *represented by* models, where everything is described in terms of a uniform metaphor: objects and message passing; the programs, the models and the meta-models coexist transparently, favoring interoperability and awareness of each other.

3.1.1 From reality to the program

Software engineering promotes modeling as a means to understand and map a portion of reality to solve a particular problem by engineering a software system. This process can be termed *programming*, and the outcome is a machine-executable computer model [39].

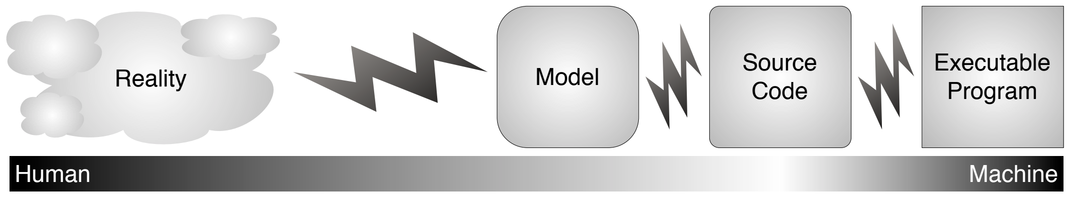


Figure 3.1. Our view on the programming process

Programming is a highly complex activity involving several steps. To simplify our discussion, we distinguish four steps that span from reality to a concrete executable computer model, see Figure 3.1. A specific artifact pertains to each step: In the first step the relevant artifact is reality itself, in the second a model of reality, in the third the source code implementing the model of reality, and in the last step the executable program. Each artifact results from applying transformations to the artifact of the previous step. For instance, a model is abstracted from reality, and a program is compiled from its source code.

None of the three leaps depicted in Figure 3.1 are performed without effort, and this is where the complexity of crafting software comes from. If we could measure the effort in distance, the first leap would be the longest of all. As

Dijkstra put it, "Humans are, undoubtedly, the main artists in the art of programming" [3]. They are in charge of the initial leap over the chasm of reality to the model, from a fuzzy real world to a more or less formal model.

The more we move away from reality towards the executable computer model, the more a machine becomes responsible. In traditional programming, source code is compiled to an executable program by machine compilers. In Model Driven Engineering (MDE), the human created explicit models can be made executable by machine automatic code generation or model interpretation. The separation of concerns is the main reason behind the division of the programming process into steps. Favre mentioned the existence of diverse technological spaces (TS), which provide different sets of associated concepts, tools and possibilities [51]. Each artifact concerns a different aspect of the system under study, therefore each step potentially involves a different TS.

Herein lies the problem: The use of heterogenous artifacts hinders the interoperability between the steps depicted in Figure 3.1. For instance, mainstream OOP languages enforce a distinction between compile-time and run-time. This distinction prohibits changes in the source code to immediately affect the executable program, because the transformation between the third and fourth step (of Figure 3.1) is unidirectional. Thus, changes in one artifact deem obsolete the corresponding artifact of the next step. Practically, this means terminating the executable program, modifying the source code, recompiling and running the program again.

Alan Kay, paraphrasing William of Occam, claimed that entities should not be multiplied unnecessarily. Kay proposed to describe computer models based on recursion: Designing the parts to have the same power as the whole—to reduce complexity—as in one of the most effective applications of this technique, *i.e.* object-oriented design [52]. A supporting evidence of the practical applications of (pure) OOP languages, such as Smalltalk [4], is that the previous dichotomy between compile-time and run-time does not occur in Smalltalk environments.

The notion that everything is a model appears to subsume the prevailing notion that everything is an object. MDE emphasizes this distinction: Bézivin argues [39] that the idea of software systems being composed of interconnected objects—the OOP view—is not in opposition with the idea of software being viewed as a chain of model transformations—the MDE view.

Our vision is to step away from the model as a high level view of the system, that can be translated into platform specific implementations by automatic code generation, closing the gap between the model and the executable programs. Our ultimate goal is to support the programming process with an environment that embraces both the unification power of models, as stated by Bézivin [39], and the uniform metaphor rule stated by Ingalls [4], by posing that *everything is an object, and the software system under construction can be described in terms of models composed of many views or perspectives*.

3.1.2 Scenarios

We investigate the differences and similarities between various software engineering practices currently in use, differentiated in the manner they approach the programming process. We start by describing an abstract scenario to provide a framework of reference from which the differences in each practice can be derived. We provide an overview of how each practice approaches the scenario. Lastly, we reflect on the advantages and drawbacks of our envisioned practice, compared to the traditional ones.

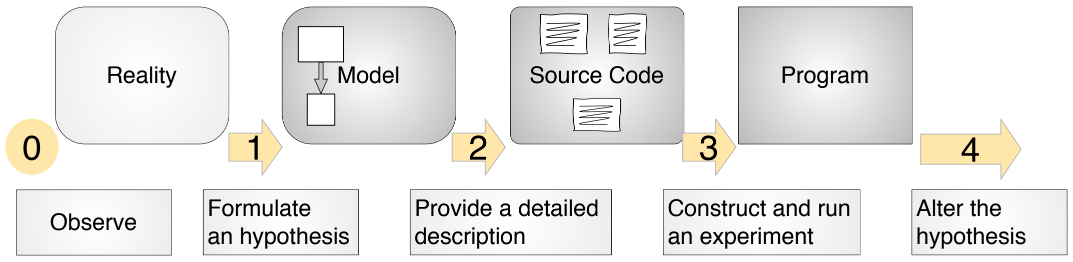


Figure 3.2. An abstract scenario of our view on the programming process

Abstract Scenario

Adele works as a senior developer at a software company focused on medical applications. The project manager has requested her to produce a simulation program, to simulate the growth of a generic multicellular organism from the early stages of conception to its demise. Figure 3.2 depicts our view on the programming process, in the context of this scenario.

To complete her task, Adele performs a sequence of actions. In particular she

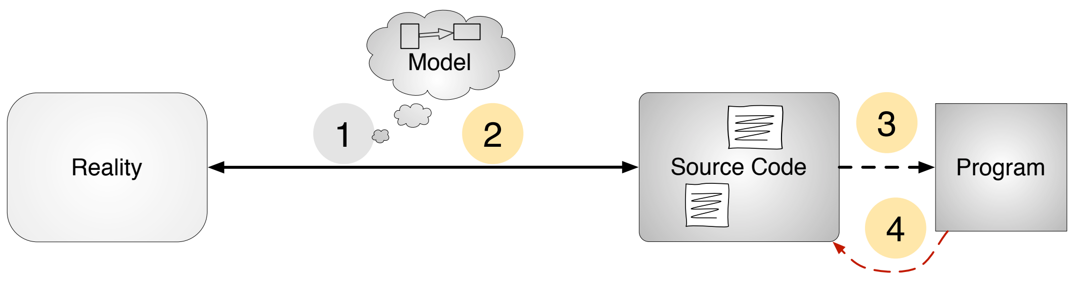
1. starts by observing the reality, to understand the laws that govern the growth and composition of multicellular organisms;

2. distinguishes three entities: a simulation, a cell, and a multicellular organism. She describes a model, capturing her current knowledge of the domain;

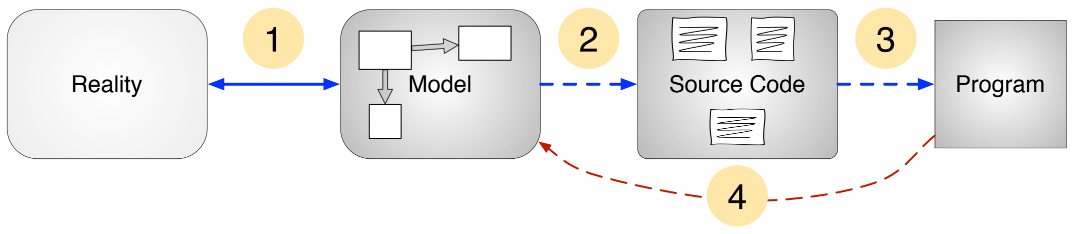
3. provides a more detailed specification of the behavior and composition of the model;

4. creates an executable computer model from the specification, and runs the simulation;

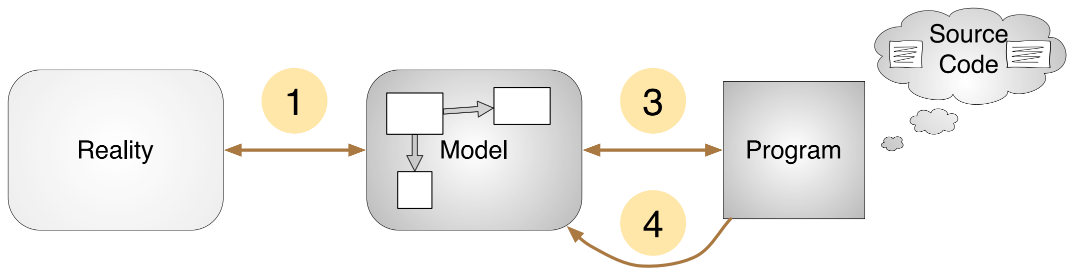
5. notices that she missed to distinguish an *environment* entity, thus needing to alter the initial model.



(a) Code centric workflow



(b) MDE workflow



(c) Object focused workflow

Figure 3.3. The workflows of the three practices under consideration

We describe how this happens for three software engineering practices, the *code centric* approach (*e.g.* by using a mainstream OOP like Java), the *MDE* approach, and the *object-focused modeling* approach, which is our vision. In Figure 3.3 we detail the workflow of each practice.

Code Centric Workflow

Mainstream OOP practices revolve around the notion of source code. Models are only implicitly part of the programming process, therefore Adele omits describing it, and goes directly to writing the source code. She refines the specification by typing precise class and method definitions. Afterwards she compiles and runs an executable program. When Adele notices that she missed to distinguish the environment entity, she is forced to discard the executable, then add a new class definition to the source code, and perform the necessary textual editions to integrate it into the existing model. Given that the source code and the executable are in different technological spaces, the program is deemed obsolete because the source code and the executable program cannot directly interoperate with each other. The latter is the main reason why code-centric practices hinder flexible modeling activities.

MDE Workflow

Adele describes the model using a modeling language, such as UML. Then she uses MDE tools to automatically generate the source code, to ultimately compile the source code into a platform dependent computer executable program. Model transformation (MT) is an essential part of MDE, thus several tools—termed roundtrip engineering tools—support UML transformations and synchronization between the diagrams and the source code. The advantage is that they automatically bridge the gap between the diagrams and the code, enabling the use of high level views for comprehending the model without losing the features offered by the tools for writing the programs.

Roundtrip engineering tools have not been widely adopted because the diagrams are inert models, while the source code remains the complete specification of the system under construction [53]. The latter tools are mainly used as palliatives for resolving navigation and comprehension issues when dealing with source code [37]. The value of MDE models increases when they serve other purposes than just contemplative models used only for documentation or visualization purposes [39]. This occurs when MDE models are made executable by introducing a set of possible actions together with a complete code generation or model interpretation engine. The drawbacks are that MDE tools are simpler than a full blown object-oriented programming language, generally based on a state machine, although recent work is improving the capabilities of such tools. Moreover, to validate the simulation model, Adele has to open another environment to run the validation, thus hindering interoperability which prevents, for instance that the model is automatically made aware of the latest validation and transformation results.

Object focused Workflow

In our vision, Adele describes the model using a visual high level language, which depicts the object model view on the simulation system under construction. Instead of source code, the program's full specification is the model itself, and the notion of code is used for specifying behavior at the method level. Adele writes methods such as *simulate* and *terminate*, and associates them to the simulation model. The program consists of a world of collaborating objects, such as the simulation model and meta-model, the simulation object and the multicellular organism object. For instance, when Adele introduces the concept of *environment* in the model, she simply resends the message simulate to the simulation and the program continues running, now collaborating with an environment object. In the object-focused workflow, the running program, the model and its multiple views, the model transformations and model validators, all coexist in the *same* environment, and are manipulated using the same tools. Objects and message passing are the means to describe the meta-models, models, and the programs. This enables full interoperability between all artifacts.

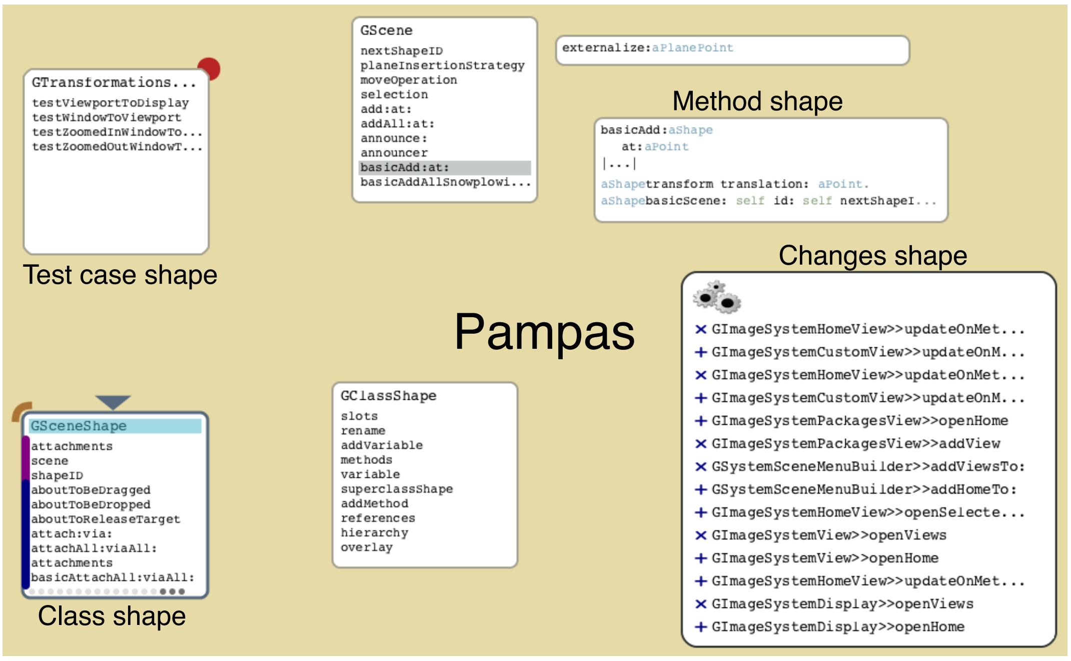
3.1.3 Summary

We described the programming process, and the different steps involved in crafting computer systems with a purpose, *i.e.* to solve a problem from reality. We presented the differences between software engineering practices that guide the process of crafting software systems, and revealed them by means of a scenario. We discussed the drawbacks of traditional code centric practices, and current model based practices: the diversity of technological spaces that the produced artifacts live in hinders interoperability between them, and more importantly they push the program too far away from the models. We proposed a vision where all the steps involved in the programming process transparently interoperate, by strictly adhering to the unifying principle which states that everything is an object. Next, we describe the embodiment of that vision.

3.2 Gaucho

In this section, we present Gaucho, an object-focused environment which minimizes the presence of tools in favor of the manipulation of objects. The interface of Gaucho is built on a desktop metaphor, that can accommodate for modeling activities, for programming in OOP languages, and to support a group of collaborating programmers while performing programming sessions.

3.2.1 Gaucho in a Nutshell



```
GTransformations...
testViewportToDisplay
testWindowToViewport
testZoomedInWindowTo...
testZoomedOutWindowT...
```

```
GScene
nextShapeID
planeInsertionStrategy
moveOperation
selection
add:at:
addAll:at:
announce:
announcer
basicAdd:at:
basicAddAllSnowplowi...
```

```
externalize:aPlanePoint
```

```
basicAdd:aShape
   at:aPoint
|...|
aShapetransform translation: aPoint.
aShapebasicScene: self id: self nextShapeI...
```

```
× GImageSystemHomeView>>updateOnMet...
+ GImageSystemCustomView>>updateOnM...
× GImageSystemHomeView>>updateOnMet...
+ GImageSystemCustomView>>updateOnM...
+ GImageSystemPackagesView>>openHome
× GImageSystemPackagesView>>addView
× GSystemSceneMenuBuilder>>addViewsTo:
+ GSystemSceneMenuBuilder>>addHomeTo:
+ GImageSystemHomeView>>openSelecte...
× GImageSystemView>>openViews
+ GImageSystemView>>openHome
+ GImageSystemHomeView>>updateOnMet...
× GImageSystemDisplay>>openViews
+ GImageSystemDisplay>>openHome
```

```
GSceneShape
attachments
scene
shapeID
aboutToBeDragged
aboutToBeDropped
aboutToReleaseTarget
attach:via:
attachAll:viaAll:
attachments
basicAttachAll:viaAll:
```

```
GClassShape
slots
rename
addVariable
methods
variable
superclassShape
addMethod
references
hierarchy
overlay
```

Figure 3.4. Gaucho: an object-focused environment

Gaucho [29] is an object-focused environment for the crafting of models composed of objects, described in Object-Oriented languages, depicted in Figure 3.5. The two pivotal concepts in Gaucho are:

1. *Pampas*: a pannable and zoomable 2D surface, hosting shapes.

2. *Shapes*: graphical elements which are high-level representations of the objects that make up the software system (e.g. instances, classes, methods, packages, test cases, models).

Gaucho is a term used to describe the "cowboys" of the pampas, the vast South American grasslands. We use the gauchos analogy to denote the freedom that developers have with Gaucho: they can place elements arbitrarily and can directly manipulate them through an uniform set of actions, regardless of the type of visual elements in focus and the presented level of detail.

Gaucho is available at `gaucho.inf.usi.ch`.

3.2.2 The Sessions and The System

The infrastructure behind the interface of Gaucho includes multiple sessions during which programmers make changes to the objects of the underlying system. In Gaucho, all programming activities take part in the context of a session. The session tracks any fine-grained change to the system, which result from manipulating the shapes that represent the modified objects.

Figure 3.5 depicts the overall design of Gaucho. Programmers may open many different sessions, to work on different systems. The systems consist in programs, models, or other meaningful abstraction described by an object oriented programming language.

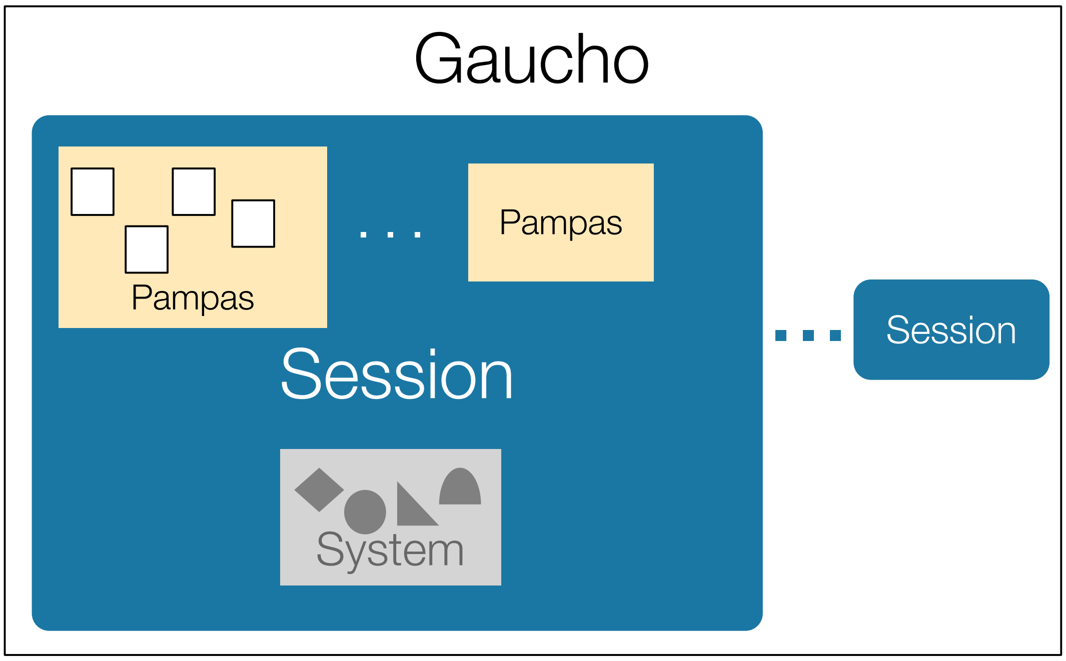


Figure 3.5. The overall design of Gaucho

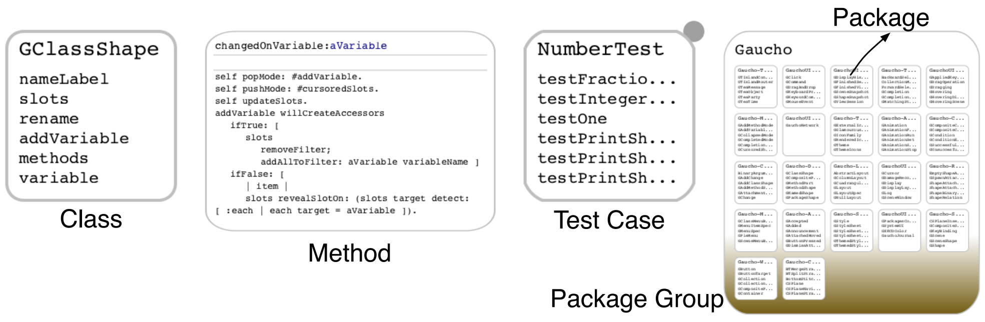
3.2.3 The Pampas

The Pampas is a two-dimensional surface, which hosts the visual objects that represent entities that make up a software system (e.g., packages, classes, methods, sessions, recent changes, etc.) in the form of shapes.

The pampas is where all programming tasks are performed. During a gaucho session, programmers perform changes to the system, interacting with the shapes that are laid out in the pampas. The shapes are freely placed on the pampas, and gently push each other away to avoid overlapping, to ease the task of arranging them into disjoint groups.

Gaucho is a zoomable user interface [54], i.e., the pampas can be panned and zoomed, enabling developers to fully customize different perspectives on the system under construction. A session can include one or more pampas, to provide multiple perspectives on the same system. A pampas forms a working set of objects for a particular task. In the pampas the objects may be opened (and closed) on demand by simply typing their names in the pampas, enabling programers to filter those which are relevant to the task at hand.

3.2.4 The Shapes



```
changedOnVariable:aVariable

self popMode: #addVariable.
self pushMode: #cursoredSlots.
self updateSlots.
addVariable willCreateAccessors
   ifTrue: [
      slots
         removeFilter;
         addAllToFilter: aVariable variableName ]
   ifFalse: [
      | item |
      slots revealSlotOn: (slots target detect:
[ :each | each target = aVariable ]).
```

Figure 3.6. Distinct *Gaucho* shapes populating a pampas

The presence of directly manipulable representations of objects is a central requirement of an object-focused environment. The system is viewed and modified, solely by interacting with the graphical depictions of the objects, thus minimizing the use of tools, and supporting the illusion that the graphical elements in the interface are the objects of the underlying system. In other words, an object and its single associated tool are one.

In gaucho the graphical elements are called Shapes. In our object-focused environment, every distinct object has its own graphical counterpart, a directly manipulable element that uniquely identifies the object within the interface. For example, a *class shape* depicts the structure of the represented class, and a *method shape* presents the method signature and the statements that make up the body. Figure 3.6 depicts several *Gaucho* shapes that represent classes, methods, test-cases and packages of the system under construction. The sessions track fine-grained changes to the system, made by uniform set of interactions with the shapes, which are based on a custom keyboard shortcuts, and the use of contextual menus.

When Ingalls introduced the interface for Smalltalk 76 [45], he stated: *"The reader may find himself evaluating the user interface presented in the figures. This misses the point, which is the aptness of communicating objects in describing the situation"*. We quoted Ingalls, to clarify that the visual appearance of the shapes is important, yet many plausible designs exist to convey the structure, composition, and semantics of the objects. The choices we made while designing the Shapes, are based on the effectiveness on which they convey the above features, regardless of the aesthetic quality of the the representations.

3.2.5 Implementation

Gaucho is written in Smalltalk [35], and implemented on top of Pharo, an open source Smalltalk system. Smalltalk is a highly dynamic and fully reflective language, were everything is an object: the classes, meta-classes, packages, and methods are first class objects. The uniformity of the language enables to one perform all the fine-grained changes to the system, by sending messages to the objects via their graphical counterparts, *i.e.* the shapes. *Gaucho* is drawn with Cairo [1], a vectorial rendering library written in C. The non-overlapping scheme of the pampas is based on the corner stitched layout, which provides an efficient structure and algorithms for the handling of a 2D region, divided into tiles [55]. 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.

[1] `http://cairographics.org/`

3.3 Modeling with Gaucho

OOP languages encourage viewing programs as a world of collaborating objects, which were instanced from a design composed of classes—the model—that conforms to a fixed meta-model, the meta-classes [51].

In our work, we focus on the "pure" OOP languages, such as Smalltalk [4] and Self [46]. Due to the language's faithful adherence to the notion that everything is an object, running programs can gracefully coexist with the rest of the system, in a fully object oriented environment, enabling changes to the meta-model, the model and the program in the same technological space [56].

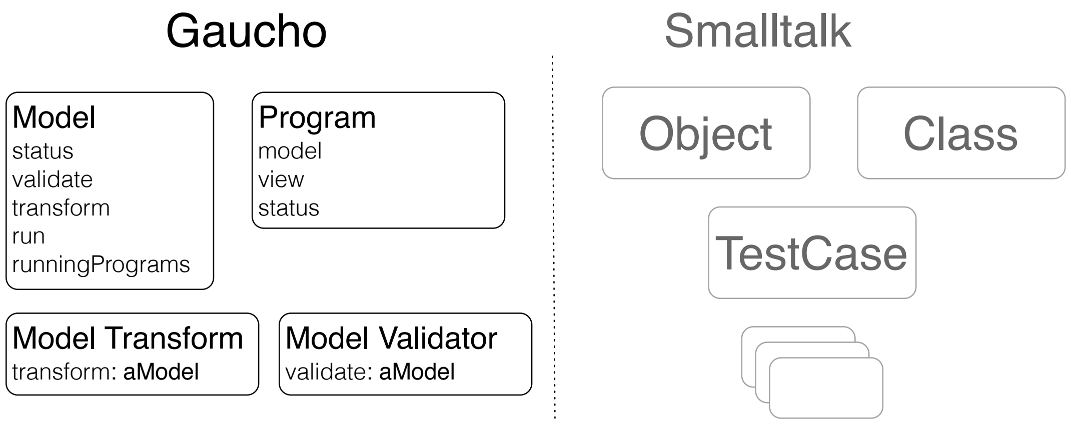


Figure 3.7. First-class Model enhancements to Smalltalk

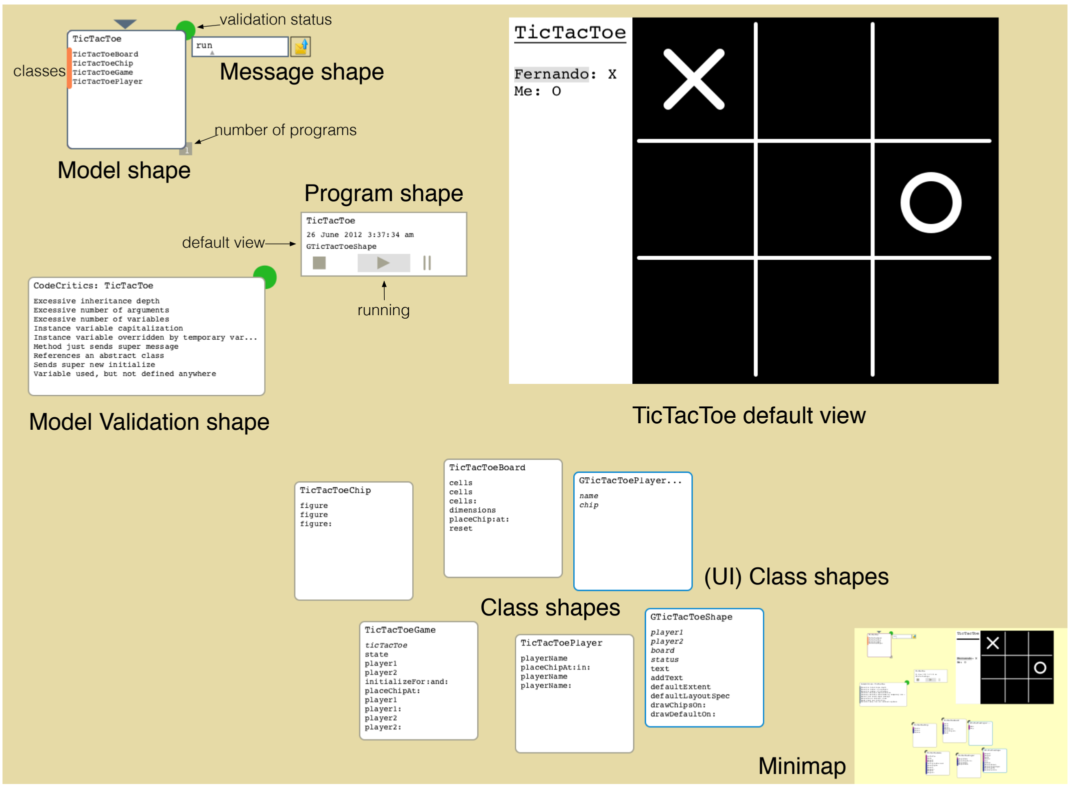
We introduced a first-class presence of a *Model* to our base language. As mentioned before, Smalltalk lacks the explicit presence of a model, thus we added several classes that add support for MDE within the system. Figure 3.7 depicts the most relevant classes and their behavior.

For instance, a model *i.e.* an instance of the class *Model* understands the following messages: run, validate, transform, and status. These instances coexist with the classes, test cases, views, transformations and validations. All these objects are part of the same environment, thus they are aware of each other and are manipulated using the same tools.

3.3.1 The MDE Shapes

Gaucho includes custom shapes for the added classes to Smalltalk, those which reify the concept of a Model, Model Validations and running Programs.

Figure 3.8 portrays a Gaucho session populated by all the relevant artifacts related to modeling a TicTacToe game [2].



```
TicTacToe
TicTacToeBoard
TicTacToeChip
TicTacToeGame
TicTacToePlayer

run

TicTacToe
26 June 2012 3:37:34 am
GTicTacToeShape

CodeCritics: TicTacToe
Excessive inheritance depth
Excessive number of arguments
Excessive number of variables
Instance variable capitalization
Instance variable overridden by temporary var...
Method just sends super message
References an abstract class
Sends super new initialize
Variable used, but not defined anywhere

TicTacToe
Fernando: X
Me: O

TicTacToeChip
figure
figure
figure:

TicTacToeBoard
cells
cells
cells:
dimensions
placeChip:at:
reset

GTicTacToePlayer...
name
chip

TicTacToeGame
ticTacToe
state
player1
player2
initializeFor:and:
placeChipAt:
player1
player1:
player2
player2:

TicTacToePlayer
playerName
placeChipAt:in:
playerName
playerName:

GTicTacToeShape
player1
player2
board
status
text
addText
defaultExtent
defaultLayoutSpec
drawChipsOn:
drawDefaultOn:
```

Figure 3.8. MDE shapes: TicTacToe example

The main entity is the TicTacToe model itself, which includes references to the classes that compose the model, and the running programs. Similarly to a Test Case Shape, a Model Shape depicts a visual cue denoting the status of the last validation run, in the form of a top right dot, colored to reflect the status: invalid is red and valid is green.

To instantiate programs, or validate a model, developers send messages to its shape, first selecting the shape and second typing the name of the action to perform. For instance, Figure 3.8 depicts the message named *run* being sent to the TicTacToe model shape, which will respond by instantiating another TicTacToe program in the current Gaucho session.

Programs may be started, stopped and paused, by manipulating the ProgramShape. The first-class presence of Programs in Gaucho, enables a more

[2]`wikipedia.org/wiki/Tic-tac-toe`

precise interaction than the traditional ad-hoc nature of programs in Smalltalk. Programs have views, that in our example consists of the TicTacToeShape depicting a running game, the board, the chips and the players.

In this example, the model validation is composed by a CodeCriticValidator, the default validator in Gaucho. Figure 3.8 portrays a model validator shape, displaying all the passed and violated rules of the TicTacToe model. We envision adding shapes pertaining to model validation and model consistency checking, for describing and running validations of the models against their meta-models, similarly to the test case and test run shapes for assessing the validity of classes and methods.

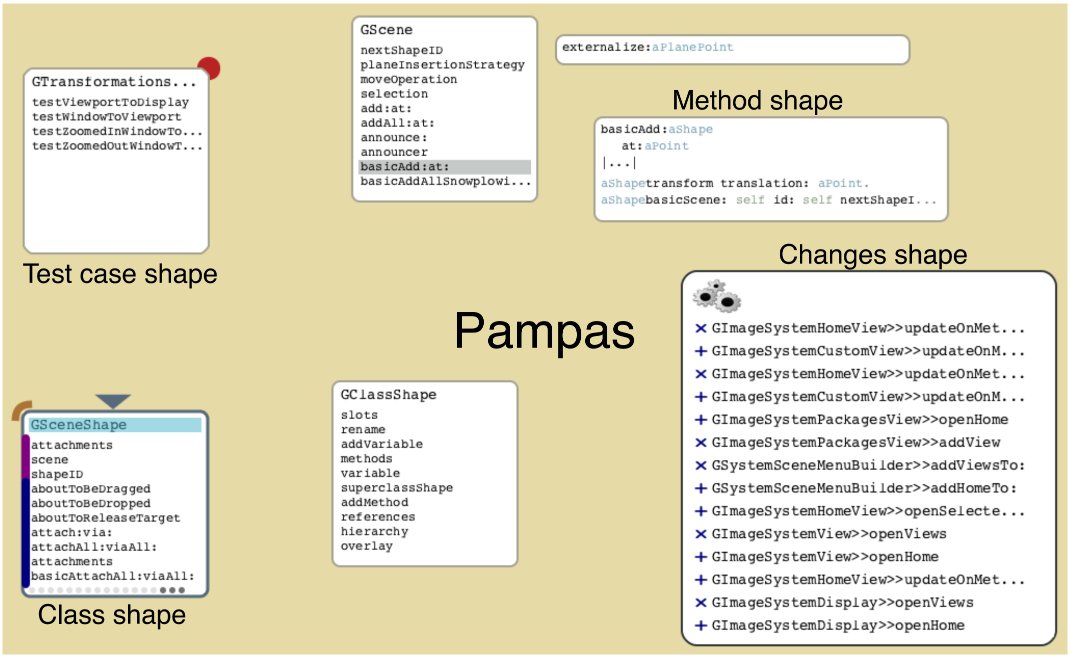
Our goal is to devise an integral modeling environment which enables the building of software systems, making use of a single metaphor, applied uniformly throughout the whole environment. We named our vision **Object-focused Environments** and describe it as a software engineering practice which remains faithful to the principle that everything is an object, and provides a flexible visual diagrammatic language for describing and manipulating all the artifacts that make up the system in an uniform manner.

3.4 Programming in Gaucho

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. 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 investigate 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.

The logical view of the system is the one describing the object model of the system [57]. 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.



```
GTransformations...
testViewportToDisplay
testWindowToViewport
testZoomedInWindowTo...
testZoomedOutWindowT...
```

```
GScene
nextShapeID
planeInsertionStrategy
moveOperation
selection
add:at:
addAll:at:
announce:
announcer
basicAdd:at:
basicAddAllSnowplowi...
```

```
externalize:aPlanePoint
```

```
basicAdd:aShape
   at:aPoint
|...|
aShapetransform translation: aPoint.
aShapebasicScene: self id: self nextShapeI...
```

```
× GImageSystemHomeView>>updateOnMet...
+ GImageSystemCustomView>>updateOnM...
× GImageSystemHomeView>>updateOnMet...
+ GImageSystemCustomView>>updateOnM...
+ GImageSystemPackagesView>>openHome
× GImageSystemPackagesView>>addView
× GSystemSceneMenuBuilder>>addViewsTo:
+ GSystemSceneMenuBuilder>>addHomeTo:
+ GImageSystemHomeView>>openSelecte...
× GImageSystemView>>openViews
+ GImageSystemView>>openHome
+ GImageSystemHomeView>>updateOnMet...
× GImageSystemDisplay>>openViews
+ GImageSystemDisplay>>openHome
```

```
GSceneShape
attachments
scene
shapeID
aboutToBeDragged
aboutToBeDropped
aboutToReleaseTarget
attach:via:
attachAll:viaAll:
attachments
basicAttachAll:viaAll:
```

```
GClassShape
slots
rename
addVariable
methods
variable
superclassShape
addMethod
references
hierarchy
overlay
```

Figure 3.9. Programming in Gaucho

Figure 3.9 includes several shapes depicting Gaucho itself: The class shape GSceneShape presents its attributes and methods, and provides means to perform modifications to the class; the test case shape GShapeTest enumerates all the test methods and their status, enabling the creation, deletion, modification and running of the tests. The methods, spawned next to the class they belong, present their signature and statements for code editions.

3.4.1 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. Figure 3.10 depicts a class shape on the class named *GClassShape*. The shape is presented in three different modes: focused, unfocused, and filtered.

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,

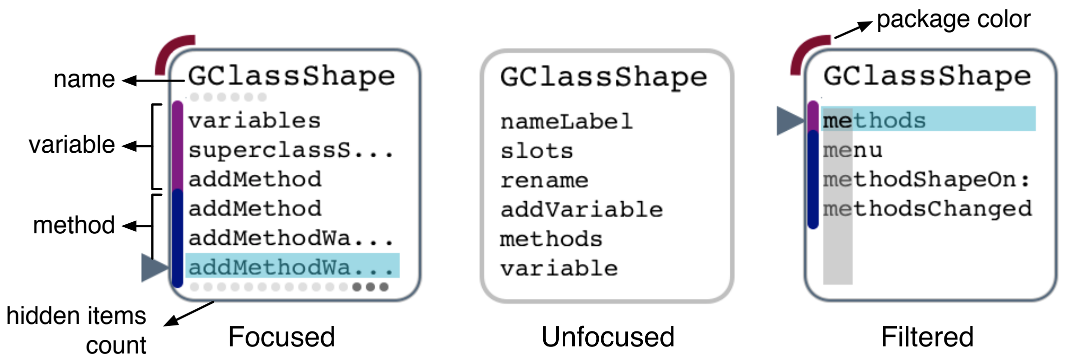


Figure 3.10. The shape representing the class shape itself

to ease the comprehension of its composition, therefore the shape only reveals extra information on selection or when it is the focus of attention. The focused and filtered shapes depicted in Figure 3.10 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, grey for class variables, and dark blue for methods). The arc to the top left indicates the unique color of the containing package, and provides a one click access to the package shape.

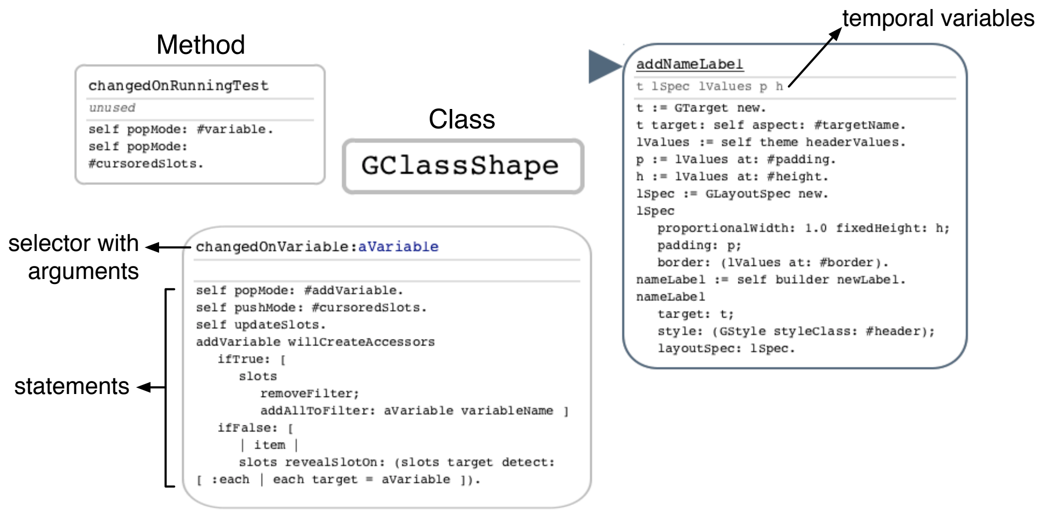
A class may contain a large number of methods and variables, thus the class shape can be filtered to reveal only those items that match a regular expression, narrowing down the displayed items to ease the location of the class components . For example, in Figure 3.10 the filtered class shape shows only the methods and variables that match the pattern *me*∗.

3.4.2 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. 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*. Figure 3.11 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 se-



```
changedOnRunningTest
unused
self popMode: #variable.
self popMode:
#cursoredSlots.
```

```
changedOnVariable:aVariable

self popMode: #addVariable.
self pushMode: #cursoredSlots.
self updateSlots.
addVariable willCreateAccessors
   ifTrue: [
      slots
         removeFilter;
         addAllToFilter: aVariable variableName ]
   ifFalse: [
      | item |
      slots revealSlotOn: (slots target detect:
[ :each | each target = aVariable ]).
```

```
addNameLabel
t lSpec lValues p h
t := GTarget new.
t target: self aspect: #targetName.
lValues := self theme headerValues.
p := lValues at: #padding.
h := lValues at: #height.
lSpec := GLayoutSpec new.
lSpec
   proportionalWidth: 1.0 fixedHeight: h;
   padding: p;
   border: (lValues at: #border).
nameLabel := self builder newLabel.
nameLabel
   target: t;
   style: (GStyle styleClass: #header);
   layoutSpec: lSpec.
```

Figure 3.11. A class with several of its methods surrounding the shape

lector 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.

3.4.3 Test Cases

The SUnit framework introduced testing into the software development process of Object-Oriented languages [58]. Since then it has become common practice to devise unitary tests that assert the correct implementation of a given feature.

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. Figure 3.12 depicts three Test Case shapes with different test results: untested, failing, passing. Typing a custom keyboard command on a focused test case shape runs the test case, and the shape reflects the last run status.

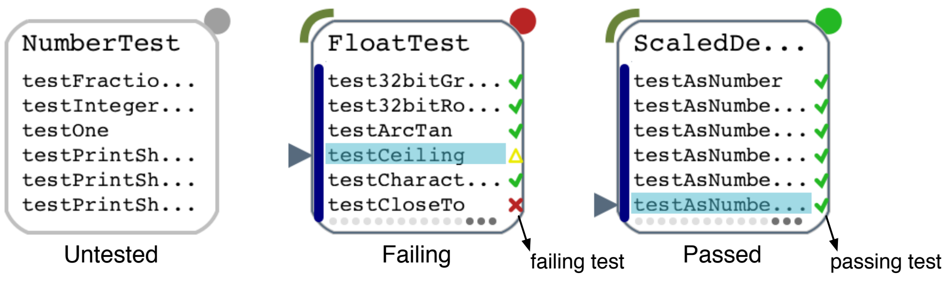


Figure 3.12. Test Case shapes with different test results

3.4.4 Packages

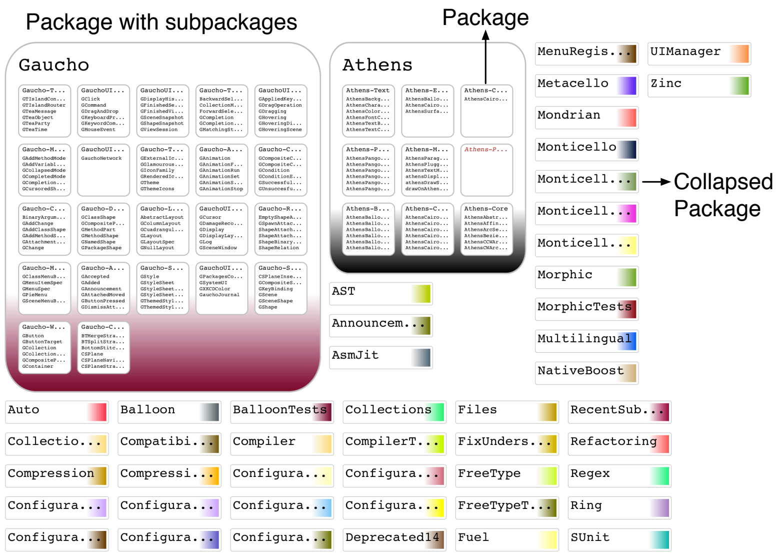


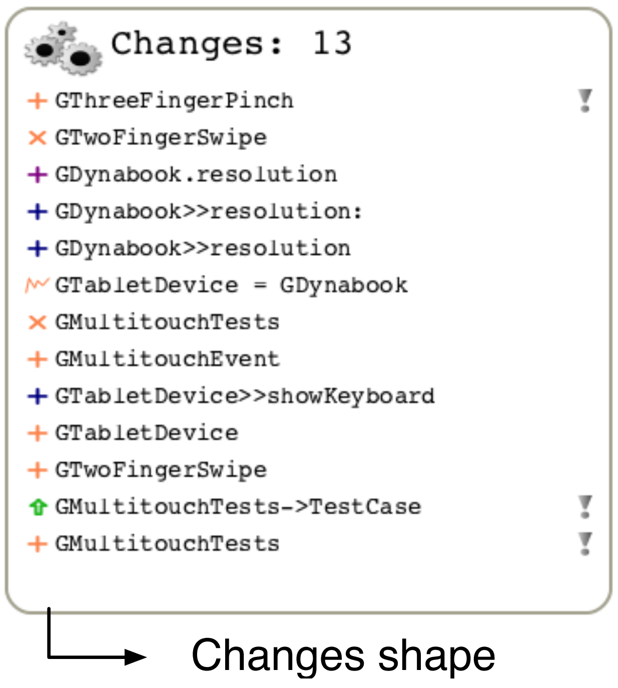
Figure 3.13. The system view presents all the available packages

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. The system view lays out all the package group shapes of the system using a rectangle packing algorithm to optimize the use of real estate, by packing a subset of the rectangles into a bigger rectangle to maximize the total profit of rectangles packed [59]. After the initial arrangement, the developer may move, collapse or expand any package, customizing the overall view of the system. The packages of the system are uniquely identified by a color used throughout the interface, such as the colored top right visual arc that denotes the package of a class shape (cf. Figure 3.10).

3.4.5 Producing Fine-Grained Changes

In Gaucho, the programmers change the system by interacting with the shapes. The changes occur in the context of a development session, which tracks them, and presents them for manipulation. Figure 3.14 depicts the *Changes shape*, which presents all the modifications to the system made so far.

The fine-grained changes result from invoking custom widgets that reify modifications to the objects; for instance, renaming a package or adding a method to a class. Every change has a unique icon, chosen to maximize the affordance of each changing operation. The shape depicted in Figure 3.14, includes many different changes, such as: adding the class GThreeFingerPinch, removing the class GTwoFingerSwiper, renaming the class GTabletDevice to GDynabook, adding the method named #showkeyboard to the class GTabletDevice, and changing the superclass of GMultitouchTests to TestCase.


Changes: 13

```
+ GThreeFingerPinch
× GTwoFingerSwipe
+ GDynabook.resolution
+ GDynabook>>resolution:
+ GDynabook>>resolution
  GTabletDevice = GDynabook
× GMultitouchTests
+ GMultitouchEvent
+ GTabletDevice>>showKeyboard
+ GTabletDevice
+ GTwoFingerSwipe
  GMultitouchTests->TestCase
+ GMultitouchTests
```

Figure 3.14. Presenting the Changes of a Session

Class Creation

Figure 3.15 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.

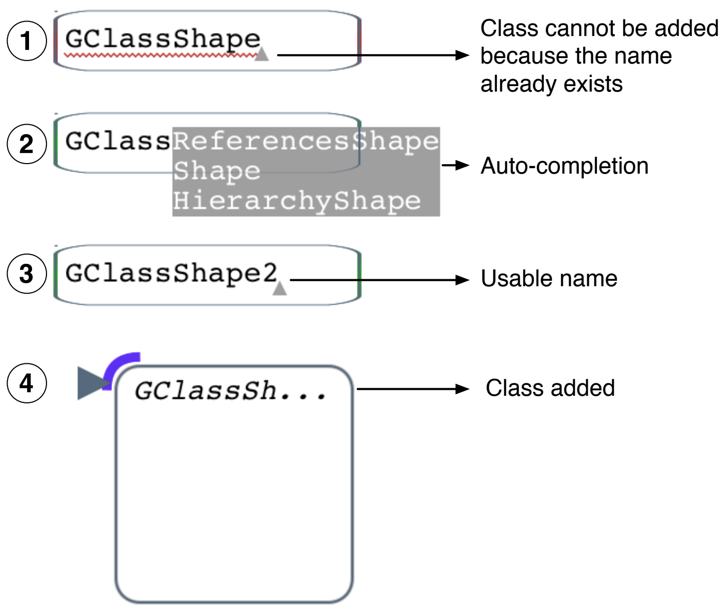


Figure 3.15. Adding classes in Gaucho

Figure 3.15 illustrates the steps to add a new class. The developer focuses on the Pampas, then presses a keyboard command to:

1. open the *add class widget*
2. the widget helps the developer in typing the new class name, 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.
3. once the developer types in a valid new class name, by pressing enter an empty class is created
4. the shape is opened, replacing the widget.
5. the change is instantly reflected on the changes shape of the ongoing session.

Changing a Class

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 menus.

The programmer invokes the changing widgets by opening a contextual menu on the shape. *Gaucho* provides a more focused interaction than with general purpose tools, thus the user quickly gets used to the mouse and keyboard actions for manipulating the shapes. The programmer can either use a pie or a "daisy" menu, according the the settings of the environment.

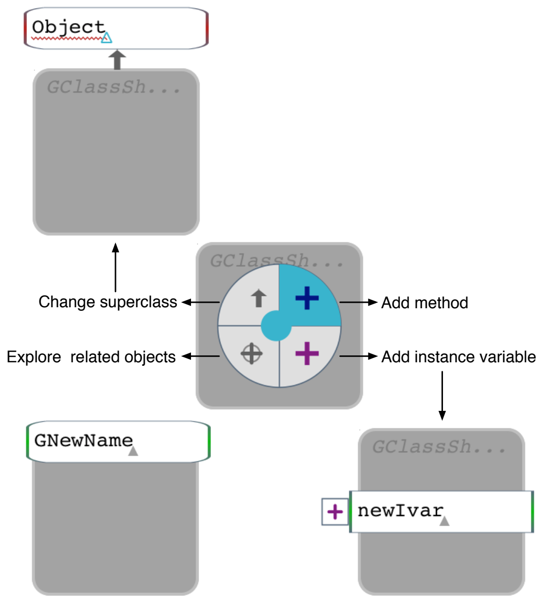


Figure 3.16. Changing a Class with the Pie Menu

Figure 3.16 depicts the *pie menu* that enables modifying a class shape. We chose to use pie menus instead of traditional list-based menus, because when the set of operations is small, they ease pointing and selecting the options [60].

On the other hand, the programmer can use a *daisy menu* (see Figure 3.17). In the daisy menu, the options surround the shape, providing feedback on the status and outcome of the modification in a more permanent manner than a transient pie menu, to ease the adoption of the interface for beginners.

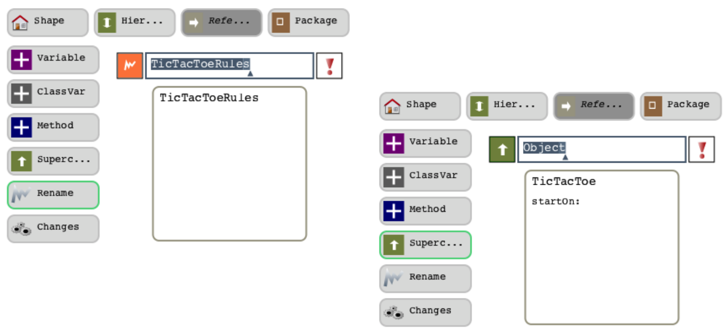


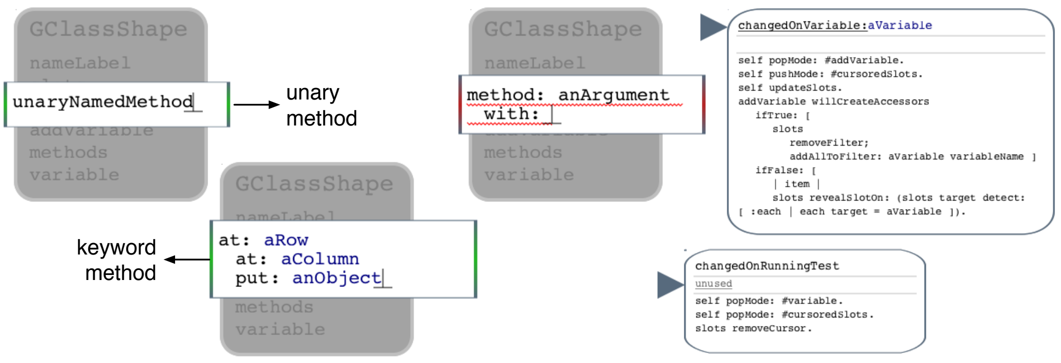
Figure 3.17. Changing a Class with the Daisy Menu

Changing a Method

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 rearranges 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. Figure 3.18 depicts two method shapes with the cursor placed on the selector. The uppermost method shape to the right, is focused on the method named *changedOnVariable:*, and clicking would open the rename method widget. The method shape to the bottom, is focused on an temporal variable of the method named *changedOnRunningTest*, to open a rename widget on the selected temporal variable.

3.4.6 Navigating the System

An object-focused environment must provide the means to explore the system by interacting with the shapes, to avoid the use of external tools which are not part to the shape itself. A study by Sillito *et al.* on developers habits [10], suggests that developers start programming sessions by finding initial focus points, and then continue expanding those points by exploring relationships between the software artifacts, *i.e.*, the objects. Other researchers stated that a direct tool support for interactive exploration helps when performing program comprehension tasks [37].



```
unaryNamedMethod
```

```
at: aRow
  at: aColumn
  put: anObject
```

```
method: anArgument
  with:
```

```
changedOnVariable:aVariable

self popMode: #addVariable.
self pushMode: #cursoredSlots.
self updateSlots.
addVariable willCreateAccessors
   ifTrue: [
      slots
         removeFilter;
         addAllToFilter: aVariable variableName ]
   ifFalse: [
      | item |
      slots revealSlotOn: (slots target detect:
[ :each | each target = aVariable ]).
```

```
changedOnRunningTest
unused
self popMode: #variable.
self popMode: #cursoredSlots.
slots removeCursor.
```

Figure 3.18. Adding and modifying methods

The shapes in *Gaucho* include a customized set of navigation actions, invoked by menu items or keyboard shortcuts that provide quick access to exploring the connected artifacts. For instance, a class shape spawns both the group of class references and the class hierarchy, which are group shapes that enable further exploration within the system.

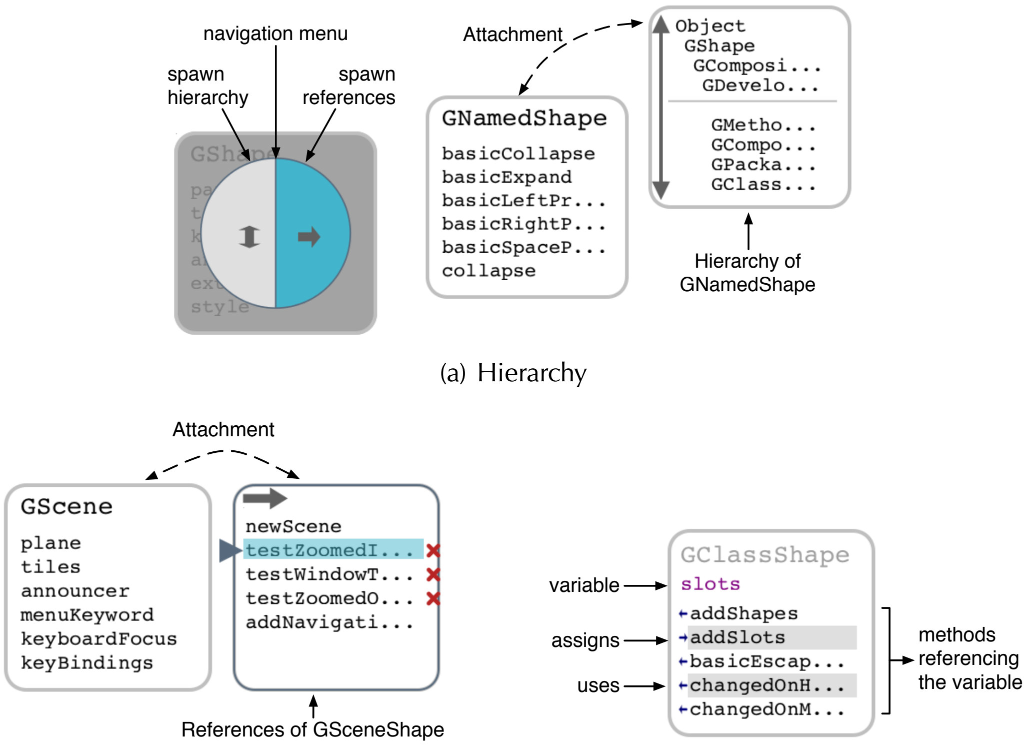
Figure 3.19(a) and 3.19(b) depict the result of spawning the hierarchy and the references of a class shape. The leftmost figure presents a class shape with a navigation menu with two options: spawn the group of references and open the hierarchy shape.

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 selection-dependent tools or tabs.

The spawned shapes in *Gaucho* are attached to the original shape, meaning that a subsequent move or a dismiss of the latter will be applied to the former, and since the represented object is always the same, the sense of connection between groups in the interface is stronger than with the use of general purpose tools or tabs. In other cases, an object-focused environment eliminates the need to spawn a distinct visual element to display the result of a query. For instance, a class shape can be asked to reveal all the methods that assign or use a variable.

System Navigation Example

The shapes enable the exploration of the interconnected entities across the system; the options are tailored to the type of each shape.



(a) Hierarchy

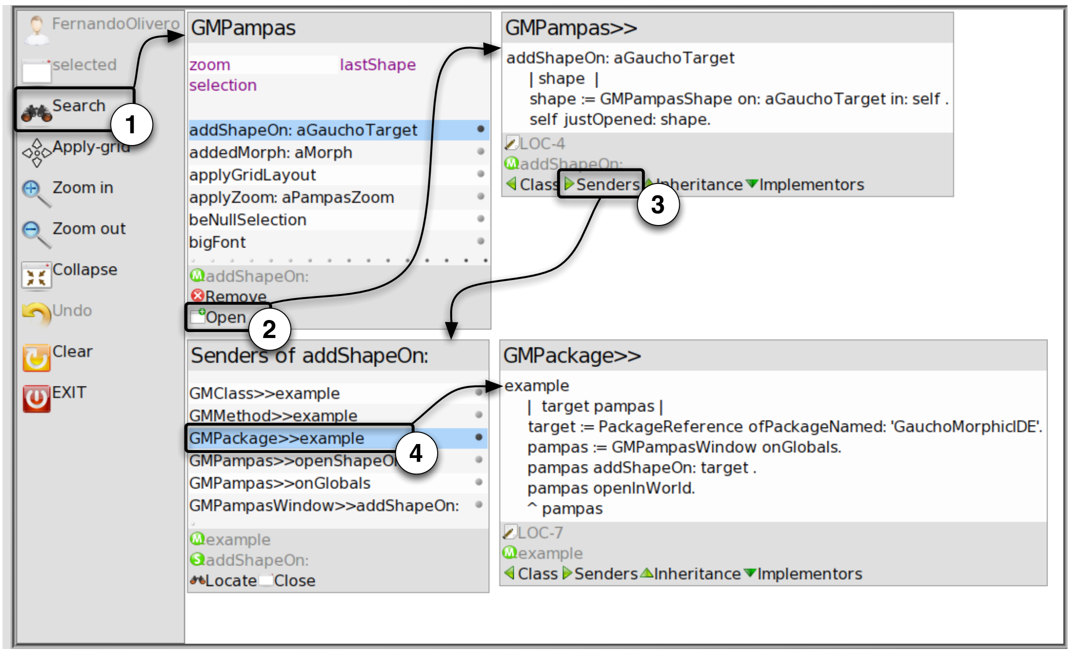
(b) References

(c) Variables

Figure 3.19. Navigating the system from a class shape

For instance, class shapes present icons to open the group of class references, the class hierarchy, the group of subclasses and the package of the class. Figure 3.20 shows a typical sequence of actions in Gaucho, when a developer interacts with the shapes to navigate the system. The figure depicts a previous version of Gaucho (1.2), where the icons were directly presented on the shape, whereas in the latest version (2.0) the options are only shown on demand, for a more concise appearance of the shapes. The actions are the following:

1. uses the search widget in the toolbar to locate the GMPampas class.
2. scrolls down the methods list and selects the method `addShapeOn:`, and presses the open button to open the method shape.
3. presses the senders button of the method shape to open the senders group.
4. selects the `example` method (of class GMPampas), pressing cmd-o to open a method shape on the selected method.



GMPampas

zoom lastShape
selection

```
addShapeOn: aGauchoTarget
addedMorph: aMorph
applyGridLayout
applyZoom: aPampasZoom
beNullSelection
bigFont
```

addShapeOn:
Remove
Open

GMPampas>>

```
addShapeOn: aGauchoTarget
    | shape |
    shape := GMPampasShape on: aGauchoTarget in: self .
    self justOpened: shape.
```

LOC-4
addShapeOn:
Class Senders Inheritance Implementors

Senders of addShapeOn:

```
GMClass>>example
GMMethod>>example
GMPackage>>example
GMPampas>>openShapeOn
GMPampas>>onGlobals
GMPampasWindow>>addShapeOn:
```

example
addShapeOn:
Locate Close

GMPackage>>

```
example
    |  target pampas |
    target := PackageReference ofPackageNamed: 'GauchoMorphicIDE'.
    pampas := GMPampasWindow onGlobals.
    pampas addShapeOn: target .
    pampas openInWorld.
    ^ pampas
```

LOC-7
example
Class Senders Inheritance Implementors

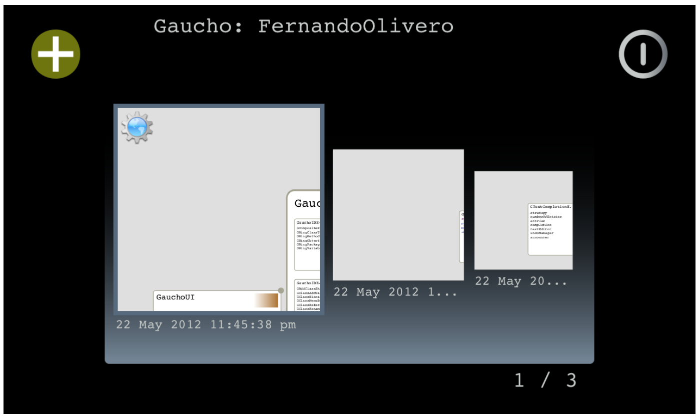
Figure 3.20. Navigating through interconnected shapes

3.4.7 Task Context Support

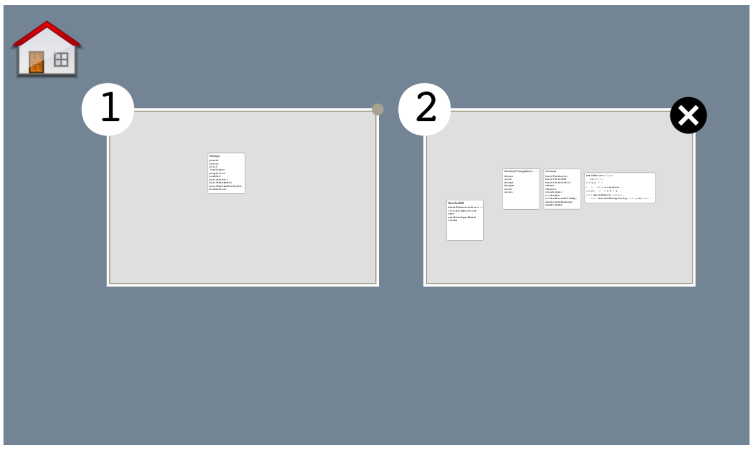
Developers start development sessions by finding initial focus points, and then continue expanding those points forming interconnected graphs [10]. We argue that navigating the system using traditional tool-based interfaces is hampered by the text-, list-, and tab-based nature of mainstream IDEs. For 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, producing back and forth navigation within the tools [17].

Gaucho programmers create their custom view of the system, by opening the shapes relevant to the task at hand. The environment does not automatically collect the relevant entities, as in Mylin's degree-of-interest model [13], 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.

Figure 3.21 depicts Gaucho on startup when developers choose to continue working on past development sessions or commence a new one. The bottom figure depicts the widget that enables managing the views within a session.



(a) Login by opening a saved session



(b) Manage the views within a session

Figure 3.21. Task context support within Gaucho

3.5 On the interface of 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 consistent manner. The shapes react uniformly to additions, removals, or modifications, produced by a 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 tool-based interfaces 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.

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. In traditional IDEs, 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, in *Gaucho* the classes are represented by a condensed graphical object that reveals more information on demand, when the shape is either focused or selected.

The concise view of the shapes eases understanding the structure of classes, to gain insights on the name and number of variables, on the amount of methods and their signature, and whether the class or a method are abstract. Figure 3.22 depicts several *Gaucho* shapes that illustrate the facilities for program comprehension tasks.

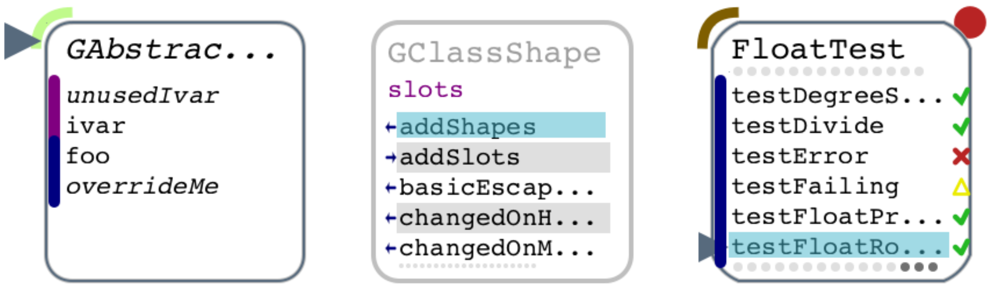


Figure 3.22. Gaucho Shapes present high level views of the objects

The leftmost shape represents an abstract class: both the labels for the name and a method in the list are displayed in italic to convey that the class and the method are abstract. The second shape depicts a class shape focused on revealing the use of the instance variable named slots. Developers can understand which methods assign or reference a particular variable, by simply instructing the class shape to reveal that information. The third depicts a test case shape, which enables a developer to understand the status of the test case (passed, failed or error) and of all its included test methods.

3.5.1 On the Textual Representation of Methods Statements

Gaucho is a visual programming environment (VPE) which makes use of spatial relationships, data abstractions, and a high level language of interaction for performing programming tasks, such as renaming a class, or adding a method to a class. *Gaucho* is in the middle ground between visual programming (VPLs) and pure textual languages [61].

In this dissertation we focused on investigating an alternate interface for the construction of OOP environments, nevertheless we decided to retain the use of text to specify behavior because of the complexities of devising a VPL for fully replacing textual descriptions of code [62; 63]. Nonetheless, we believe that *Gaucho* could easily be modified to incorporate other means of specifying behavior, including a shape that manipulates methods via a VPL instead of editing textual descriptions of statements. For instance, the VPL could use an iconographic language to denote operations, and specify the method body using a data-flow language such as Coherence [64], which relies on the abstraction of a virtual tree, similar to the abstract syntax tree of a parsed method.

Chapter 4

Evaluation

In the previous chapter of the thesis, we introduced Gaucho, which is our approach to the design of a development environment for OOP, and described its application to several modeling and programming activities.

Gaucho is an **object-focused environment** that allows developers to write programs by creating and manipulating lightweight and intuitive depictions of object-oriented constructs.

On the other hand, traditional 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.

The research question we investigate here is how does an object focused environment such as Gaucho compare 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 acknowledge that software is created by humans, for humans, thus software engineering is—above all—a human activity. However, the intrinsically non-deterministic nature of humans introduces a number of threats to the validity of controlled experiments, performed by researchers to evaluate software tools. One of them concerns how to record information without influencing the behavior of the subjects involved. Another one relates to providing means to assure the correctness of the gathered data, for further analyses and replication.

Furthermore, instrumenting an evaluation of a software development environment, such as Gaucho, places a heavy burden on the experimenters, due to the short duration of the actions programmers perform to fulfill the tasks composing the experiment. The brevity of the tasks makes it hard to track the time and effort of the subject under observation, which hampers with the correctness of the ulterior observations regarding the experiment.

To alleviate this burden, we focused on improving the tool support that enables researchers to perform controlled experiments with human subjects to evaluate the performance and usability of novel approaches and software engineering tools. We devised *Biscuit*, a toolset to specify tasks to be undertaken by the subjects of the experiment, and precisely time the subjects and record their complete behavior as they perform the tasks.

Structure of the chapter. In Section 4.1 we discuss often overlooked issues that come up during controlled experiments with human subjects, and we present *Buiscuit*, our toolset to alleviate some of the issues. In Section 4.2 we discuss the context of the evaluation. We describe the experimental design in Section 4.3, and the operation and results in Section 4.4. We conclude the chapter by reflecting on the outcome of the evaluation in Section 4.5, and discussing the advantages of instrumenting controlled experiments with tools such as Biscuit in Section 4.6.

4.1 Instrumenting the Evaluation of Software Tools

If software engineering is above all a human activity, then taking the human aspect out of the loop would be definitely wrong, when it comes to evaluating software engineering approaches.

Consequently, these past years have seen a steady increase of evaluations based on controlled experiments performed with human subjects. For example, Cornelissen *et al.* evaluated a tool for visualizing execution traces [65], Stasko *et al.* contrasted two visualization tools for depicting hierarchies using different space-filing layouts [66], Marcus *et al.* analyzed the support for program comprehension tasks of their sv3D tool [67], and Wettel *et al.* assessed the validity of the city metaphor for visualizing software systems [68].

Controlled experiments are prone to a large number of pitfalls, due to the one and only uncontrollable element of any controlled experiment involving human subjects: humans. Humans have individual talents, skills, behaviors, and quirks. In any experiment humans behave in a non-deterministic way, which introduces various threats to the validity of any experiment of this kind.

The pitfalls in question regard an issue that might be seen as a corollary to Heisenberg's uncertainty principle: How is the information that one wants to record actually being recorded, and does the fact that one records information in an intrusive way influence or modify the behavior of subjects?. If so, how can this be minimized?.

4.1.1 The Crux of Human-centric Experiments

There are diverse possibilities to record information in a controlled experiment, where a usual scenario is that a set of subjects is divided into two groups, the control group and the experimental group. Subjects in the former group are given some baseline setting, while the subjects of the latter group are given the tool to be evaluated. The goal is then to assess whether the experimental group can perform a set of tasks better, faster, etc. as opposed to the control group.

A literature survey [65; 69; 67; 66] reveals commonly adopted practices to tackle the issues which must be confronted when performing controlled experiments to evaluate software tools:

1. *Give the subjects tasks to perform and/or questions to answer, and the possibility to provide answers/findings in some form.* The subjects know what is expected from them; this can either come as a set of questions or a set of tasks. After performing the tasks the subjects then provide some form of feedback about what they have done, i.e., they need to answer questions. The experimenter's role in this situation is then, to obtain data about the correctness (i.e., was a task fulfilled or not/only partially) and the time taken by the subjects to perform a task.

   In the large majority of cases, experimenters opt for questionnaires, (often in the form of multiple choice questions or free-form text questions) to the subjects who fill them out during the experiment. Questionnaires can either be on paper or also in electronic form (such as survey websites).

2. *Keep track of the time taken by each subject on each task.* How can one reliably record the time it took for a subject to perform a task? One possibility is to have the subject write down the time (as part of the questionnaire); this introduces the risk of subjects writing down wrong information.

Another possibility is for the experimenter to record the timing, which makes it hard, if not impossible, to perform an experiment with multiple subjects at the same time or a remote experiment—without removing measurement issues due to human fallibility.

3. *Record what the subjects do while they perform tasks to try to find answers.* To record what subjects do, the often adopted solutions are "think-aloud" protocols, and/or filming and screencasts and/or audio-recording. Think-aloud protocols involve experiment participants thinking aloud as they are performing a set of specified tasks. Users are asked to say whatever they are looking at, thinking, doing, and feeling, as they go about their task.

   This enables observers to see first-hand the process of task completion (rather than only its final product). Observers at such a test are asked to objectively take notes of everything that users say, without attempting to interpret their actions and words. Test sessions are often audio and video taped so that developers can go back and refer to what participants did, and how they reacted.

   The purpose of this method is to make explicit what is implicitly present in subjects who are able to perform a specific task. A screencast is a digital recording of computer screen output, also known as a video screen capture, often containing audio narration. Experimenters can use them to record the full interaction of the subjects with the tools they used.

4. *Process the previously recorded data to extract additional information, and allow the experimenters to gather the data easily.* An additional problem is that the data recorded using such approaches needs to be post-processed since it comes as a digital movie/audio recording.

   The post-processing, e.g., transcribing what happened, can be lengthy and imprecise—especially when the number of subjects is high—due to the lack of formalism in natural language and human behavior. Raising the level of abstraction of the data would allow for easier and more automated processing.

   Related to this issue, gathering the data in itself can be a challenge—either because of manual processing of hand-written questionnaires, or because it comes from multiple sources (e.g. questionnaires, timing data, and other recorded data).

We do not address other important issues pertinent to controlled experiment involving humans, e.g., the choice of participants, controlled variables, tasks, etc. These and others are all important questions which go beyond the scope of the present discussion. We focus on a seemingly smaller set of issues, which however can and does contribute to a loss of precision or even a falsification of the recorded data.

Summing up. Due to issues related to timing, data processing, and mostly inaccurate tracking of what happens during the experiment, the risk is that the data that is being collected during an experiment is distorted or even wrong. To mitigate such risks we devised *Biscuit*, an tool infrastructure for non-intrusive and precise tracking of data related to experiments with human subjects.

4.1.2 Biscuit: Tracking Human-Centric Controlled Experiments

Biscuit supports performing controlled experiments, by recording relevant pieces of information regarding an experiment performed with human subjects. At the moment, it is geared towards experiments evaluating development tools.

Next, we detail how experimenters using Biscuit address the issues related to conducting evaluations with human subjects, we stated in the previous section:

1. *Give the subjects tasks to perform and/or questions to answer, and the possibility to provide answers/findings in some form.* To address the first issue, Biscuit can set up an experiment made up of tasks which in turn, consist in a description and goals that need to be accomplished by the subjects.

   The format of answers to the goals range from multiple choice questions and free form text entries, to modifying the underlying system entities—such as adding/removing classes/methods until automated tests pass.

   Using *Biscuit*, the experimenter can automatically generate a user interface for any experiment; it presents the experiment to the subjects and guides them through the tasks until completion.

2. *Keep track of the time taken by each subject on each task.* An automatically generated user interface for each experiment run, tracks the answers and durations of each task while the user is performing them. This addresses the first two issues; namely giving the subjects tasks to perform or questions to answer; and registering the answers and completion times.

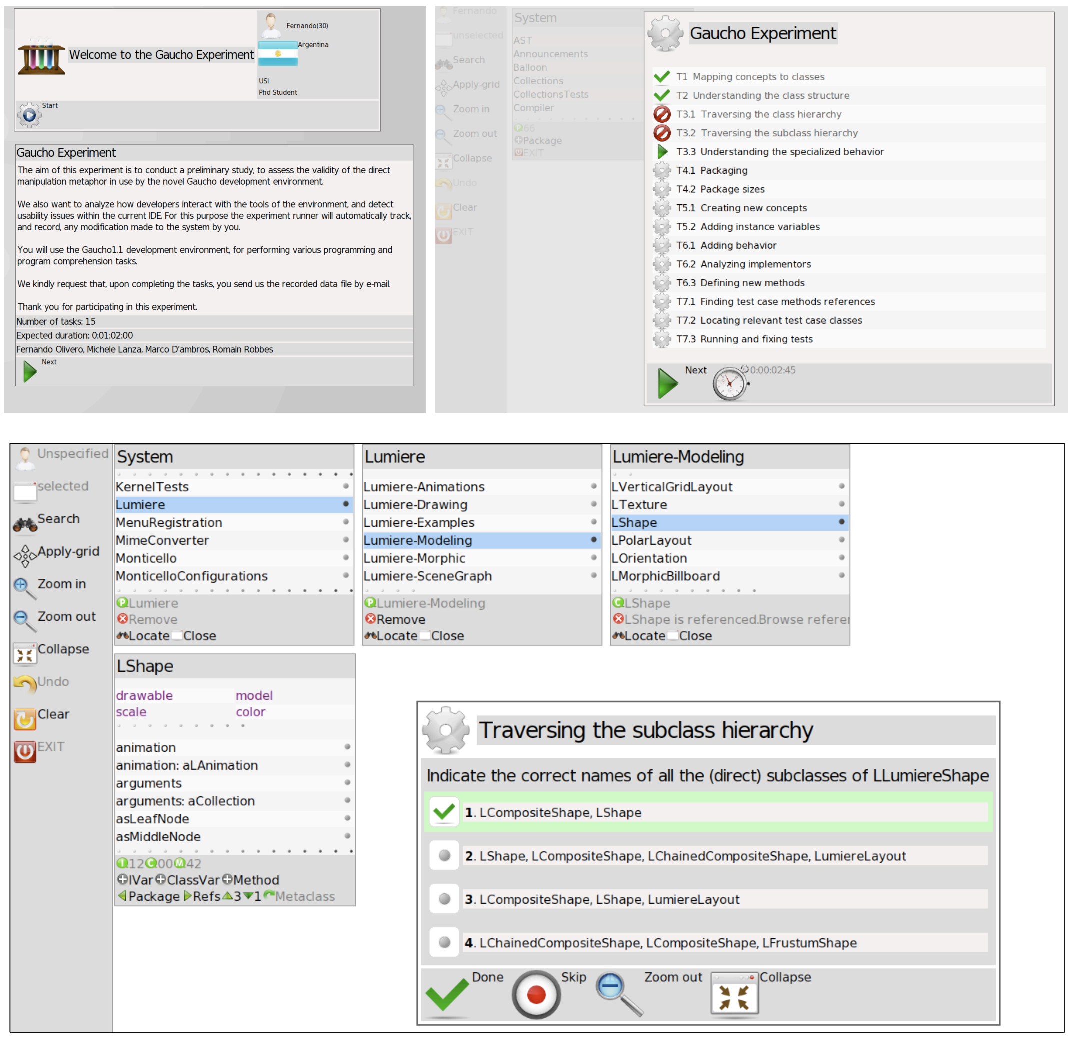
3. *Record what the subjects do while they perform tasks to try to find answers*. To address the third issue prior to running each task, *Biscuit* installs a spy that records *every user interaction*: from simple mouse move events, to more complex interactions such as performing changes to the code base, or providing answers to the experiment goals.

   The spy is built on top of a system monitoring tool called Spyware [70], enhanced with a complete instrumentation of the Event-Command pattern. The recording enables one to keep track of every user interaction in the form of recorded events that trigger commands, thus eliminating a great level of uncertainty from the correctness of the subjects answers, due to the *faithful replicability* of each experiment run.

   The events recorded by Biscuit correspond to actual user actions (source code navigation and modifications), in contrast to video recordings that require significant further interpretation, thus addressing this fourth issue: the event trace is open to automated analyses and can be replicated.

4. *Process the previously recorded data to extract additional information, and allow the experimenters to gather the data easily*. To address the fourth and final issue, upon completing an experiment the subject is asked to send (via email) an automatically created zipped file containing all the recorded data to the experimenters. This file contains every user interaction recorded by the spy during the experiment run, together with the answers and solutions to each task, and additional meta-data such as the participant name or identifier, and the total completion time. The process is straightforward both from the subject's and the experimenter's point of view.

Figure 4.1 depicts the experiment description that is shown once the subject has completed the pre-test questionnaire and entered basic data, and an actual experimental run. The screenshot illustrates how the Biscuit task runner records and informs users of the passage of time, presents the tasks, and collects the answers from the subjects. On the bottom corner, we see the task widget with the answer form; the rest of the screen is occupied by the tool being evaluated, in this case Gaucho.



Gaucho Experiment

The aim of this experiment is to conduct a preliminary study, to assess the validity of the direct manipulation metaphor in use by the novel Gaucho development environment.

We also want to analyze how developers interact with the tools of the environment, and detect usability issues within the current IDE. For this purpose the experiment runner will automatically track, and record, any modification made to the system by you.

You will use the Gaucho1.1 development environment, for performing various programming and program comprehension tasks.

We kindly request that, upon completing the tasks, you send us the recorded data file by e-mail.

Thank you for participating in this experiment.

Number of tasks: 15

Expected duration: 0:01:02:00

Fernando Olivero, Michele Lanza, Marco D'ambros, Romain Robbes

Gaucho Experiment

- ✓ T1 Mapping concepts to classes
- ✓ T2 Understanding the class structure
- T3.1 Traversing the class hierarchy
- T3.2 Traversing the subclass hierarchy
- T3.3 Understanding the specialized behavior
- T4.1 Packaging
- T4.2 Package sizes
- T5.1 Creating new concepts
- T5.2 Adding instance variables
- T6.1 Adding behavior
- T6.2 Analyzing implementors
- T6.3 Defining new methods
- T7.1 Finding test case methods references
- T7.2 Locating relevant test case classes
- T7.3 Running and fixing tests

Traversing the subclass hierarchy

Indicate the correct names of all the (direct) subclasses of LLumiereShape

✓ **1**. LCompositeShape, LShape

2. LShape, LCompositeShape, LChainedCompositeShape, LumiereLayout

3. LCompositeShape, LShape, LumiereLayout

4. LChainedCompositeShape, LCompositeShape, LFrustumShape

Figure 4.1. An experiment run instrumented with Biscuit

Related Work

To our knowledge, the only other toolset dedicated to recording fine-grained activity in controlled experiments is Emperior [71], an IDE for the Java and Groovy programming languages that records programmer navigation at the file level (Biscuit records it at the method level), records compilations of the programs, unit test runs, and periodically takes snapshots of the source code (Biscuit tracks the changes themselves); the snapshots generate an extremely large amount of redundant data. A new instance of Emperior needs to be launched for each task.

There are other tools that record usage data, such as HackyStat [72]. It collects navigation data, metrics data, and test runs. HackyStat has not been used to record controlled experiments. Similarly, Mylyn [73] records navigation and edit information as part of its activity; this data was used to evaluate Mylyn in a field study, not an experiment. Compared to these tools, Biscuit records more kinds of data (such as actual changes, not edit activity), and allows the definition of actual experimental tasks.

4.2 Context

Nowadays object-oriented developers perform programming tasks aided by integrated development environments (IDEs), which feature numerous tools that provide the means to construct programs. Nevertheless, IDEs present difficulties, for example 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"* [74].

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. We claim that an environment based on a different user interface can intrinsically improve the support for program comprehension, and minimize the need to recur to external tools for understanding the programs.

The two contrasting interfaces here are the *view-focused* and the *object-focused* environments. 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 [46], foster the notion that developers are in direct contact with the objects themselves, instead of abstract entities subordinated to the tools.

We argue that an object focused environment such as 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, favoring 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.

4.3 Experimental Design

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.

4.3.1 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?

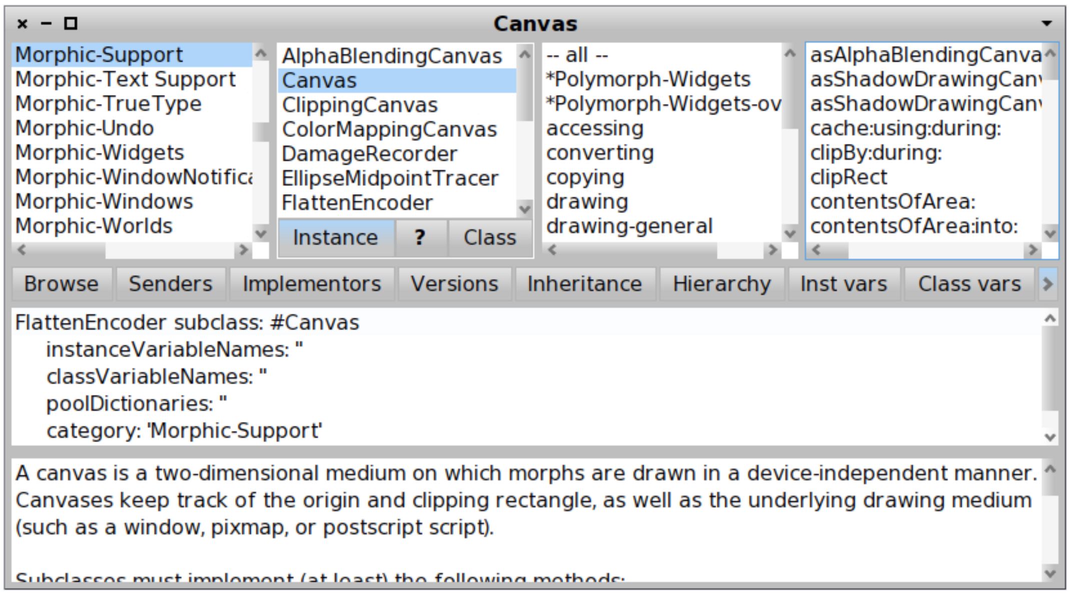
RQ2. Does the use of Gaucho increase the *correctness* of the answers to the program comprehension tasks, compared to a traditional IDE?

4.3.2 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.

4.3.3 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 [43]. 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, see Figure 4.2.



```
FlattenEncoder subclass: #Canvas
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Morphic-Support'
```

A canvas is a two-dimensional medium on which morphs are drawn in a device-independent manner.
Canvases keep track of the origin and clipping rectangle, as well as the underlying drawing medium
(such as a window, pixmap, or postscript script).

Subclasses must implement (at least) the following methods:

Figure 4.2. A System Browser of the Pharo IDE

4.3.4 Object system & Treatments

We chose *Lumière* [75] 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 treatments in Table 4.1.

Table 4.1. Treatments presented to the subjects

Group	**Tool**	**Description**
Experimental	Gaucho	Gaucho application with a loaded model of Lumiere
Control	Pharo	Pharo IDE with default development tools and a loaded model of Lumiere

4.3.5 Tasks

We designed 3 programming and 11 program comprehension tasks based on the set of questions composed by Sillito *et al.* [10]. 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. In Table 4.2 we list the tasks, and in Table 4.3 the subset of the questions compiled by Sillito *et al.* we used to design the tasks of the experiment.

Table 4.2. The fourteen 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

Table 4.3. Referenced questions from Silito's framework

Id	Description
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?

4.4 Experimental Operation and Results

4.4.1 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, using the automated experiment runner toolset, called *Biscuit*, described in Section 4.1.2.

4.4.2 Pre-Experiment Questionnaire

Using Biscuit, we designed and presented a questionnaire that served to allow capturing the personal information that we used for analyzing the results of the experiment run (See Figure 4.3).

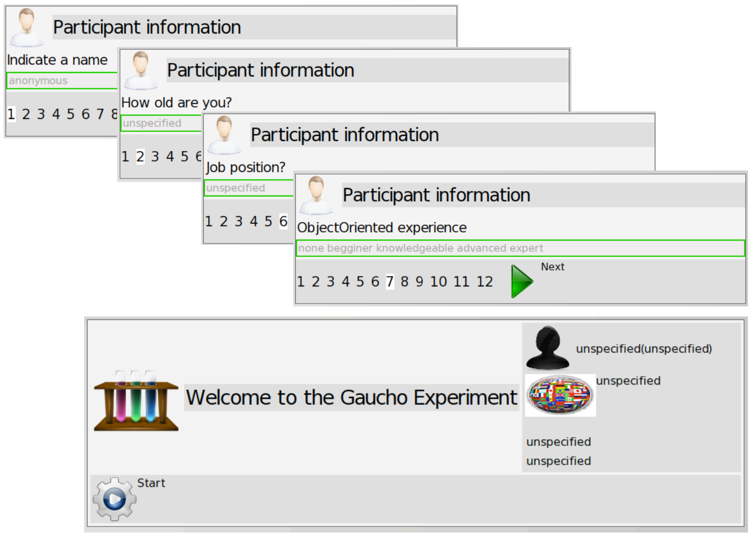

Participant information
Indicate a name
anonymous
Participant information
How old are you?
unspecified
Participant information
Job position?
unspecified
Participant information
ObjectOriented experience
none begginer knowledgeable advanced expert
Next
Welcome to the Gaucho Experiment
unspecified(unspecified)
unspecified
unspecified
unspecified
Start

Figure 4.3. The questions presented by Biscuit at the start

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 4.4.

Table 4.4. The declared expertise of the subjects

	Control Group			Experimental Group	
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

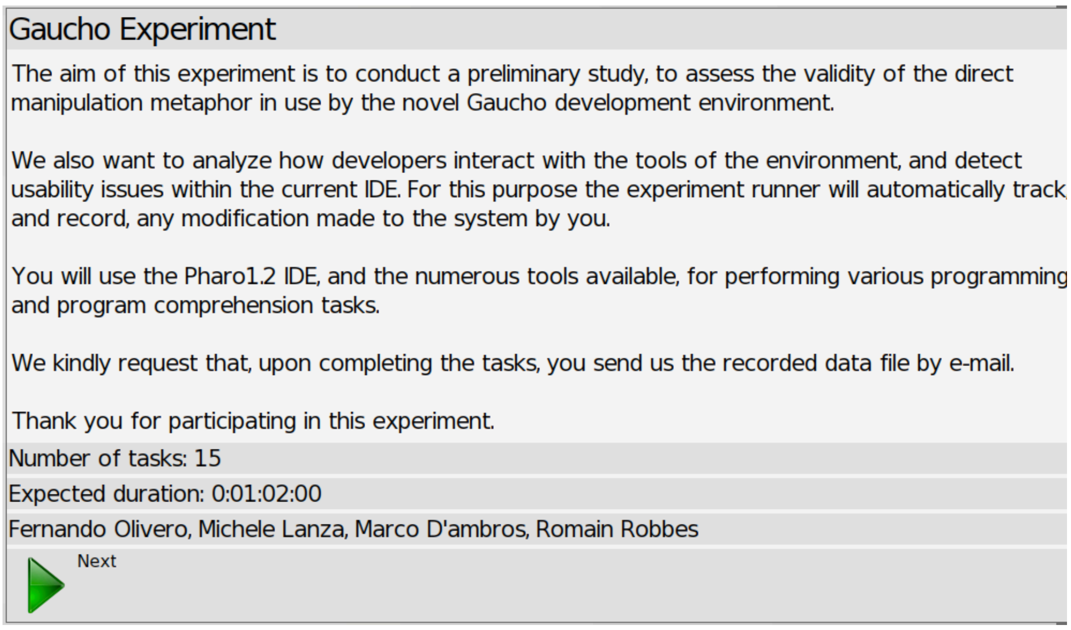
4.4.3 Experiment Questionnaire

In this section we present the the content of the questionnaires forming the experiment, separated into tasks.

Introduction

The aim of this experiment is to conduct a preliminary study, to assess the validity of the direct manipulation metaphor in use by the novel Gaucho development environment.

We also want to analyze how developers interact with the tools of the environment, and detect usability issues within the current IDE. For this purpose the experiment runner will automatically track and record any modification made to the system. You will use the <toolset>[1], and the numerous tools available, for performing various programming and program comprehension tasks.
Thank you for participating in this experiment.
Fernando Olivero, Michele Lanza, Marco D'ambros, and Romain Robbes


Gaucho Experiment

The aim of this experiment is to conduct a preliminary study, to assess the validity of the direct manipulation metaphor in use by the novel Gaucho development environment.

We also want to analyze how developers interact with the tools of the environment, and detect usability issues within the current IDE. For this purpose the experiment runner will automatically track and record, any modification made to the system by you.

You will use the Pharo1.2 IDE, and the numerous tools available, for performing various programming and program comprehension tasks.

We kindly request that, upon completing the tasks, you send us the recorded data file by e-mail.

Thank you for participating in this experiment.

Number of tasks: 15

Expected duration: 0:01:02:00

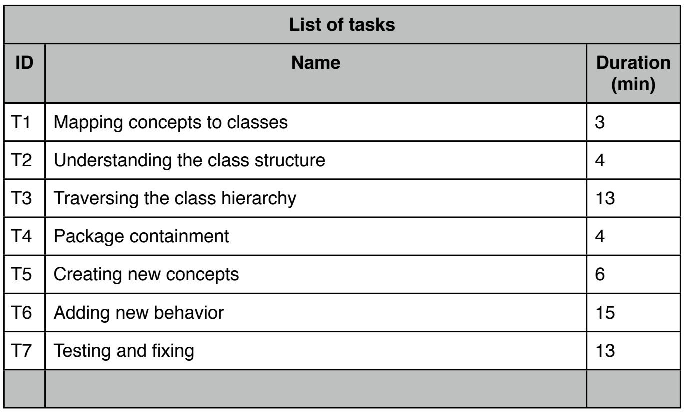
Fernando Olivero, Michele Lanza, Marco D'ambros, Romain Robbes

Figure 4.4. The Introduction presented by Biscuit for the treatment using Pharo

[1]The Pharo IDE for treatment 1, and Gaucho for treatment 2

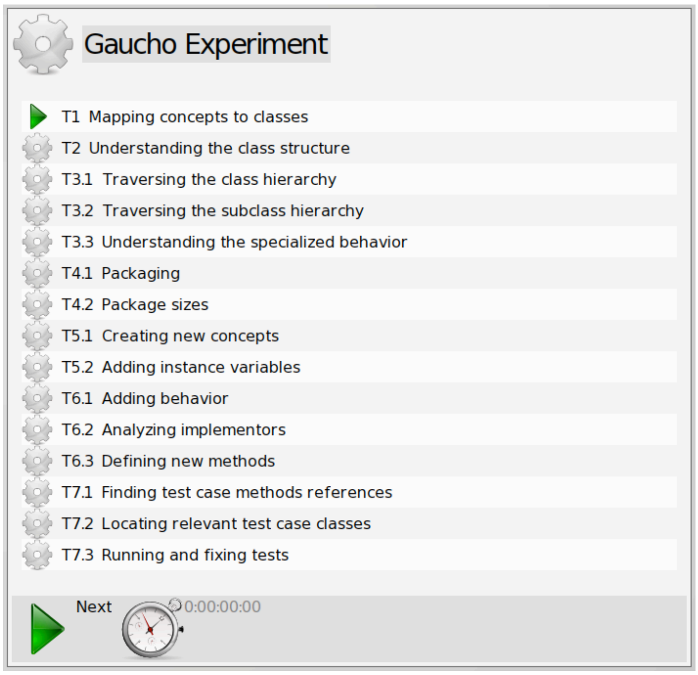
Tasks

In this section we illustrate both the initial design of each task, and its final instrumentation in Biscuit. Figure 4.5 depicts the overall design of the tasks, and Figure 4.6 illustrates each task in more detail.



List of tasks		
ID	**Name**	**Duration (min)**
T1	Mapping concepts to classes	3
T2	Understanding the class structure	4
T3	Traversing the class hierarchy	13
T4	Package containment	4
T5	Creating new concepts	6
T6	Adding new behavior	15
T7	Testing and fixing	13

(a) The design



Gaucho Experiment

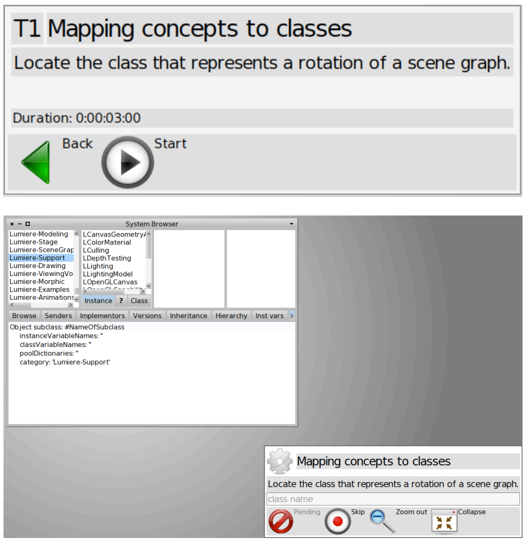
T1 Mapping concepts to classes
T2 Understanding the class structure
T3.1 Traversing the class hierarchy
T3.2 Traversing the subclass hierarchy
T3.3 Understanding the specialized behavior
T4.1 Packaging
T4.2 Package sizes
T5.1 Creating new concepts
T5.2 Adding instance variables
T6.1 Adding behavior
T6.2 Analyzing implementors
T6.3 Defining new methods
T7.1 Finding test case methods references
T7.2 Locating relevant test case classes
T7.3 Running and fixing tests

Next 0:00:00:00

(b) The instrumentation in Biscuit

Figure 4.5. The outline of the tasks of the experiment

T1	Mapping concepts to classes
Goal	Locate the class that represents a rotation of a scene graph.
Rationale	Programming tasks usually begin with some form of concept location, which relates to the first category of questions which is "finding initial focus points". In this task we want to demonstrate that Gaucho provides a useful global search widget, somehow similar to the one found in Pharo.
Questions	Q1: Which type represents this domain concept or this UI element or action ?
Solution	LRotationNode
Implementation	•Paper: Use the lexical or static analysis based search. •Gaucho: Use the global search widget. •Pharo:Use the find class tool opened from the contextual menu of the system category pane, or use the mercury toolbar widget
Duration	3 min
Pre-reqs	A clean pampas.
Constraints	Use the tools, don't look for the answer by sending messages to the class. Why?: To leverage the subject programming skills.

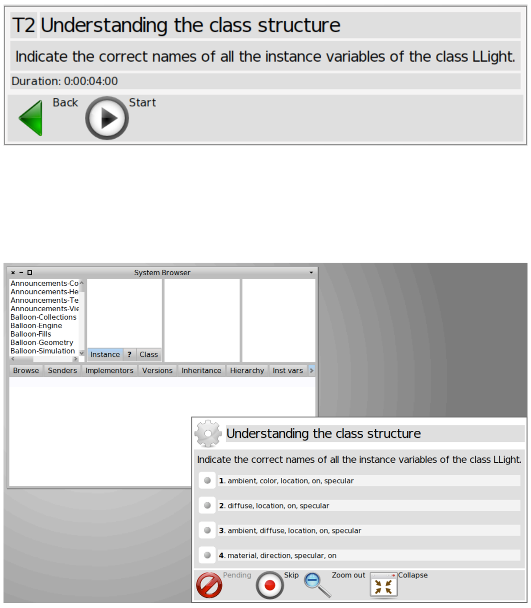

T1 Mapping concepts to classes
Locate the class that represents a rotation of a scene graph.
Duration: 0:00:03:00

```
Object subclass: #NameOfSubclass
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Lumiere-Support'
```

Mapping concepts to classes
Locate the class that represents a rotation of a scene graph.
class name

(a) T1: Mapping concepts to classes

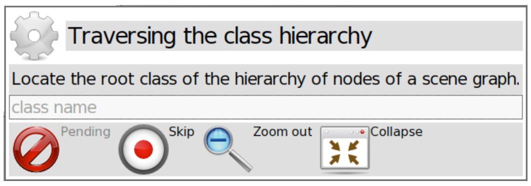
T2	Understanding the class structure
Goal	Indicate the correct names of all the instance variables of the class LLight. *1.ambient color location on specular* *2.diffuse location on specular* *3.ambient diffuse location on specular* *4.material direction specular on*
Rationale	Developers often have the need to understand the structure of classes: With this group of tasks (T2) we want to analyze if this structure is better understood visualized by abstractions than a class definition string, and the methods list of the browsers. Class Definition strings are just strings, instance variable names don't necessarily have an order (they can be typed by the user in any order). Therefore the subject might choose incorrectly, because in the original definition the names aren't ordered. With abstractions this problem is not encountered, because the shape always displays the names in order. This evidences that class definition are just treated as strings.
Questions	Q6: What are the parts of this type ?
Solution	*2.ambient diffuse location on specular*
Implementation	•Tools: Structural overviews tools •Gaucho: Look at the attributes group of the shape. •Pharo: Read the class definition string , and detect the instance variable names.
Duration	4 min
Pre-reqs	Visual Element depicting the class LLight
Constraints	Use the tools, don't look for the answer by sending messages to the class. Why?: To leverage the subject programming skills.


T2 Understanding the class structure
Indicate the correct names of all the instance variables of the class LLight.
Duration: 0:00:04:00

Understanding the class structure
Indicate the correct names of all the instance variables of the class LLight.
1. ambient, color, location, on, specular
2. diffuse, location, on, specular
3. ambient, diffuse, location, on, specular
4. material, direction, specular, on

(b) T2: Understanding the class structure

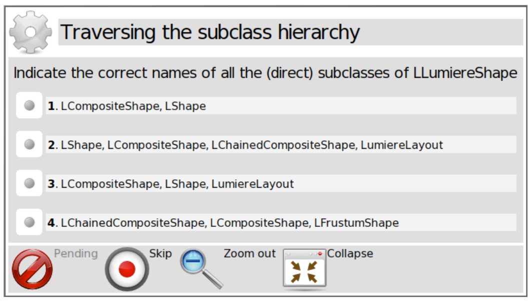
T3.1	Traversing the class hierarchy
Goal	Locate the root class of the hierarchy of nodes of a scene graph.
Rationale	Locating the root of a class hierarchy is usually needed when the task requires extending the hierarchy with another concept, to understand how this new concept can be plugged in. With this task we want to asses if Gaucho allows navigating a class hierarchy.
Questions	Q8 Were does this type fit in the type hierarchy ?
Solution	LNode
Implementation	•Tools: Structural analysis techniques. •Gaucho: Use the superclass navigation icon located in the toolbar of the LRotationNode shape, to open the superclass group. •Pharo: Open a hierarchy browser on the class LRotationNode.
Duration	3 min
Pre-reqs	Visual Element depicting the class LRotationNode
Constraints	Use the tools, don't look for the answer by sending messages to the class. Why?: To leverage the subject programming skills.


Traversing the class hierarchy
Locate the root class of the hierarchy of nodes of a scene graph.
class name

(c) T3.1: Traversing the class hierarchy

Figure 4.6. The tasks of the experiment

T3.2	Traversing the subclass hierarchy
Goal	Indicate the correct names of all the (direct) subclasses of LLumiereShape? *1.LChainedCompositeShape LCompositeShape LShape LumiereLayout* *2.LChainedCompositeShape LCompositeShape LFrustumShape* *3.LCompositeShape LShape LumiereLayout* *4.LCompositeShape LShape*
Rationale	Understand the composition of a class hierarchy
Questions	Q9 Does this type have any siblings in the type hierarchy ?
Solution	*1.LChainedCompositeShape LCompositeShape LShape LumiereLayout*
Implementation	•Tools: Structural analysis techniques •Gaucho:The subclasses navigation icon located in the toolbar of the LNode shape, indicates the number of direct siblings and opens the subclasses group. •Pharo: Open a hierarchy browser on the class LGeometry , and analyze the subclasses.
Duration	4 min
Pre-reqs	Visual Element depicting the class LLumiereShape
Constraints	Use the tools, don't look for the answer by sending messages to the class. Why?: To leverage the subject programming skills.


Traversing the subclass hierarchy

Indicate the correct names of all the (direct) subclasses of LLumiereShape

1. LCompositeShape, LShape

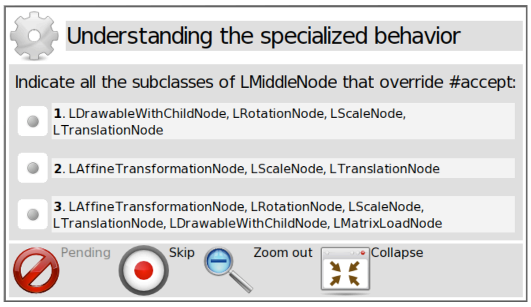
2. LShape, LCompositeShape, LChainedCompositeShape, LumiereLayout

3. LCompositeShape, LShape, LumiereLayout

4. LChainedCompositeShape, LCompositeShape, LFrustumShape

(d) T3.2: Traversing the subclass hierarchy

T3.3	Understanding the specialized behavior
Goal	Indicate all the subclasses in the hierarchy of middle nodes that override #accept: *1.LAffineTransformationNode, LRotationNode,LScaleNode, LTranslationNode, LDrawableWithChildNode.* *2.LAffineTransformationNode, LScaleNode, LTranslationNode.* *3.LAffineTransformationNode, LRotationNode,LScaleNode, LTranslationNode, LDrawableWithChildNode, LMatrixLoadNode.*
Rationale	•Force the subject to look at multiple classes, methods and relationships at once, to evidence the supposed benefit of the Gaucho interface against the more rigid tool based interfaces. •Asses if reifying the subclasses group is useful in this kind of programming tasks.
Questions	Q9 Does this type have any siblings in the type hierarchy ?
Solution	*1.LAffineTransformationNode, LRotationNode,LScaleNode, LTranslationNode, LDrawableWithChildNode.*
Implementation	•Tools: Structural analysis techniques, Structural overviews tools •Gaucho:Use the subclasses navigation icon located in the toolbar of the LNode shape, to open the subclasses group. Spawn the group, and analyze the subclasses methods. •Pharo: Open a hierarchy browser on the class LNode or LMiddleNode. Navigate through the methods list, remember those that satisfy the condition, or open several browsers
Duration	6 min
Pre-reqs	Visual Element depicting the class LNode
Constraints	


Understanding the specialized behavior

Indicate all the subclasses of LMiddleNode that override #accept:

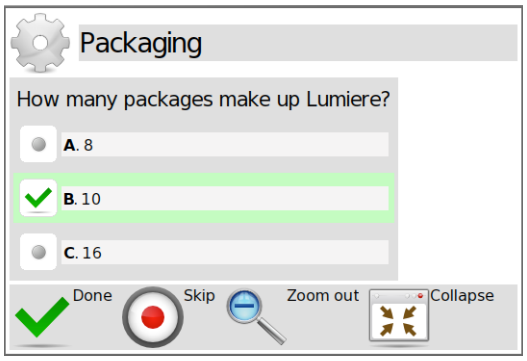
1. LDrawableWithChildNode, LRotationNode, LScaleNode, LTranslationNode

2. LAffineTransformationNode, LScaleNode, LTranslationNode

3. LAffineTransformationNode, LRotationNode, LScaleNode, LTranslationNode, LDrawableWithChildNode, LMatrixLoadNode

(e) T3.3: Understanding the specialized behavior

T4.1	Packaging
Goal	How many packages make up Lumiere? A.8 B.10 C.16
Rationale	Asses the usability of the Gaucho system shape, and the package shape icons .
Questions	Q7 Which types is type a part of ?
Solution	B.10
Implementation	
Duration	3min
Pre-reqs	Visual element depicting the System
Constraints	


Packaging

How many packages make up Lumiere?

A. 8

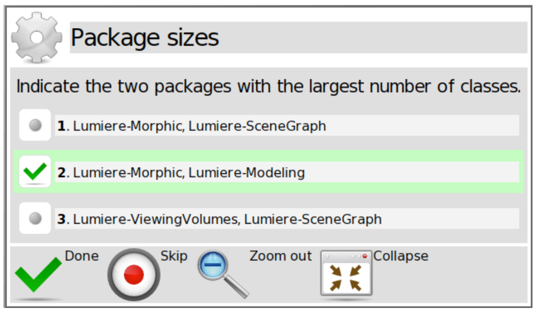
✓ B. 10

C. 16

(f) T4.1: Packaging

Figure 4.6. The tasks of the experiment (cont)

T4.2	Package sizes
Goal	Indicate the packages with the biggest amount of classes. 1.'Lumiere-Morphic' 'Lumiere-SceneGraph' 2.'Lumiere-Morphic' 'Lumiere-Modeling' 3.'Lumiere-ViewingVolumes' 'Lumiere-SceneGraph'
Rationale	Asses the usability of the Gaucho system shape, and the package shape icons .
Questions	Q7 Which types is type a part of ?
Solution	1.Lumiere-Morphic' 'Lumiere-SceneGraph'
Implementation	
Duration	4min
Pre-reqs	Visual element depicting the System
Constraints	

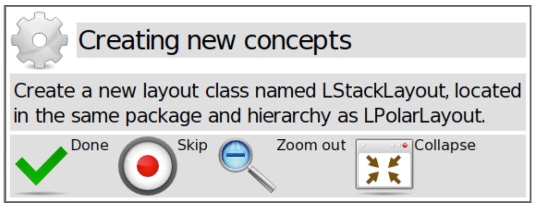

Package sizes

Indicate the two packages with the largest number of classes.

1. Lumiere-Morphic, Lumiere-SceneGraph
2. ✓ Lumiere-Morphic, Lumiere-Modeling
3. Lumiere-ViewingVolumes, Lumiere-SceneGraph

(g) T4.2: Package sizes

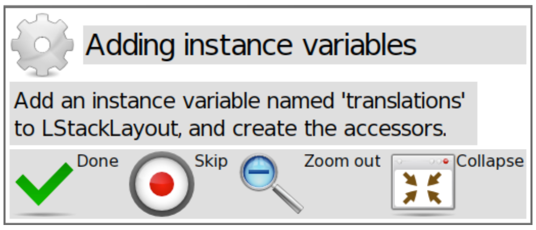
T5.1	Creating new concepts
Goal	Create a new layout class named LStackLayout in the same package and hierarchy as LPolarLayout.
Rationale	Asses the effectiveness of the Gaucho navigation toolbar to access the package from a class, and create a class within that package.
Questions	Q7 Which types is type a part of ? Q8 Were does this type fit in the type hierarchy ?
Solution	LStackLayout class creation, subclass of LumiereLayout, and packaged in Lumiere-Modeling.
Implementation	•Tools: Structural overviews tools •Gaucho:Use the containment navigation icon located in the toolbar of the LPolarLayout shape, to open the package. In the package shape create a new class named LStackLayout, using the add class entry field. •Pharo: In the browser select the package Lumiere-Modeling, and reuse an LumiereLayout subclass definition string, by replacing the name of the subclass string with LStackLayout.
Duration	4 min
Pre-reqs	Visual element depicting the class LPolarLayout.
Constraints	


Creating new concepts

Create a new layout class named LStackLayout, located in the same package and hierarchy as LPolarLayout.

(h) T5.1: Creating new concepts

T5.2	Adding instance variables
Goal	Add an instance variable named 'translations' to LStackLayout, and create the accessors.
Rationale	•Asses the ease of use of the add instance variable entry field •Compare the ease of use of large contextual menus against the icons of the class shapes.
Questions	
Solution	LStackLayout.translations LStackLayout>>translations: LStackLayout>>translations
Implementation	•Gaucho: Use the add instance variable entry field, and press the create accessors icon. •Pharo: Edit the class definition string to add the instance variable, and use the class contextual menu and click on the create accessors item.
Duration	3 min
Pre-reqs	Visual element depicting the class LStackLayout.
Constraints	

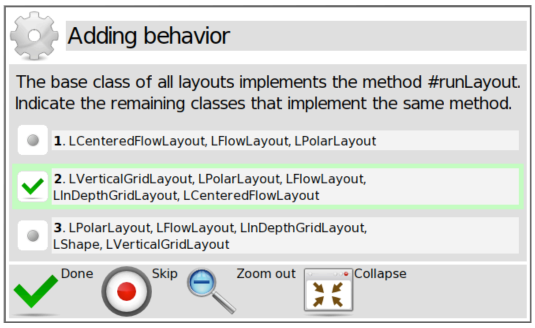

Adding instance variables

Add an instance variable named 'translations' to LStackLayout, and create the accessors.

(i) T5.2: Adding instance variables

Figure 4.6. The tasks of the experiment (cont)

T6.1	Adding behavior
Goal	The base class of all layouts implements the method #runLayout. Indicate the remaining classes that implement the same method. *1.LCenteredFlowLayout, LFlowLayout, LPolarLayout.* *2.LPolarLayout, LCenteredFlowLayout, LFlowLayout, LInDepthGridLayout, LVerticalGridLayout.* *3.LPolarLayout, LFlowLayout, LInDepthGridLayout, LShape, LVerticalGridLayout.*
Rationale	•Search for abstract methods in the base class, and locate existing implementors of these methods. Usually when adding new behavior that is an abstract method in the base class, developers start by looking at relevant implementors of the same method. •Asses the comprehension of the navigation icons of methods items in the class shape, to provide quick access to an implementors group.
Questions	Q6 What are the parts of this type ? Q11 Who implements this interface or these abstract methods ?
Solution	*2.LPolarLayout, LCenteredFlowLayout, LFlowLayout, LInDepthGridLayout, LVerticalGridLayout.*
Implementation	•Tools: Structural overviews tools, Static structural analysis techniques. •Gaucho: Scroll the methods list of the LumiereLayout shape to find the abstract method #runLayout, and spawn the implementors group. •Pharo: Similar, but scrolling the browser and spawning an implementors list.
Duration	4 min
Pre-reqs	Visual element depicting the class LumiereLayout.
Constraints	

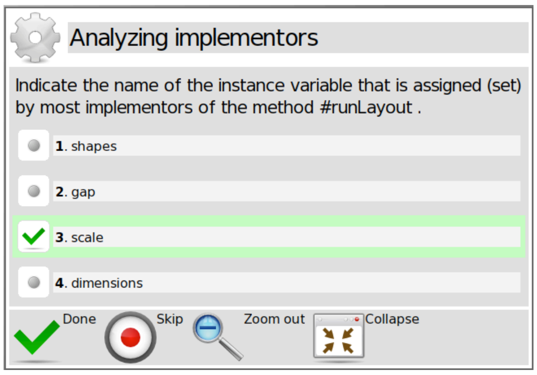

Adding behavior

The base class of all layouts implements the method #runLayout. Indicate the remaining classes that implement the same method.

1. LCenteredFlowLayout, LFlowLayout, LPolarLayout
[✓ 2. LVerticalGridLayout, LPolarLayout, LFlowLayout, LInDepthGridLayout, LCenteredFlowLayout]
3. LPolarLayout, LFlowLayout, LInDepthGridLayout, LShape, LVerticalGridLayout

(j) T6.1: Adding behavior

T6.2	Analyzing implementors
Goal	Indicate the name of the instance variable that is assigned (set) by most implementors of the method #runLayout. 1.shapes 2.gap 3.scale 4.dimensions
Rationale	Force the subject to compare multiple classes and methods at once.
Questions	Q15 Where is this variable or data structure being accessed ?
Solution	4.dimensions
Implementation	•Tools: Structural overviews tools, Static structural analysis techniques •Gaucho:Spawn the implementors group, and compare the methods to find the instance variable. •Pharo: Scroll up and down the list of implementors, reading and comparing the code. Open and manually rearrange several browsers to ease the comparison task.
Duration	8 min
Pre-reqs	•Visual element depicting the implementors #runLayout. •Visual element depicting the base class, LumiereLayout.
Constraints	


Analyzing implementors

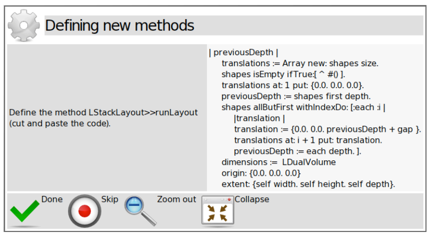
Indicate the name of the instance variable that is assigned (set) by most implementors of the method #runLayout .

1. shapes
2. gap
[✓ 3. scale]
4. dimensions

(k) T6.2: Analyzing implementors

Figure 4.6. The tasks of the experiment (cont)

T6.3	Defining new methods
Goal	Define the method LStackLayout>>runLayout Cut and paste the following code: I previousDepth I self resetResult. shapes isEmpty ifTrue:[^ #()]. translations at: 1 put: {0.0. 0.0. 0.0}. previousDepth := shapes first depth. shapes allButFirst withIndexDo: [:each :i I Itranslation I translation := {0.0. 0.0. previousDepth + gap }. translations at: i + 1 put: translation. previousDepth := each depth.]. dimensions := LDualVolume origin: {0.0. 0.0. 0.0} extent: {self width. self height. self depth}
Rationale	Asses the usability of method creation using Gaucho .The code is given because it is not our goal to improve method edition.
Questions	
Solution	LStackLayout>>runLayout I previousDepth I self resetResult. shapes isEmpty ifTrue:[^ #()]. translations at: 1 put: {0.0. 0.0. 0.0}. previousDepth := shapes first depth. shapes allButFirst withIndexDo: [:each :i I Itranslation I translation := {0.0. 0.0. previousDepth + gap }. translations at: i + 1 put: translation. previousDepth := each depth.]. dimensions := LDualVolume origin: {0.0. 0.0. 0.0} extent: {self width. self height. self depth}.
Implementation	•Gaucho: Click on the add method entry field, and type the #runLayout. In the automatically created method shape, cut and paste the code and accept •Pharo: Click on the methods pane of the browser and cut and past the code and accept.
Duration	3 min
Pre-reqs	Visual element depicting the class LStackLayout.
Constraints	

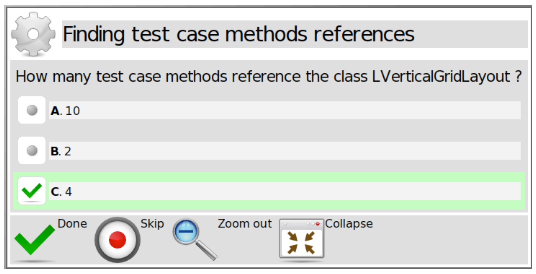

Defining new methods

Define the method LStackLayout>>runLayout (cut and paste the code).

```
| previousDepth |
    translations := Array new: shapes size.
    shapes isEmpty ifTrue:[ ^ #() ].
    translations at: 1 put: {0.0. 0.0. 0.0}.
    previousDepth := shapes first depth.
    shapes allButFirst withIndexDo: [:each :i |
        |translation |
        translation := {0.0. 0.0. previousDepth + gap }.
        translations at: i + 1 put: translation.
        previousDepth := each depth. ].
    dimensions := LDualVolume
    origin: {0.0. 0.0. 0.0}
    extent: {self width. self height. self depth}.
```

(l) T6.3: Defining new methods

T7.1	Testing
Goal	How many tests refer to an instance of the class LVerticalGridLayout ? A.10 B.2 C.4
Rationale	•Whenever a new behavior is going to be tested, usually we start by understanding examples of usage of similar classes, therefore we look for references to classes that implement the same behavior. •Asses the comprehension of the metric toolbar icons of the class shape. The subject can use the tests references icon, instead of opening the references group, which presents the number.
Questions	Q15 Where is this method called or type referenced ?
Solution	B.2
Implementation	•Tools: •Gaucho: The tests references icon, shows the number of tests references. •Pharo: Open an class references browser, and manually count the appearances of LVerticalGridLayout in the code pane of each item that is a test.
Duration	3 min
Pre-reqs	Visual element depicting the class LVerticalGridLayout.
Constraints	


Finding test case methods references

How many test case methods reference the class LVerticalGridLayout ?

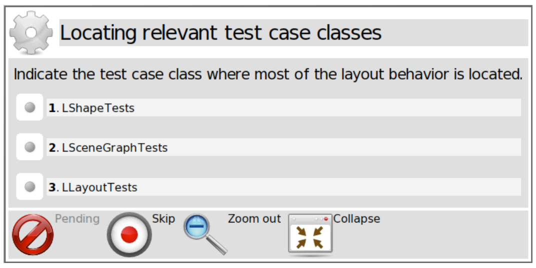
A. 10

B. 2

✓ C. 4

(m) T7.1: Finding test case methods references

T7.2	Finding test references
Goal	Indicate the test case class where most of the layout behavior is located. 1.LShapeTests 2.LSceneGraphTests 3.LLayoutTests
Rationale	Locating the correct test case that will contain the test method to implement, is a pre-requisite before actually implementing it.
Questions	Q6 What are the parts of this type ? Q12 Where is this method called or type referenced ?
Solution	1.LShapeTests
Implementation	
Duration	4 min
Pre-reqs	Visual element depicting the subclasses group of LumiereLayout
Constraints	


Locating relevant test case classes

Indicate the test case class where most of the layout behavior is located.

1. LShapeTests

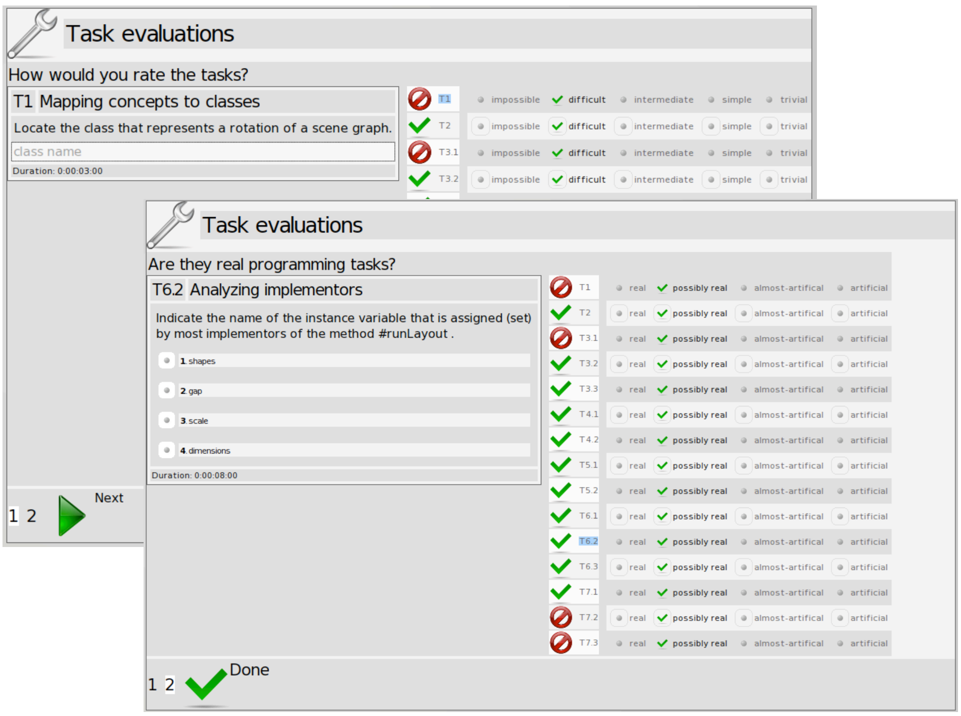
2. LSceneGraphTests

3. LLayoutTests

(n) T7.2: Locating relevant test case classes

Figure 4.6. The tasks of the experiment (cont.)

4.4.4 Debriefing Questionnaire



Task evaluations

How would you rate the tasks?

T1 Mapping concepts to classes

Locate the class that represents a rotation of a scene graph.

class name

Duration: 0:00:03:00

⊘	T1	impossible	[✓ difficult]	intermediate	simple	trivial
✓	T2	impossible	[✓ difficult]	intermediate	simple	trivial
⊘	T3.1	impossible	[✓ difficult]	intermediate	simple	trivial
✓	T3.2	impossible	[✓ difficult]	intermediate	simple	trivial

1 2 Next

Task evaluations

Are they real programming tasks?

T6.2 Analyzing implementors

Indicate the name of the instance variable that is assigned (set) by most implementors of the method #runLayout .

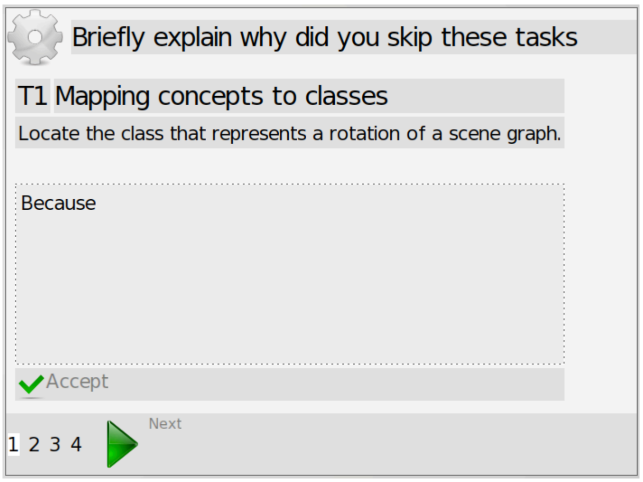
1. shapes
2. gap
3. scale
4. dimensions

Duration: 0:00:08:00

⊘	T1	real	[✓ possibly real]	almost-artifical	artificial
✓	T2	real	[✓ possibly real]	almost-artifical	artificial
⊘	T3.1	real	[✓ possibly real]	almost-artifical	artificial
✓	T3.2	real	[✓ possibly real]	almost-artifical	artificial
✓	T3.3	real	[✓ possibly real]	almost-artifical	artificial
✓	T4.1	real	[✓ possibly real]	almost-artifical	artificial
✓	T4.2	real	[✓ possibly real]	almost-artifical	artificial
✓	T5.1	real	[✓ possibly real]	almost-artifical	artificial
✓	T5.2	real	[✓ possibly real]	almost-artifical	artificial
✓	T6.1	real	[✓ possibly real]	almost-artifical	artificial
✓	T6.2	real	[✓ possibly real]	almost-artifical	artificial
✓	T6.3	real	[✓ possibly real]	almost-artifical	artificial
✓	T7.1	real	[✓ possibly real]	almost-artifical	artificial
⊘	T7.2	real	[✓ possibly real]	almost-artifical	artificial
⊘	T7.3	real	[✓ possibly real]	almost-artifical	artificial

1 2 Done

(a) Perceived difficulty of each task



Briefly explain why did you skip these tasks

T1 Mapping concepts to classes

Locate the class that represents a rotation of a scene graph.

Because

Accept

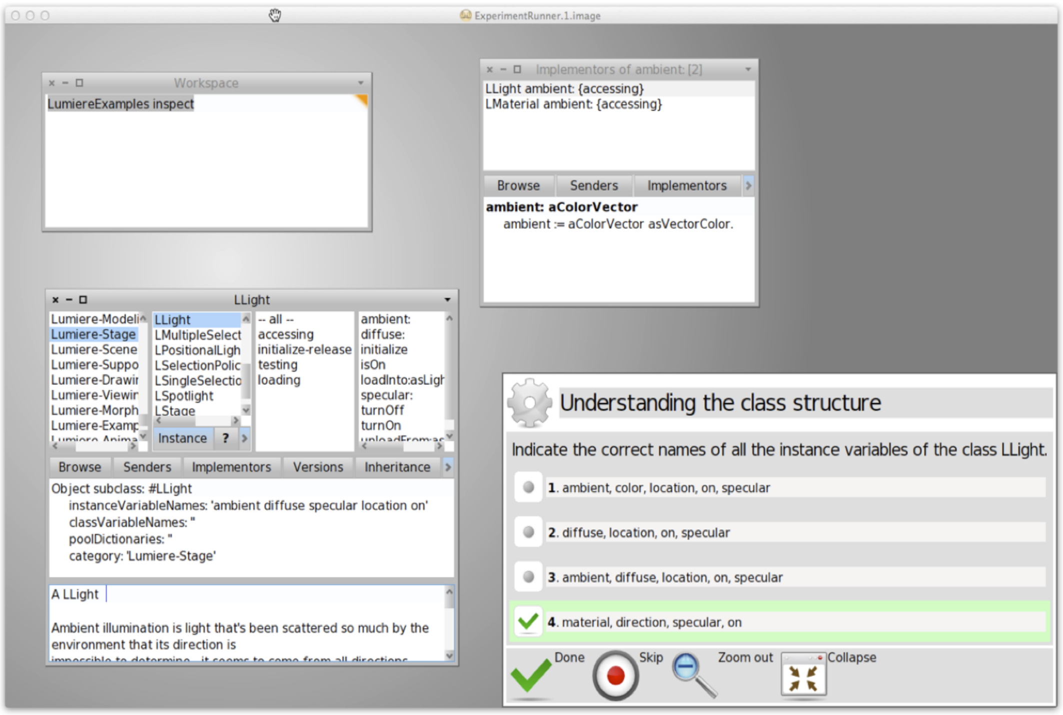
1 2 3 4 Next

(b) Skipped Tasks

Figure 4.7. The post-experiment questionnaire presented by Biscuit

4.4.5 Data collection

Using Biscuit, we automatically generated a user interface for each experimental session, via an application running either on top of Gaucho or Pharo. Figure 4.8 shows Biscuit running on top of Pharo.



```
LumiereExamples inspect
```

```
LLight ambient: {accessing}
LMaterial ambient: {accessing}
```

```
ambient: aColorVector
    ambient := aColorVector asVectorColor.
```

```
Object subclass: #LLight
    instanceVariableNames: 'ambient diffuse specular location on'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Lumiere-Stage'
```

A LLight

Ambient illumination is light that's been scattered so much by the environment that its direction is

Understanding the class structure

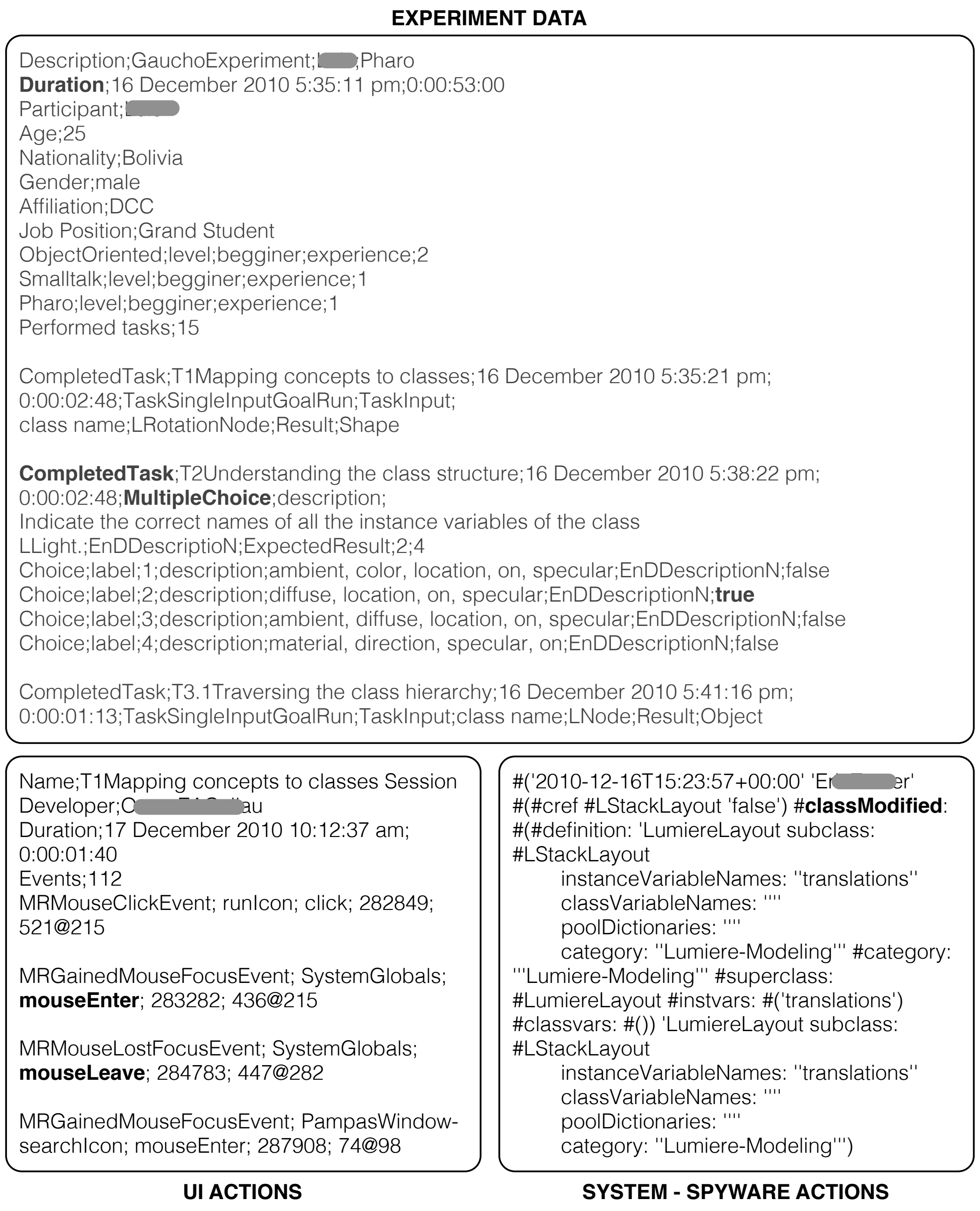
Indicate the correct names of all the instance variables of the class LLight.

1. ambient, color, location, on, specular
2. diffuse, location, on, specular
3. ambient, diffuse, location, on, specular
4. [✓ material, direction, specular, on]

Figure 4.8. Biscuit running on top of the Pharo IDE

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.

Figure 4.9 depicts the contents of the files that constitute the output of an experiment run. Biscuit records all the data pertinent to the subjects activity, ranging from meta-data such as the level of expertise of the subjects, to low level user interface actions, such as mouse clicks and keystrokes. We used Biscuit to analyze the results of the experiment by extracting the subjects' activity from these output files.



EXPERIMENT DATA

```
Description;GauchoExperiment;[redacted];Pharo
Duration;16 December 2010 5:35:11 pm;0:00:53:00
Participant;[redacted]
Age;25
Nationality;Bolivia
Gender;male
Affiliation;DCC
Job Position;Grand Student
ObjectOriented;level;begginer;experience;2
Smalltalk;level;begginer;experience;1
Pharo;level;begginer;experience;1
Performed tasks;15

CompletedTask;T1Mapping concepts to classes;16 December 2010 5:35:21 pm;
0:00:02:48;TaskSingleInputGoalRun;TaskInput;
class name;LRotationNode;Result;Shape

CompletedTask;T2Understanding the class structure;16 December 2010 5:38:22 pm;
0:00:02:48;MultipleChoice;description;
Indicate the correct names of all the instance variables of the class
LLight.;EnDDescriptioN;ExpectedResult;2;4
Choice;label;1;description;ambient, color, location, on, specular;EnDDescriptionN;false
Choice;label;2;description;diffuse, location, on, specular;EnDDescriptionN;true
Choice;label;3;description;ambient, diffuse, location, on, specular;EnDDescriptionN;false
Choice;label;4;description;material, direction, specular, on;EnDDescriptionN;false

CompletedTask;T3.1Traversing the class hierarchy;16 December 2010 5:41:16 pm;
0:00:01:13;TaskSingleInputGoalRun;TaskInput;class name;LNode;Result;Object
```

```
Name;T1Mapping concepts to classes Session
Developer;[redacted]
Duration;17 December 2010 10:12:37 am;
0:00:01:40
Events;112
MRMouseClickEvent; runIcon; click; 282849;
521@215

MRGainedMouseFocusEvent; SystemGlobals;
mouseEnter; 283282; 436@215

MRMouseLostFocusEvent; SystemGlobals;
mouseLeave; 284783; 447@282

MRGainedMouseFocusEvent; PampasWindow-
searchIcon; mouseEnter; 287908; 74@98
```

UI ACTIONS

```
#('2010-12-16T15:23:57+00:00' '[redacted]'
#(#cref #LStackLayout 'false') #classModified:
#(#definition: 'LumiereLayout subclass:
#LStackLayout
        instanceVariableNames: ''translations''
        classVariableNames: ''''
        poolDictionaries: ''''
        category: ''Lumiere-Modeling''' #category:
'''Lumiere-Modeling''' #superclass:
#LumiereLayout #instvars: #('translations')
#classvars: #()) 'LumiereLayout subclass:
#LStackLayout
        instanceVariableNames: ''translations''
        classVariableNames: ''''
        poolDictionaries: ''''
        category: ''Lumiere-Modeling''')
```

SYSTEM - SPYWARE ACTIONS

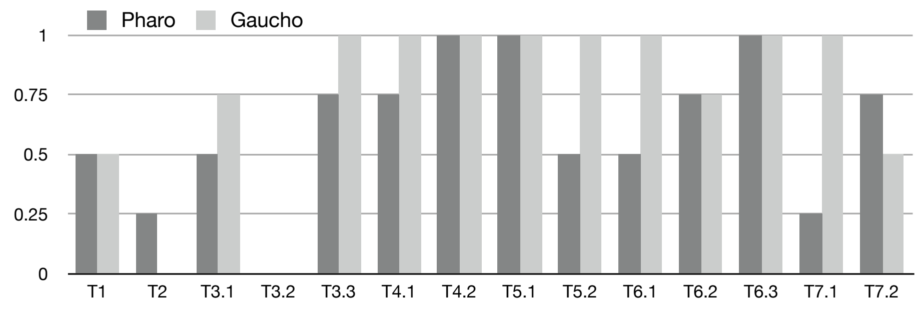
Figure 4.9. Biscuit output: reliable and precise data

4.4.6 Results

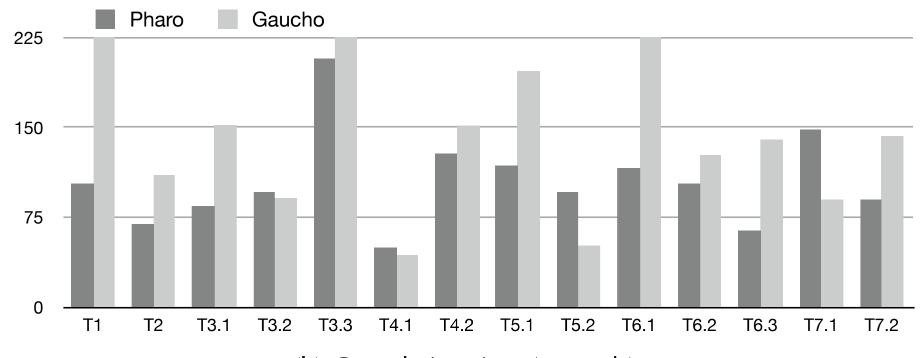
The 14 tasks were automatically graded, yielding a maximum score of 14 points, and the time taken to solve each task was measured by means of the output analyzer features of Biscuit. From that analysis we concluded that the subjects of the experimental group outperformed the subjects of the control group regarding the correctness of the tasks (cf. Figure 4.10(a)).

The subjects using Gaucho scored, on average, 10.4 points, while the subjects using the Pharo treatment scored, on average, 8.5 points. While regarding the completion time, the subjects of the control group outperformed the experimental group (cf. Figure 4.10(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.



(a) Task correctness



(b) Completion time (seconds)

Figure 4.10. The time and correctness of the subjects on the performed tasks

(a) Control Group using Pharo

Subject	**T1**	**T2**	**T3.1**	**T3.2**	**T3.3**	**T4.1**	**T4.2**
P1	🟥 02:48	02:48	🟥 01:13	🟥 02:14	05:59	01:20	03:42
P2	🟥 00:29	🟥 00:40	🟥 01:50	🟥 01:42	02:54	🟥 00:37	01:10
P3	01:22	🟥 00:30	00:41	🟥 00:58	🟥 02:47	00:19	01:20
P4	02:13	🟥 00:39	01:53	🟥 01:29	02:09	01:03	02:39
Avg	01:43	01:09	01:24	01:36	03:27	00:50	02:08

	T5.1	**T5.2**	**T6.1**	**T6.2**	**T6.3**	**T7.1**	**T7.2**
P1	03:32	🟥 02:20	🟥 03:44	03:38	01:40	🟥 03:03	00:45
P2	01:25	🟥 01:31	🟥 01:13	🟥 00:04	01:01	🟥 00:48	🟥 00:26
P3	01:40	01:18	00:54	00:41	00:31	🟥 03:07	01:01
P4	01:15	01:15	01:52	02:29	01:04	02:54	03:47
Avg	01:58	01:36	01:56	01:43	01:04	02:28	01:30

(b) Experimental Group using Gaucho

Subject	**T1**	**T2**	**T3.1**	**T3.2**	**T3.3**	**T4.1**	**T4.2**
G1	🟥 01:40	🟥 02:08	04:06	🟥 00:51	04:22	01:12	03:43
G2	06:31	🟥 01:58	01:22	🟥 01:42	06:14	00:28	01:50
G3	🟥 03:51	🟥 00:51	00:58	🟥 01:59	05:07	00:46	02:51
G4	03:00	🟥 02:24	🟥 03:41	🟥 01:31	04:27	00:28	01:42
Avg	03:46	01:50	02:32	01:31	05:03	00:44	02:32

	T5.1	**T5.2**	**T6.1**	**T6.2**	**T6.3**	**T7.1**	**T7.2**
G1	05:48	00:27	05:50	02:25	04:10	02:16	🟥 03:47
G2	01:27	01:47	02:36	00:55	00:45	00:39	02:40
G3	03:32	00:43	03:23	02:04	01:23	00:57	02:07
G4	02:20	00:29	05:16	🟥 03:04	03:01	02:06	🟥 00:56
Avg	03:17	00:52	04:16	02:07	02:20	01:30	02:23

Table 4.5. The Completion Times and Correctness (incorrect answers are colored red) of the subjects on the performed tasks

We now proceed with a qualitative evaluation of the results of each task. 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.

Concept Location: 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.

Structural comprehension of classes and packages: T2, T4.1, T4.2

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 the textual representation of the class definition.

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, 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 (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.

System Navigation: 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.

Unconstrained System Layout: 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, to assess the supposed benefit of the unconstrained layout of the Pampas and the simple graphical representations of objects.

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.

Class References: 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.

Programming: 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 between the correctness and completion time in T5.2, 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.

4.5 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.

4.5.1 On the correctness of the performed tasks

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 derived 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 estate, and hence compare two graphical elements.

4.5.2 On the completion time of the performed tasks

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 [14].

The subjects using Gaucho spent more time on most of the tasks because of some usability issues found in Gaucho 1.2, which resulted from the lack of an automatic layout or non-overlapping scheme. The latter 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 solved this issue by implementing a better layout policy scheme in the latest version of Gaucho.

4.5.3 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 increased the chances that we devised real tasks that developers frequently solve using traditional IDES. A supporting fact are the average perceived difficulties of the tasks, we gathered from the subjects during the post-experiment questionnaire (See Figure 4.11).

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.

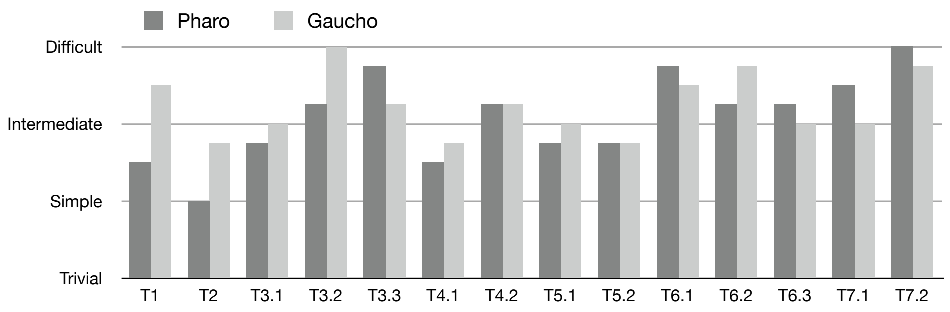


Figure 4.11. The Subject's Perceived Difficulty of the Tasks

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 (Goals, Operators, Methods, and Selection rules) [16]. Another threat is the small number of subjects who performed the experiment (8). Moreover, a better distribution of the expertise of the subjects would provide the means to analyze beginners and advanced IDE developers, and the experiment would benefit from a more balanced number of subjects from academia and industry.

4.6 Conclusions

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. The conclusions are two-fold, the first relating to the outcome of the experiment itself, whilst the second pertains to the instrumentation of the experiment using Biscuit. Regarding the experiment, 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.

4.6.1 Biscuit to the Rescue

Performing controlled experiments with human subjects is a difficult task, subject to many threats. Biscuit is an experimental toolkit aimed at reducing some of the threats related to recording and gathering experimental data.

Biscuit supports the recording of answers and timing information necessary for quantitative experiments; it also collects finer-grained data—user interactions—that, on the one hand, is useful for a qualitative analysis of the data and, on the other hand, makes the experiments fully replicable.

We raise the question of whether tools such as Biscuit can improve the quality of the process by which we currently perform controlled experiments; and more importantly, if the availability of more precise and reliable data can eliminate some of the numerous threats present in controlled experiments, which are manually conducted by fallible humans. Contrasting Gaucho's controlled experiment [32] (conducted using Biscuit) and our previous experiment [68] (conducted in the traditional manner) we noticed that:

- by relieving ourselves from book-keeping tasks, we removed ourselves as threats to validity;
- we were able to better observe the subjects during the experimental run, gathering in the process much more qualitative data about Gaucho's usability;
- data post-processing was greatly simplified;
- since the Gaucho experiment featured several very short tasks (some with a duration under a minute), we doubt that conducting the same experiment would have been possible in a "traditional" (*i.e.,* manual) way.

Conducting controlled experiments with human subjects to evaluate software engineering tools and approaches has become a necessity. Such experiments come with many threats to validity because of the humans involved; we believe Biscuit helps to remove some of the threats that regard the operation of controlled experiments.

Chapter 5

Object-Focused Collaboration

In this dissertation we discuss an extensible design for the interface of object oriented development environments. To demonstrate the extensibility of our approach, in this chapter we investigate its application to collaboration.

The art of crafting programs to solve problems is rarely accomplished by a single human working in solitude. Early psychological theories of programming [5] acknowledged that the software development process is a social human task, and practitioners have observed that therein lies the main cause of project failures [76]. Nonetheless, the main vehicle for programming—the integrated development environment (IDE)—remains as it was conceived in the 1970s, focused on a single point of view of the system.

IDEs were designed without collaboration in mind, hence team members are aware of system changes only after the code is committed to the versioning system, which delays discussions that would otherwise prevent conflicts. There have been attempts to provide better collaboration support in existing IDEs, such as Palantir [77] and Syde [78]. However, in these cases collaboration support is an afterthought stapled on top of existing environments that struggle to overcome their single-developer nature.

In this chapter we present *Ronda*, an extension to *Gaucho*, which offers first-class support for collaborative development sessions, and promotes awareness of fine-grained changes to the system under development.

Structure of the chapter. In Section 5.1 we motivate the need for novel development environments designed to support a team of programmers. In Section 5.2 we describe Ronda, a collaborative object-focused environment. In Section 5.3 we conclude the chapter, detailing the infrastructure we built to support team collaboration within the scope of sessions in Ronda.

5.1 On Object-Focused Collaborative Environments

In this section we motivate the need for *Ronda*, a novel IDE devised from the ground up to support collaborative sessions within an object-focused development environment.

5.1.1 Shared Development Sessions

The interaction amongst people assembled in groups can be categorized into focused and unfocused gatherings. In the former, several participants get together for a clearly stated purpose, while in the latter each of them might have different goals during the rendezvous [79].

Most of the collaborative development environments developed to date, provide some form of unfocused gathering by means of a shared editable view of the system, or by visualizing the modifications made by other developers. For example, in the 1990s researchers at Sun Microsystems devised Kansas, a 2D space for real-time collaboration, including a shared large flat space which hosted directly-manipulable representations of the objects [80] (see Figure 5.1). In Kansas, any change to the system is immediately displayed to every developer, but the system lacks the concept of a session to guide developers into which modifications must be performed.

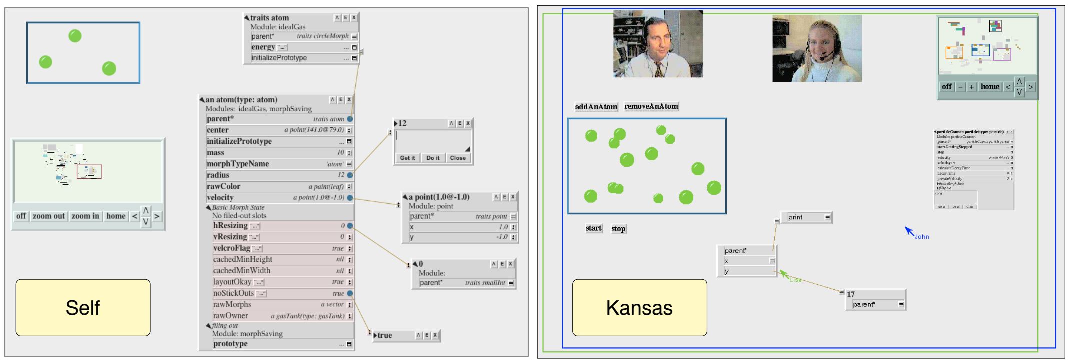
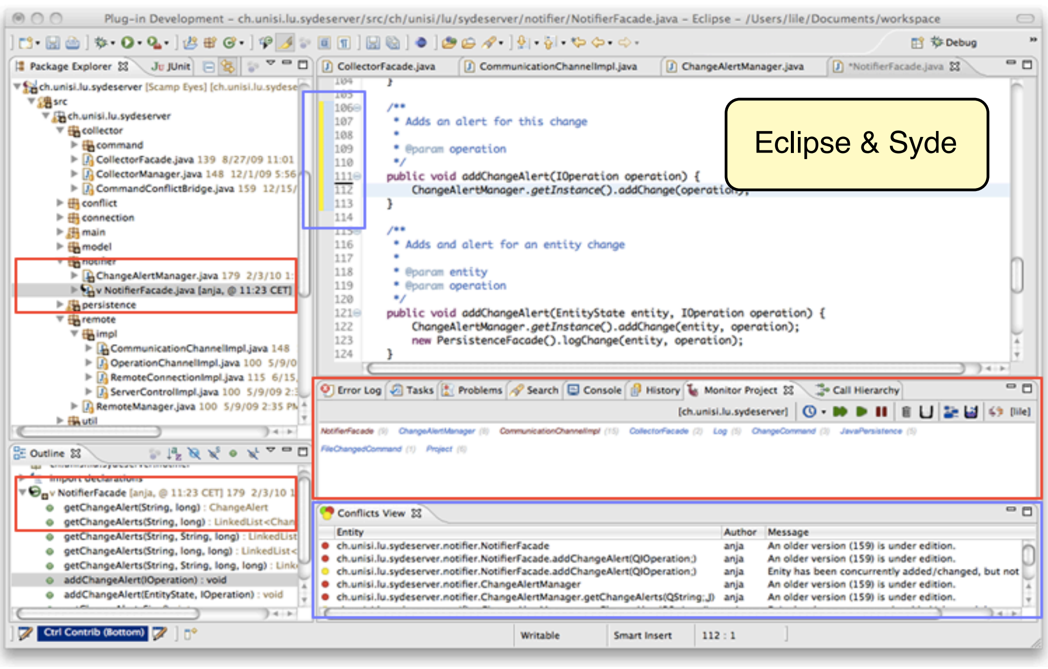


Figure 5.1. Self and Kansas: collaboration within an object-focused IDE

More recent examples are Syde and Jazz. Syde is a set of plugins that augment the Eclipse IDE with awareness of fine-grained changes to the system [78] (see Figure 5.2). Jazz[1], is a collaboration platform that can integrate with the IDE to enable task tracking capabilities and source control.

[1]`https://jazz.net/`

We argue that such environments are missing a fundamental piece of the puzzle: A first-class presence of shared development sessions, with clearly defined boundaries, objectives, and outcome. In *Ronda*, development sessions are first class objects, which provide the overall context when accomplishing tasks using the IDE.



```
    }

    /**
     * Adds an alert for this change
     *
     * @param operation
     */
    public void addChangeAlert(IOperation operation) {
        ChangeAlertManager.getInstance().addChange(operation);
    }

    /**
     * Adds and alert for an entity change
     *
     * @param entity
     * @param operation
     */
    public void addChangeAlert(EntityState entity, IOperation operation) {
        ChangeAlertManager.getInstance().addChange(entity, operation);
        new PersistenceFacade().logChange(entity, operation);
    }
```

Entity	Author	Message
ch.unisi.lu.sydeserver.notifier.NotifierFacade	anja	An older version (159) is under edition.
ch.unisi.lu.sydeserver.notifier.NotifierFacade.addChangeAlert(QIOperation;)	anja	An older version (159) is under edition.
ch.unisi.lu.sydeserver.notifier.NotifierFacade.addChangeAlert(QIOperation;)	anja	Enity has been concurrently added/changed, but not
ch.unisi.lu.sydeserver.notifier.ChangeAlertManager	anja	An older version (159) is under edition.
ch.unisi.lu.sydeserver.notifier.ChangeAlertManager.getChangeAlerts(QString;J)	anja	An older version (159) is under edition.

Figure 5.2. Syde: collaboration support within the Eclipse IDE

5.1.2 Object-focused environments

We designed *Ronda* around an alternative user interface to traditional IDEs, to avoid the many problems that they struggle to overcome, mostly related to the allocation of real estate resources [24]. Our goal is to escape from the bento box model that confines the IDE within a single window with sub-panes [24]. The bento-box model forces Syde's plugins to compete for a portion of real estate with the traditional tools of Eclipse (see Figure 5.2). A 2D open-space IDE in the vein of Self, on the other hand, easily accommodates collaborative aspects due to its libertarian usage of screen space, and a more concrete representation of the objects in the interface.

The use of direct manipulation enables a more focused display of visual cues to denote changes to the system, given the continuous representation of the objects of interest characteristic of such interfaces [44]. For instance, in Syde, which is built on top of a traditional IDE, the notification of changes is detached from the actual portion of the source code describing the changed entities, resulting in a so called “ping pong” interface [81].

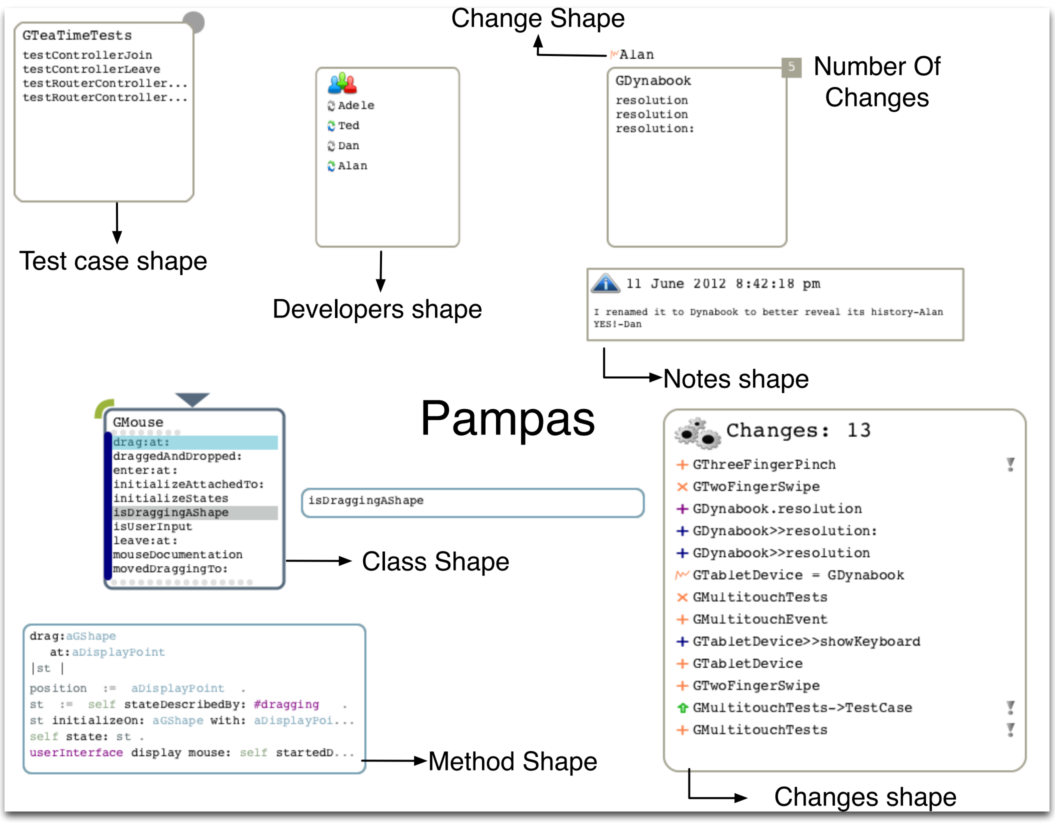
We want to provide a framework that enables the creation, announcement, development, and tracking of sessions, enabling team members to engage in *focused gatherings* within an object-focused environment. This is the main principle of *Ronda*, presented next.

5.2 Ronda

Ronda is an object-oriented development environment that enables a group of developers to remotely collaborate to accomplish tasks within the scope of sessions. *Ronda* is built on top of *Gaucho*.

5.2.1 Awareness of Fine-Grained Changes

Ronda is a change-centric development environment that includes several shapes providing the means to fully manipulate the represented objects. For example, developers can create, rename, remove, and add methods and variables to classes by solely interacting with a *Class Shape*. We track fine-grained changes within the IDE, to provide real-time awareness support, hence every (minor) change to the system is immediately broadcasted to the other participants to attain a level of awareness, which is simply not possible in single-person mainstream IDEs, where—as previous research pointed out [82]—often developers engage in a blind race to commit first and avoid the merge of conflicting changes. However, we believe the use of the object-focused metaphor, as opposed to traditional bento-box interfaces, provides better support for visualizing those changes and revealing conflicts, mostly because of the stronger presence of objects within the interface (*i.e.* high-level views of objects vs enriched source code editors). For instance, Figure 5.3 depicts a class shape that was renamed by another participant. The shape presents visual cues for quickly understanding the nature of the change and who produced it.



```
GTeaTimeTests
testControllerJoin
testControllerLeave
testRouterController...
testRouterController...
```

```
Adele
Ted
Dan
Alan
```

```
Alan
GDynabook                5
resolution
resolution
resolution:
```

```
11 June 2012 8:42:18 pm
I renamed it to Dynabook to better reveal its history-Alan
YES!-Dan
```

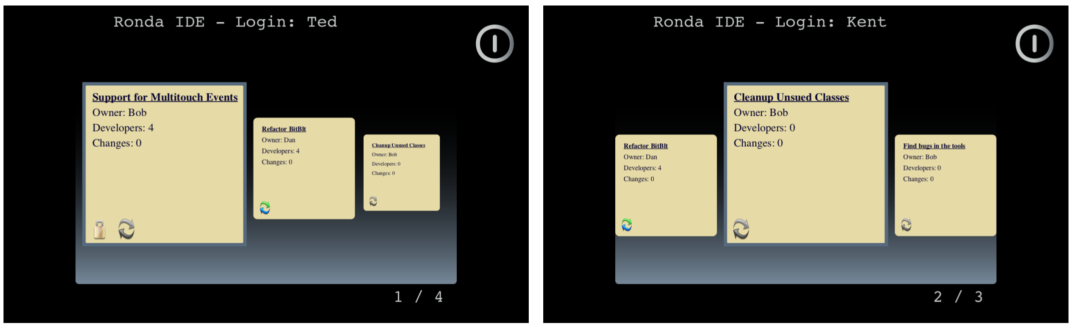
```
GMouse
drag:at:
draggedAndDropped:
enter:at:
initializeAttachedTo:
initializeStates
isDraggingAShape
isUserInput
leave:at:
mouseDocumentation
movedDraggingTo:
```

```
isDraggingAShape
```

```
drag:aGShape
   at:aDisplayPoint
|st |
position  :=  aDisplayPoint  .
st  :=  self stateDescribedBy: #dragging  .
st initializeOn: aGShape with: aDisplayPoi...
self state: st .
userInterface display mouse: self startedD...
```

```
Changes: 13
+ GThreeFingerPinch
× GTwoFingerSwipe
+ GDynabook.resolution
+ GDynabook>>resolution:
+ GDynabook>>resolution
  GTabletDevice = GDynabook
× GMultitouchTests
+ GMultitouchEvent
+ GTabletDevice>>showKeyboard
+ GTabletDevice
+ GTwoFingerSwipe
⇧ GMultitouchTests->TestCase
+ GMultitouchTests
```

Figure 5.3. A *Ronda* session: the Pampas including several Shapes



```
Ronda IDE - Login: Ted

Support for Multitouch Events
Owner: Bob
Developers: 4
Changes: 0

Refactor BitBit
Owner: Dan
Developers: 4
Changes: 0

Cleanup Unused Classes
Owner: Bob
Developers: 0
Changes: 0

1 / 4
```

```
Ronda IDE - Login: Kent

Refactor BitBit
Owner: Dan
Developers: 4
Changes: 0

Cleanup Unused Classes
Owner: Bob
Developers: 0
Changes: 0

Find bugs in the tools
Owner: Bob
Developers: 0
Changes: 0

2 / 3
```

Figure 5.4. Ronda: the initial display

5.2.2 Shared Development Sessions

Figure 5.4 depicts two different initial displays presented when *Ronda* developers open the environment, and are presented with the available sessions they can join. The sessions have a named *task* that describes the purpose of the gathering, an *owner* who is responsible for closing and committing its outcome, a list of *developers* who can participate, and a list of those who are logged in. When a developer joins a session, *Ronda* synchronizes to an updated state of the ongoing session, by downloading and installing a snapshot containing the system under development and all the changes performed so far, and then opens the session in the interface. Figure 5.4 depicts Ronda at startup.

5.2.3 Change Authoring and Trust Levels

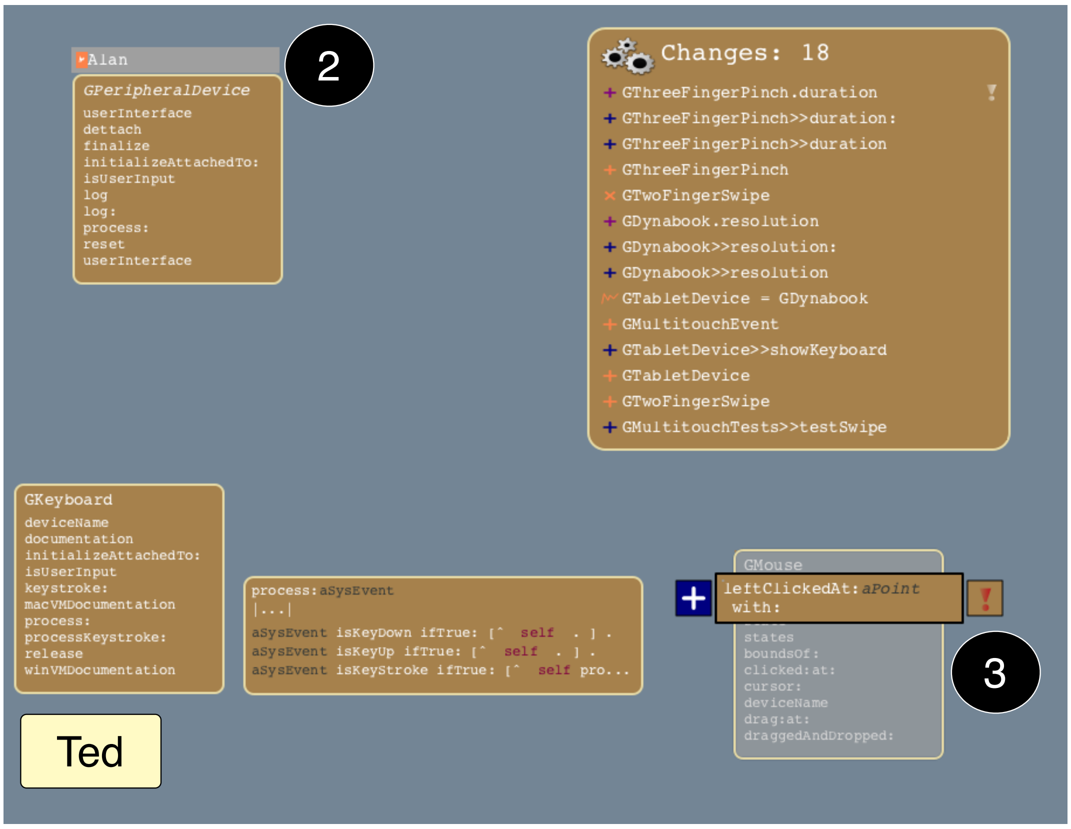
In *Ronda*, we distinguish between trusted and untrusted developers. The former produce trusted changes, whereas modifications of the latter result in untrusted changes, which are visualized differently, and might be discarded by more knowledgeable trusted developers. The trust levels are granted by the owner when creating the session, by enumerating the trusted developers and specifying whether untrusted developers can join.

Figure 5.5 portrays the *Ronda* interface of an ongoing session of two different developers. To the right, the pampas of Ted, and to the left the pampas of Alan. The session has several past changes and ongoing editions, revealed in the depicted shapes. For instance:

① The class GThreeFingerPinch was modified twice, the last one performed by Kent, an untrusted change that added a variable. ② The edition in progress consisting in the renaming of the class GPeripheralDevice by Alan. ③ Another edit in the form of a method addition to the class GMouse by Ted.

5.2.4 Avoiding Conflicts

Even though developers in the same team seldom work on the same objects at the same time, conflicts may occur because they are working to solve the same task. In *Ronda*, we avoid conflicts by broadcasting any shape edit which might lead to a system change, thus developers have a consistent view of the shapes—the objects—currently under manipulation: A session tracks both changes and edits, which are any manipulation which might result in a system change. For instance, opening a class rename shape or a method add shape, and receiving input from the developer.



```
Alan
GPeripheralDevice
userInterface
dettach
finalize
initializeAttachedTo:
isUserInput
log
log:
process:
reset
userInterface
```

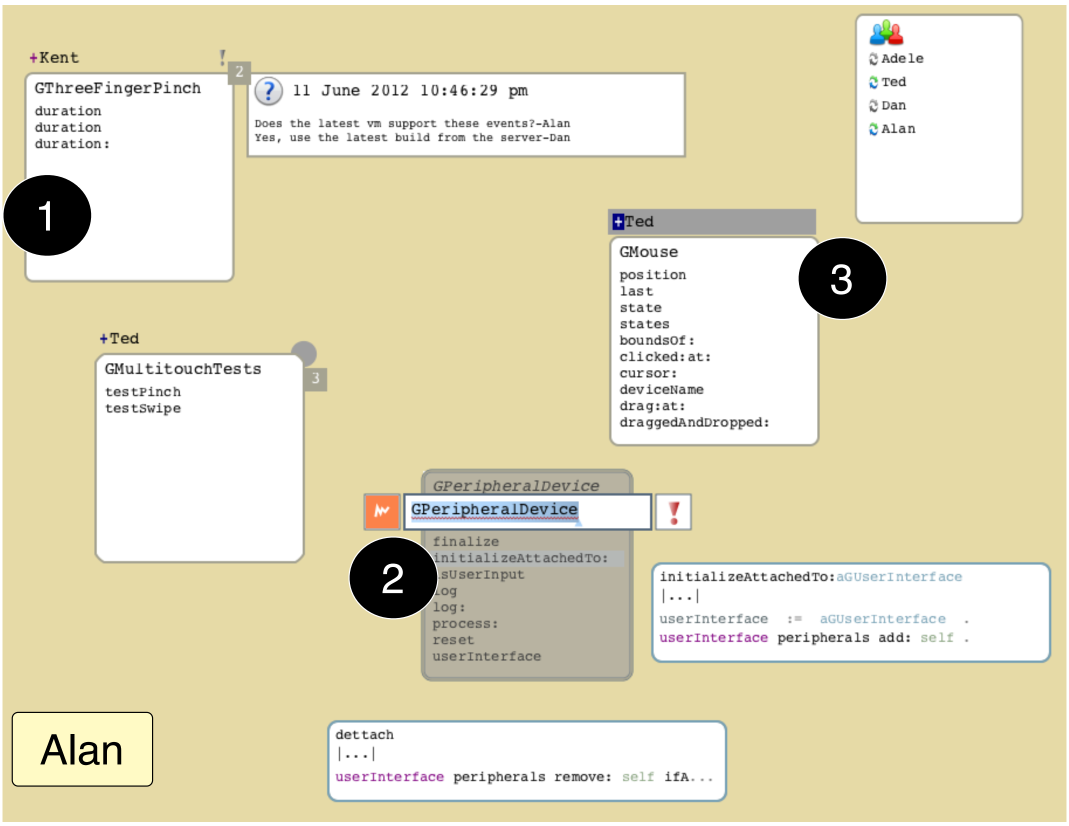
```
Changes: 18
+ GThreeFingerPinch.duration
+ GThreeFingerPinch>>duration:
+ GThreeFingerPinch>>duration
+ GThreeFingerPinch
× GTwoFingerSwipe
+ GDynabook.resolution
+ GDynabook>>resolution:
+ GDynabook>>resolution
  GTabletDevice = GDynabook
+ GMultitouchEvent
+ GTabletDevice>>showKeyboard
+ GTabletDevice
+ GTwoFingerSwipe
+ GMultitouchTests>>testSwipe
```

```
GKeyboard
deviceName
documentation
initializeAttachedTo:
isUserInput
keystroke:
macVMDocumentation
process:
processKeystroke:
release
winVMDocumentation
```

```
process: aSysEvent
|...|
aSysEvent isKeyDown ifTrue: [^ self . ] .
aSysEvent isKeyUp ifTrue: [^ self . ] .
aSysEvent isKeyStroke ifTrue: [^ self pro...
```

```
GMouse
leftClickedAt: aPoint
 with:
states
boundsOf:
clicked:at:
cursor:
deviceName
drag:at:
draggedAndDropped:
```

(a) The ronda session of Ted



```
Kent
GThreeFingerPinch
duration
duration
duration:
```

```
11 June 2012 10:46:29 pm
Does the latest vm support these events?-Alan
Yes, use the latest build from the server-Dan
```

```
Adele
Ted
Dan
Alan
```

```
Ted
GMultitouchTests
testPinch
testSwipe
```

```
Ted
GMouse
position
last
state
states
boundsOf:
clicked:at:
cursor:
deviceName
drag:at:
draggedAndDropped:
```

```
GPeripheralDevice
GPeripheralDevice
finalize
initializeAttachedTo:
isUserInput
log
log:
process:
reset
userInterface
```

```
initializeAttachedTo: aGUserInterface
|...|
userInterface := aGUserInterface .
userInterface peripherals add: self .
```

```
dettach
|...|
userInterface peripherals remove: self ifA...
```

(b) The ronda session of Alan

Figure 5.5. Two ronda sessions depicting generation and awareness of changes

5.3 Tea Time

Ronda implements a simplified version of TeaTime [83], a decentralized distributed framework that relies on replication of computation instead of data. TeaTime revolves around the concept of an Island, which is a secure container of objects. An Island is an abstract concept with no inherent location.

The Islands are *projected* onto numerous concrete replicas, located in hosts of the network. Consistency amongst replicas is maintained by broadcasting any message that alters the state of the Island, via controllers connected to the same router, following a two-phase commit protocol. TeaTime messages originate in a host, then travel from the controller to the router, and finally get dispatched to all the connected controllers, including the original one.

The state-changing messages are generated in response to events performed by the developer, when manipulating the objects of the Island via their graphical counterpart within the user interface. The messages sent by the router are ordered by a sequence number and a timestamp, to preserve the order of execution of all received messages in each replica. Thus, the replicas deterministically evolve over time, because each replica is an exact copy of the Island *i.e.* they hold the same objects, and send the same ordered stream of messages.

5.3.1 Customizing TeaTime for *Ronda*

In *Ronda*, a TeaTime Island includes the shared development sessions, replicated in the IDE of all collaborating developers. A development session includes the system under construction, a list of trusted developers, and all the past changes.

Figure 5.6 depicts the core architecture in *Ronda*, consisting of one or more developers running a *Ronda* IDE (Alan, Ted, Dan, and Adele), composed of a controller connected to the Island's router via a TCP Socket, a replica of the ongoing Session, and an augmented *Gaucho* IDE.

When developers manipulate *Gaucho* shapes:

① a Tea Time message in the form of a UI command is generated ②, that either represents a fine grained system change or a UI element editing, like a class rename. The command is sent to the controller and forwarded to the island's router ③. The router broadcasts the command to all the connected controllers ④. Afterwards, upon reception, the command is executed producing the same result in every replica ⑤, which results in a change ⑥ that alters the ongoing session, and is presented in the user interface of the IDE ⑦.

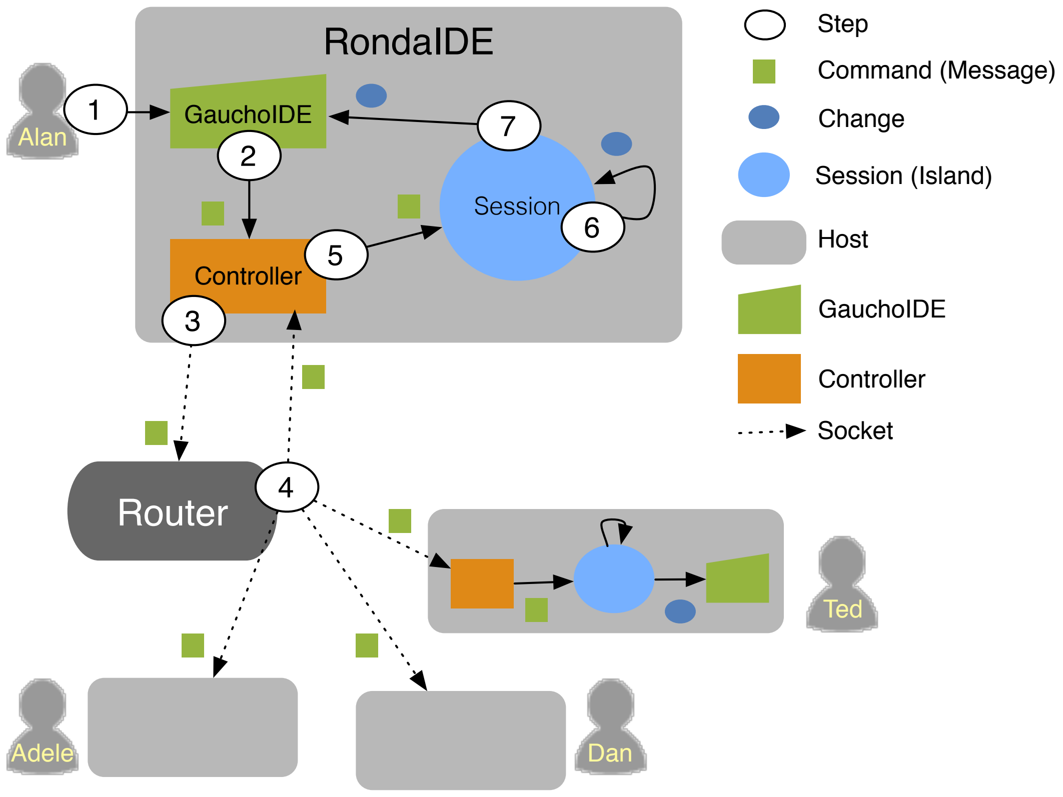


Figure 5.6. The infrastructure for collaboration in Ronda

A UI command communicates the intention of changing an object of the session, broadcasted and executed on each island replica to produce the same changes. In the non-collaborative Gaucho environment, the session simply *performs* and records the change to the isolated local copy of the system. On the other hand, in ronda the session *broadcasts* the system change to all its replicas, which finally perform the command when they receive the broadcasted message from the router, *i.e.* note that this occurs on each replica.

5.4 Summary

We have presented *Ronda*, a novel IDE designed to support collaboration by means of shared development sessions. Ronda is a change-centric environment that tracks and visualizes fine-grained changes to the system under construction. We also described the infrastructure based on Tea-Time, a distributed framework that relies on replication of computation instead of data.

Chapter 6

Conclusions

Computer programming requires humans with the necessary skill to create the programs, translating the solutions to problems from the human brain, into a computer readable form. Since the beginning of computing, programmers recur to the aid of tools. From the punch cards of yore, to the word processors of teleprinters or terminal displays of mainframe computers. With the advent of the personal computer era, programming tools irrevocably turned to an interactive nature, making use of graphical user interfaces.

Nowadays, programmers use IDEs, which include numerous tools to effectively construct programs, and perform all the tasks related to software engineering. From the original Smalltalk systems, which introduced the IDE, Object-Oriented Programming is achieved using a tool-based interface: the tools relegate the objects within the interface. An IDE using a tool-based interface has been found to produce both conceptual and technical problems, the former occurs because of the objects remain hidden behind the tools. The latter relates to a need for a finer-grained manipulation, for better real state management of a scarce resource–the display–, and the use of software visualization techniques.

In this dissertation we introduced an approach that promotes direct manipulation of objects as the sole means of programming in OOP languages. Our thesis is that an object-focused environment supports an broad and extensible range of software engineering tasks, and alleviates the conceptual and technical problems of traditional IDEs. To this aim, we designed several interfaces based on our approach, and we implemented them in several tools. To validate our thesis, we created and evaluated, on top of our approach, an object-focused environment for modeling, programming, and collaborating. We presented all the interfaces, showing that they support various software engineering tasks, and provide a plausible alternative to the traditional tool-based IDEs.

6.1 Contributions

During the course of this dissertation, we made a series of contributions to the state of the art in software development environments for object oriented programming languages.

We summarize the major ones in the following.

The definition of an object-focused interface for software development [29]. We defined an interface for object-oriented development based on two pivotal concepts: the *Shapes* and the *Pampas*. Shapes are directly manipulable representations of objects, that enable a fine-grained manipulation of all the software artifacts.

We designed the shapes to concisely visualize the structure and composition of the objects, and customized to each kind of object in the system to provide a complete set of interactions. The behavioral completeness of the Shapes disregards the need for extrinsic tools, to enhance the illusion that the graphical element on display is the object.

The Pampas is a 2D surface, where the shapes are freely placed to favor the use of spatial reasoning and memory, the comparison of objects by creating side by side views, and the customization of own views of the program, persisted across multiple sessions to support task context.

The application of the object-focused interface to programming [30]. We designed a set of shapes for the crafting of models composed of objects, described in class-based object oriented languages.

The shapes represent classes, methods, test-cases and packages of the system under construction: a class shape depicts the structure of the represented class, and a method shape presents the method signature and the statements that make up the body.

We described the interactions between the programmers and the shapes to craft the programs, and undertake program comprehension tasks.

The implementation of a tool which supports our object-focused interface [31]. We developed **Gaucho**, an object-focused programming environment for the crafting of models composed of objects, described in class-based object oriented languages. Gaucho is publicly available and has been used in academic research.

The empirical validation of our approach through a controlled experiment [32]. We performed a controlled experiment to compare a traditional IDE and Gaucho with respect to a set of program comprehension tasks extracted from the literature, such as creating, navigating, refactoring, and understanding an object-oriented system.

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. The experiment was fully instrumented using Biscuit. We include the entire experimental data set and everything that is required to make the experiment repeatable, in Chapter 4.

The implementation of a tool to support human-centric controlled experiments [33]. We designed and implemented **Biscuit**: to support researchers when performing controlled experiments. Biscuit records relevant pieces of information regarding an experiment performed with human subjects.

Conducting experiments to evaluate software engineering tools and approaches has become a necessity. Such experiments come with many threats to validity because of the humans involved, Biscuit helps to remove some of the threats that regard the operation of experiments. We illustrate the main concepts of Biscuit, and the design of our experiment in Appendix A.

The application of the object-focused interface to collaboration [34]. We investigated the use of object-focused environments to support collaboration by means of shared development sessions.

We designed a change-centric environment which tracks and visualizes fine-grained changes to the system under construction, to provide real-time awareness support amongst the team members.

We shown how an object-focused interface provides better support for visualizing those changes and revealing conflicts, because of the stronger presence of objects within the interface: the shape presents visual cues for quickly understanding the nature of the change and who produced it.

The implementation of a tool to support collaboration [34] We presented **Ronda**, an extension to Gaucho, which provides first-class support for collaborative development sessions. Ronda is a change-centric environment: It promotes awareness of fine-grained changes to the system.

6.2 Future Work

Our design of an object-focused environment could be enhanced in the following directions:

6.2.1 Semantics

In Gaucho, the various types of objects are represented differently, the interface uses a set of customized shapes for the models, classes, methods, packages, and test cases. Nevertheless, there are still missing opportunities for conveying the semantics, the state and behavior of the particular objects within each category. For example, Code City is a tool that uses software visualization techniques, to detect design flaws in the programs, by presenting distinct visual cues to denote several Object-Oriented Metrics [84; 85].

We could could use a similar approach in Gaucho, given the inherent visual nature of an object-focused environment, which enables to customize the appearance of each individual shape. Another possibility is to empower programmers to come up with their own visual language for expressing the meaning of their programs, as in the Silhouette environment [86].

6.2.2 Relations

Programmers spend most of their time performing maintenance tasks, navigating the system, and building an understanding of the relevant objects and the relationships between them. The Gaucho Shapes assume a passive role, instead of actively helping programmers into forming working sets of relevant objects. We envision a recommendation system built into the shapes, that automatically suggests hints for further exploration of the system to the programmer, visualizing dependencies between related objects to reduce the navigation [14]. Another direction of research involves the implementation of an organic layout of the system, where the objects place themselves dynamically, according to one or more relations with their collaborators, such as inheritance and package containment [87].

6.2.3 Education

Using graphical environments for teaching object oriented programming to students is beneficial [88]. Many different object oriented learning environments exist, for example BlueJ for Java [89; 90], and LOOP for Smalltalk [91]. An object focused environment such as Gaucho, could be enhanced to accommodate pedagogical tasks. In Gaucho, the Shapes abstract away from source code, thus easing the understanding of the components of the program, and the directness of the interface enables beginners to quickly produce models, to learn the main concepts of the object oriented paradigm.

6.2.4 Live Programming Tools

In the recent year, the concept of live programming has been included in many teaching environments [1] [2]. A live programming interface augments the textual representation of the statements, *i.e.* the lines of code, with several visual cues to better understand their meaning -syntax- and results -semantics. In this dissertation we investigated alternative interfaces for developing in class-based OOP languages, where manipulating objects that form the programs is paramount. The interface proposed in this thesis, could be extended to include such features by augmenting the method shapes, which present the statements of the methods.

6.3 Closing Words

Object oriented programming is based on the computational model of a world of collaborating objects. An inherently dynamic world, where live objects are ready to carry out each others requests in the form of messages.

The user interface of a traditional IDE forces a misconception of the concept of a program in the object oriented programming paradigm, which is that the program and the executable are separate entities, therefore the objects come to life when the program is up and running, and cease to exist when the program terminates or aborts its execution.

Acknowledging that the tools influence our own thinking habits, and we humans –in particular programmers– tend to disregard the impact of this influence, we advocate for a different design of the interfaces of our development environments, one that avoids the misconception and "brings objects closer to the programmers".

In our work we designed an integral development environment, were programmers converse with the objects that make up the system, by interacting

[1] `http://worrydream.com/#!/LearnableProgramming`

[2] `http://toplap.org`

with reactive visual shapes, from the models to the running programs. Enabling a dynamic "conversational interface", because an object-focused environment permeates through all the environment, differently than the traditional tool-based IDEs, were the tools are separated from the software artifacts.

Given that the technology for the instrumenting of user interfaces is evolving fast, its time for adapting our programming habits as well, and adopt a more immersive environment for manipulating objects all the way.

Appendix A

Experiment Design with Biscuit

In this chapter, we describe the infrastructure of Biscuit. We start with the classes that model an experiment, then continue with the classes that represent a running experiment, and finish by presenting the support in Biscuit for automatically generating a user interface for the experiment run. We present the classes and their relations with several UML class diagrams. We conclude this chapter by illustrating the actual classes and methods that compose our experiment on Gaucho, we presented in Chapter 4.

A.1 Modeling and Running an Experiment

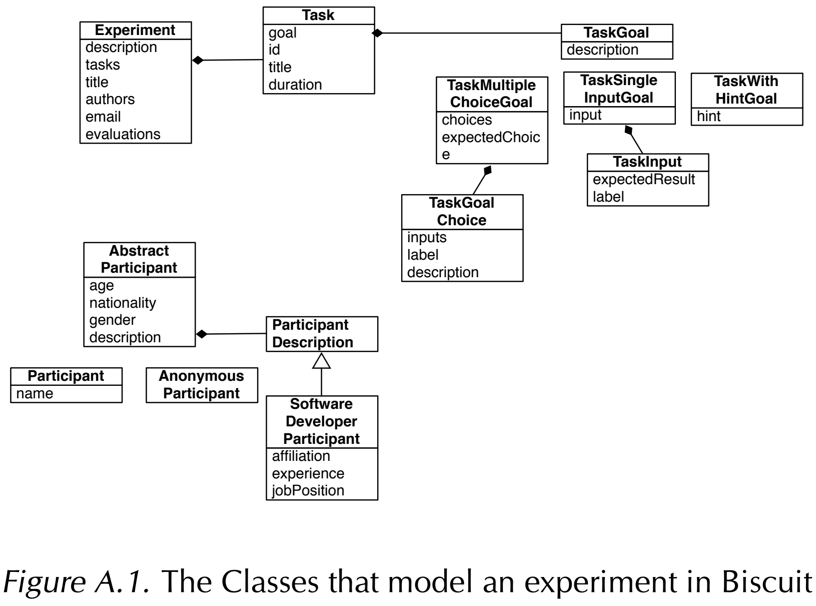


Figure A.1. The Classes that model an experiment in Biscuit

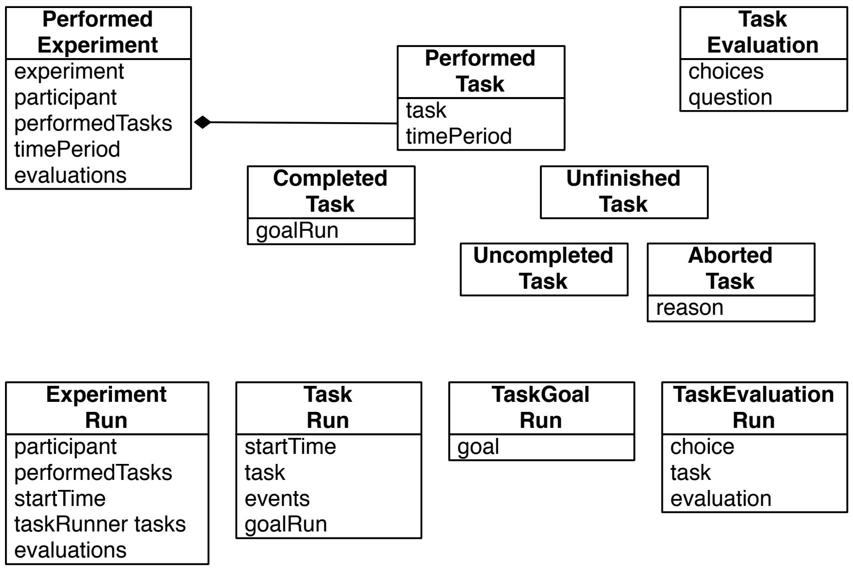


Figure A.2. The Classes that model an experiment in Biscuit

A.2 The User Interface of the Experiment Run

Biscuit is implemented in Smalltalk, the same language we use for showcasing the actual classes and methods that make up our experiment. If the reader is not familiar with the language we advise him to read the summary presented by Nierstrasz and Gîrba [56].

The main responsible for generating and running an experiment in Biscuit is the following:

```
Object subclass: #ExperimentRunnerGUI
  instanceVariableNames: 'experimentRunner currentMorph fullscreenMorph
     performedExperiments dialog performedExperiment displaySize experiment'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'ExperimentRunnerGUI−Morphic'
```

To start running an experiment by automatically generating a user interface to the subjects, the following message must be evaluated:

```
ExperimentRunnerGUI>>startRunning: anExperiment
  self adjustForDisplaySize.
  Display fullScreen: true .
  Author fullName: 'ExperimentRunner'.
  experiment := anExperiment.
  self openCreateParticipant.
```

The running experiment takes up the whole screen, and starts by presenting the pre-questionaire to the subject. When the experiment is completed, Biscuit terminates itself by sending the following message:

```
ExperimentRunnerGUI>>endExperimentRun
  Display fullScreen: false.
  performedExperiment := experimentRunner asPerformedExperiment.
  performedExperiments add: performedExperiment.
  self saveLastPerformedExperiment.
  self openSendUsTheResults
```

Biscuit is built on top of Pharo, and uses its base graphic framework named Morphic. We extended the framework with our own set of basic widgets, and several more specialized ones representing the running experiment in the user interface. Amongst the many Morphs that compose the interface of a running experiment are the ExperimentWelcomeMorph, ExperimentMorph, ExperimentRunnerMorph, TaskMorph, TaskGoalMorph, TaskChoiceMorph, TaskRunMorph, and ParticipantMorph.

A.3 Designing the Experiment on Gaucho

The entry point to our experiment is the class **Experiment**, which answers the message *#gaucho2010* with the actual object representing the experiment on Gaucho.

```
Experiment>>gaucho2010
  ^ GauchoExperiment2010Builder current build.
```

In Biscuit the experiments are created using the Builder pattern. The following class models an experiment builder.

```
Object subclass: #ExperimentBuilder
  instanceVariableNames: 'experiment'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'ExperimentRunner'
```

The class **ExperimentBuilder** has the responsibility of returning a complete instance of an particular experiment, by answering the message *#build*.

```
ExperimentBuilder>>build
  experiment := Experiment
    entitled: self title
    describedBy: self description
    withTasks: self tasks
    withEvaluations: self evaluations
    authoredBy: self authors
    sendingResultsTo: self email
    from: self participantEmail.
  ^ experiment
```

The ExperimentBuilder class is abstract, and must be specialized for each concrete experiment that a researcher wants to design. In our example, we illustrate the GauchoExperiment2010Builder class, which has a boolean instance variable to indicate which treatment –either Pharo or Gaucho– the experiment should be customized to.

```
ExperimentBuilder subclass: #GauchoExperiment2010Builder
  instanceVariableNames: 'usingPharo'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'ExperimentRunner'
```

Other example is the CodeCityExampleBuilder, which reproduces in Biscuit the complete experiment performed by Wettel for assessing the validity of visualizing software systems as cities [68].

```
ExperimentBuilder subclass: #CodeCityExampleBuilder
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'ExperimentRunner'
```

The GauchoExperiment2010Builder class overrides the method *#tasks* with the actual tasks that form our experiment. The tasks are first class objects, and are created by the builder. All the objects that compose a Task are modeled in Biscuit. The following illustrates the creation of tasks of our experiment.

```
GauchoExperiment2010Builder>>tasks
  | tasks |
  tasks := OrderedCollection new.
  tasks
    add: self taskT1 ;
    add: self taskT2 ;
    add: self taskT31 ;
    add: self taskT32 ;
    add: self taskT33 ;
    add: self taskT41 ;
    add: self taskT42 ;
    add: self taskT51 ;
    add: self taskT52 ;
    add: self taskT61 ;
    add: self taskT62 ;
    add: self taskT63 ;
    add: self taskT71 ;
    add: self taskT72 ;
    add: self taskT73 .
  ^ tasks.

GauchoExperiment2010Builder>>taskT42
  ^ Task
    entitled: 'Package sizes'
    withGoal: self taskGoalT42
    taking: ( Duration minutes: 4 )
    identifiedBy: ( TaskID labeled: $T numbered: 4 withLevels: {2} )
    orderedAt: 7.

GauchoExperiment2010Builder>>taskGoalChoicesT42
  | first second third |
  first := TaskGoalChoice
    labeled: '1'
    describedBy: 'Lumiere−Morphic, Lumiere−SceneGraph'
    requiringInputsLabeled: #().
  second := TaskGoalChoice
    labeled: '2'
    describedBy: 'Lumiere−Morphic, Lumiere−Modeling'
    requiringInputsLabeled: #().
  third := TaskGoalChoice
    labeled: '3'
    describedBy: 'Lumiere−ViewingVolumes, Lumiere−SceneGraph'
    requiringInputsLabeled: #().
  ^ { first. second. third }.
```

Bibliography

[1] D. E. Knuth, *Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd Edition)*. Addison-Wesley Professional, 1997.

[2] D. E. Knuth, "Computer programming as an art," *Commun. ACM*, vol. 17, Dec. 1974.

[3] E. W. Dijkstra, "The humble programmer," *Commun. ACM*, vol. 15, pp. 859–866, 1972.

[4] D. H. Ingalls, "Design principles behind smalltalk," *BYTE Magazine*, no. 8, 1981.

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

[6] M. Hiltzik, *Dealers of Lightning: Xerox Parc and the Dawn of the Computer Age*. HarperCollins Publishers, Apr. 1999.

[7] A. C. Kay, "The early history of smalltalk," in *The second ACM SIGPLAN conference on History of programming languages*, pp. 69–95, ACM, 1993.

[8] I. E. Sutherland, "Sketch pad a man-machine graphical communication system," in *Proceedings of the SHARE design automation workshop*, DAC '64, pp. 6.329–6.346, ACM, 1964.

[9] T. A. Corbi, "Program understanding: challenge for the 1990's," *IBM Syst. J.*, vol. 28, pp. 294–306, June 1989.

[10] J. Sillito, G. C. Murphy, and K. D. Volder, "Questions programmers ask during software evolution tasks," in *Proceedings of FSE-14 (14th International Symposium on Foundations of Software Engineering)*, pp. 23–34, ACM Press, 2006.

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

[12] J. Johnson, T. L. Roberts, W. Verplank, D. C. Smith, C. H. Irby, M. Beard, and K. Mackey, "The xerox star: A retrospective," *Computer*, vol. 22, pp. 11–26, 28–29, Sept. 1989.

[13] G. C. Murphy, M. Kersten, and L. Findlater, "How are java software developers using the eclipse ide?," *IEEE Softw.*, vol. 23, pp. 76–83, July 2006.

[14] 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 *Proceedings of ICSE 2005 (27th ACM/IEEE International Conference on Software engineering)*, pp. 126–135, ACM, 2005.

[15] M. Kersten and G. C. Murphy, "Mylar: a degree-of-interest model for ides," in *Proceedings of the 4th international conference on Aspect-oriented software development*, AOSD '05, pp. 159–168, ACM, 2005.

[16] J. Raskin, *The Humane Interface - New Directions for Designing Interactive Systems*. Addison-Wesley, 2000.

[17] 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*, pp. 455–464, ACM, 2010.

[18] J. K. Ousterhout, "Corner stitching: a data structuring technique for vlsi layout tools," Tech. Rep. UCB/CSD-83-114, EECS Department, University of California, Berkeley, Dec 1982.

[19] D. Roethlisberger, O. Nierstrasz, and S. Ducasse, "Autumn leaves: Curing the window plague in ides," in *Reverse Engineering, 332009. WCRE '09. 16th Working Conference on*, pp. 237 –246, 2009.

[20] R. DeLine, M. Czerwinski, B. Meyers, G. Venolia, S. Drucker, and G. Robertson, "Code thumbnails: Using spatial memory to navigate source code," in *Proceedings of the Visual Languages and Human-Centric Computing*, VLHCC '06, pp. 11–18, IEEE Computer Society, 2006.

[21] F. Brooks, *The Mythical Man-Month*. Addison-Wesley, 2nd ed., 1995.

[22] T. Ball and S. Eick, "Software visualization in the large," *Computer*, vol. 29, no. 4, pp. 33–43, 1996.

[23] M. Lanza, *Object-Oriented Reverse Engineering — Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Berne, May 2003.

[24] R. DeLine and K. Rowan, "Code canvas: zooming towards better development environments," in *Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering) - Volume 2*, pp. 207–210, ACM, 2010.

[25] A. Cockburn and M. Smith, "Hidden messages: Evaluating the efficiency of code elision in program navigation," *Interacting with Computers: The Interdisciplinary Journal of Human-Computer Interaction*, vol. 15, pp. 387–407, 2003.

[26] W. Lidwell, K. Holden, and J. Butler, *Universal Principles of Design*. Rockport, 2003.

[27] S. Ducasse and M. Lanza, "The class blueprint: Visually supporting the understanding of classes," *Transactions on Software Engineering (TSE)*, vol. 31, pp. 75–90, Jan. 2005.

[28] A. Kuhn, P. Loretan, and O. Nierstrasz, "Consistent layout for thematic software maps," in *Proceedings of the 2008 15th Working Conference on Reverse Engineering*, WCRE '08, pp. 209–218, IEEE Computer Society, 2008.

[29] 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.

[30] F. Olivero, "Object focused environments as vehicles for object-oriented modeling," in *Proceedings of ECOOP '11 (25th European Conference on Object-Oriented Programming), Doctoral Symposium*, p. to be published, 2011.

[31] F. Olivero, M. Lanza, M. D'Ambros, and R. Robbes, "Gaucho: Programming $\longleftrightarrow$ modeling," in *Proceedings of ECOOP '11 (25th European Conference on Object-Oriented Programming), Demonstration*, 2011.

[32] F. Olivero, M. Lanza, M. D'Ambros, and R. Robbes, "Enabling program comprehension through a visual object-focused development environment," in *Proceedings of VL/HCC '11 (IEEE Symposium on Visual Languages and Human-Centric Computing)*, pp. 127–134, 2011.

[33] F. Olivero, M. Lanza, M. D'Ambros, and 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)*, p. to be published, 2012.

[34] F. Olivero, M. Lanza, and M. D'Ambros, "Ronda: A fine grained collaborative development environment," in *Proceedings of CDVE 2012 (9th International Conference on Cooperative Design, Visualization and Engineering)*, pp. 155–162, 2012.

[35] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Longman Publishing Co., Inc., May 1983.

[36] D. Roberts, J. Brant, and R. Johnson, "A refactoring tool for smalltalk," *Theor. Pract. Object Syst.*, vol. 3, pp. 253–263, Oct. 1997.

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

[38] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr., "Code bubbles: a working set-based interface for code understanding and maintenance," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pp. 2503–2512, ACM, 2010.

[39] J. Bézivin, "On the unification power of models," *Software and Systems Modeling*, vol. 4, pp. 171–188, 2005.

[40] R. Lemma, "Software modeling in essence," Master's thesis, University of Lugano, 2012.

[41] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 3. ed., 2003.

[42] B. Sharif and J. I. Maletic, "An empirical study on the comprehension of stereotyped uml class diagram layouts," in *in Proceedings of 17th IEEE Intl. Conf. on Program Comprehension (ICPC)*, pp. 268–272, 2009.

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

[44] E. Hutchins, J. Hollan, and D. Norman, "Direct manipulation interfaces," *Human-Computer Interaction*, vol. 1, pp. 311–338, 1985.

[45] D. H. H. Ingalls, "The smalltalk-76 programming system design and implementation," in *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '78, pp. 9–16, ACM, 1978.

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

[47] J. H. Maloney and R. B. Smith, "Directness and liveness in the morphic user interface construction environment," in *Proceedings of UIST 1995 (8th ACM symposium on User interface and software technology)*, pp. 21–28, ACM, 1995.

[48] J. Maloney, "An introduction to morphic: The squeak user interface framework," tech. rep., Walt Disney Imagineering, 2000.

[49] D. Ingalls, K. Palacz, S. Uhler, A. Taivalsaari, and T. Mikkonen, "The lively kernel a self-supporting system on a web page," in *Self-Sustaining Systems*, vol. 5146 of *Lecture Notes in Computer Science*, pp. 31–50, Springer Berlin Heidelberg, 2008.

[50] R. Pawson, "Naked objects," *PhD thesis*, 2004.

[51] J.-M. Favre, "Foundations of model (driven) (reverse) engineering: Models – episode i: Stories of the fidus papyrus and of the solarus," in *POST-PROCEEDINGS OF DAGSTHUL SEMINAR ON MODEL DRIVEN REVERSE ENGINEERING*, 2004.

[52] A. Kay, "Computer software," *Scientific American*, 9 1984.

[53] G. Booch, "Why don't developers draw diagrams?," in *SOFTVIS*, pp. 3–4, 2010.

[54] B. B. Bederson, "The promise of zoomable user interfaces," in *Proceedings of the 3rd International Symposium on Visual Information Communication*, VINCE '10, ACM, 2010.

[55] M. Shand, “Algorithms for corner stitched data-structures,” *Algorithmica*, vol. 2, pp. 61–80, 1987.

[56] O. Nierstrasz and T. Gîrba, “Lessons in software evolution learned by listening to smalltalk,” in *Proceedings of the 36th Conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM ’10, pp. 77–95, Springer-Verlag, 2010.

[57] P. Kruchten, “Architectural Blueprints—The "4+1" View Model of Software Architecture,” *IEEE Software*, vol. 12, pp. 42–50, Nov. 1995.

[58] K. Beck and C. Andres, *Extreme Programming Explained*. Addison-Wesley, 2nd ed., 2005.

[59] K. Jansen and G. Zhang, “On rectangle packing: maximizing benefits,” in *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA ’04, pp. 204–213, Society for Industrial and Applied Mathematics, 2004.

[60] G. P. Kurtenbach, A. J. Sellen, and W. A. S. Buxton, “An empirical evaluation of some articulatory and cognitive aspects of marking menus,” *Hum.-Comput. Interact.*, vol. 8, pp. 1–23, Mar. 1993.

[61] M. M. Burnett, *Visual Programming*. John Wiley & Sons, 1999.

[62] M. Petre, “Why looking isn’t always seeing: Readership skills and graphical programming,” *Communications of the ACM*, vol. 38, pp. 33–44, 1995.

[63] A. F. Blackwell, “Metacognitive theories of visual programming: What do we think we are doing?,” in *Proceedings of the 1996 IEEE Symposium on Visual Languages*, IEEE Computer Society, 1996.

[64] J. Edwards, “Coherent reaction,” in *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA ’09, pp. 925–932, ACM, 2009.

[65] B. Cornelissen, A. Zaidman, and A. van Deursen, “A controlled experiment for program comprehension through trace visualization,” *IEEE Transactions on Software Engineering*, vol. 99, 2010.

[66] J. Stasko, "An evaluation of space-filling information visualizations for depicting hierarchical structures," *Int. J. Hum.-Comput. Stud.*, vol. 53, pp. 663–694, Nov. 2000.

[67] A. Marcus, D. Comorski, and A. Sergeyev, "Supporting the evolution of a software visualization tool through usability studies," in *in Proceedings International Workshop on Program Comprehension*, pp. 307–316, Prentice Hall, 1997.

[68] R. Wettel, M. Lanza, and R. Robbes, "Software systems as cities: A controlled experiment," in *Proceedings of ICSE 2011 (33rd International Conference on Software Engineeering)*, pp. 551 – 560, ACM Press, 2011.

[69] A. J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," in *Proceedings of ICSE 2007 (29th ACM/IEEE International Conference on Software Engineering)*, pp. 344–353, IEEE Computer Society, 2007.

[70] R. Robbes and M. Lanza, "Spyware: A change-aware development toolset," in *Proceedings of ICSE 2008 (30th ACM/IEEE International Conference in Software Engineering)*, pp. 847–850, ACM Press, 2008.

[71] M. Steinberg, "What is the impact of static type systems on maintenance tasks? an empirical study of differences in debugging time using statically and dynamically typed languages," master Thesis, University of Duisburg-Essen, 2011.

[72] P. M. Johnson, H. Kou, J. Agustin, C. Chan, C. A. Moore, J. Miglani, S. Zhen, and W. E. J. Doane, "Beyond the personal software process: Metrics collection and analysis for the differently disciplined," in *Proceedings of ICSE 2003*, pp. 641–646, 2003.

[73] M. Kersten and G. Murphy, "Using task context to improve programmer productivity," in *Proceedings of FSE 2006 (16th SIGSOFT Symposium on the Foundations of Software Engineering)*, pp. 1–11, ACM Press, 2006.

[74] E. Chikofsky and J. Cross, "Reverse engineering and design recovery: A taxonomy," *IEEE Software*, vol. 7, pp. 13–17, Jan. 1990.

[75] F. Olivero, M. Lanza, and R. Robbes, "Lumiére: A novel framework for rendering 3d graphics in smalltalk," in *Proceedings of IWST 2009 (1st International Workshop on Smalltalk Technologies)*, pp. 20–28, ACM Press, 2009.

[76] T. D. Marco, *Peopleware - Productive Projects and Teams*. Dorset House, 1999.

[77] A. Sarma, *Palantir: enhancing configuration management systems with workspace awareness to detect and resolve emerging conflicts*. PhD thesis, CalState University, 2008.

[78] L. Hattori and M. Lanza, "Syde: A tool for collaborative software development," in *Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering)*, pp. 235–238, 2010.

[79] E. Goffman and A. J. Wootton, *Exploring the interaction order*. Polity Press, 1988.

[80] R. B. Smith, M. Wolczko, and D. Ungar, "From kansas to oz: collaborative debugging when a shared world breaks," *Commun. ACM*, vol. 40, 1997.

[81] H. Lieberman and C. Fry, "Bridging the gulf between code and behavior in programming," in *CHI*, pp. 480–486, ACM/Addison-Wesley, 1995.

[82] C. R. B. de Souza, D. Redmiles, and P. Dourish, "Breaking the code, moving between private and public work in collaborative software development," in *Proceedings of GROUP 2003 (International ACM SIGGROUP Conference on Supporting Group Work)*, pp. 105–114, ACM Press, 2003.

[83] D. A. Smith, A. Kay, A. Raab, and D. P. Reed, "Croquet - a collaboration system architecture," IEEE Computer Society, 2003.

[84] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[85] R. Wettel and M. Lanza, "Visually localizing design problems with disharmony maps," in *Proceedings of Softvis 2008 (4th ACM International Symposium on Software Visualization)*, pp. 155–164, ACM Press, 2008.

[86] C. G. Myers and E. L. A. Baniassad, "Silhouette: visual language for meaningful shape," in *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA* (S. Arora and G. T. Leavens, eds.), pp. 917–924, ACM, 2009.

[87] A. Noack and C. Lewerentz, “A space of layout styles for hierarchical graph models of software systems,” in *Proceedings of the 2005 ACM symposium on Software visualization*, SoftVis ’05, pp. 155–164, ACM, 2005.

[88] S. Cooper, W. Dann, and R. Pausch, “Teaching objects-first in introductory computer science,” in *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, SIGCSE ’03, pp. 191–195, ACM, 2003.

[89] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg, “The BlueJ system and its pedagogy,” *Journal of Computer Science Education*, vol. 13, Dec. 2003.

[90] M. Kolling, “Teaching object orientation with the Blue environment,” *Journal of Object-Oriented Programming*, vol. 12, pp. 14–23, December 1999.

[91] C. Griggio, G. Leiva, G. Polito, N. Passerini, and G. Decuzzi, “A programming environment supporting a prototype-based introduction to oop,” pp. 45–50, ACM Press, 2011.