ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/259118316

TAUPE: Visualizing and analyzing eye-tracking data

Article *in* Science of Computer Programming · January 2014

DOI: 10.1016/j.scico.2012.01.004

CITATIONS
31

READS
600

6 authors, including:

Zohreh Sharafi
University of Michigan
17 PUBLICATIONS **240** CITATIONS
SEE PROFILE

Yann-Gaël Guéhéneuc
Concordia University Montreal
309 PUBLICATIONS **7,070** CITATIONS
SEE PROFILE

Giuliano Antoniol
Polytechnique Montréal
328 PUBLICATIONS **10,116** CITATIONS
SEE PROFILE

Naji Habra
University of Namur
61 PUBLICATIONS **634** CITATIONS
SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Lattice-based software re-engineering View project

Software Quality and Security View project

All content following this page was uploaded by Yann-Gaël Guéhéneuc on 23 November 2018.

The user has requested enhancement of the downloaded file.

Science of Computer Programming 79 (2014) 260–278

Contents lists available at ScienceDirect

Science of Computer Programming

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

TAUPE: Visualizing and analyzing eye-tracking data☆

Benoît De Smet [a], Lorent Lempereur [a], Zohreh Sharafi [b], Yann-Gaël Guéhéneuc [b,*], Giuliano Antoniol [c], Naji Habra [a]

[a] Research Center in Information Systems Engineering, FUNDP, Namur, Belgium
[b] Ptidej Team, École Polytechnique de Montréal, Québec, Canada
[c] Soccer Lab., École Polytechnique de Montréal, Québec, Canada

ARTICLE INFO

Article history:
Received 12 February 2011
Received in revised form 19 January 2012
Accepted 25 January 2012
Available online 11 February 2012

Keywords:
Eye-tracking
Visualization
Analysis
Compatibility
Extensibility

ABSTRACT

Program comprehension is an essential part of any maintenance activity. It allows developers to build mental models of the program before undertaking any change. It has been studied by the research community for many years with the aim to devise models and tools to understand and ease this activity. Recently, researchers have introduced the use of eye-tracking devices to gather and analyze data about the developers' cognitive processes during program comprehension. However, eye-tracking devices are not completely reliable and, thus, recorded data sometimes must be processed, filtered, or corrected. Moreover, the analysis software tools packaged with eye-tracking devices are not open-source and do not always provide extension points to seamlessly integrate new sophisticated analyses. Consequently, we develop the TAUPE software system to help researchers visualize, analyze, and edit the data recorded by eye-tracking devices. The two main objectives of TAUPE are compatibility and extensibility so that researchers can easily: (1) apply the system on any eye-tracking data and (2) extend the system with their own analyses. To meet our objectives, we base the development of TAUPE: (1) on well-known good practices, such as design patterns and a plug-in architecture using reflection, (2) on a thorough documentation, validation, and verification process, and (3) on lessons learned from existing analysis software systems. This paper describes the context of development of TAUPE, the architectural and design choices made during its development, and its documentation, validation and verification process. It also illustrates the application of TAUPE in three experiments on the use of design patterns by developers during program comprehension.

© 2012 Elsevier B.V. All rights reserved.

1. Introduction

The software life-cycle [51] is traditionally divided into several macro phases: from inception to implementation, maintenance, and retirement. The most expensive phase of any software project is maintenance [6]. A software life-cycle often spans decades [33] and thus maintenance is rarely performed by the developers that first developed the program. Therefore, maintainers must first understand the program before implementing any change. During program

☆ Source code, examples, and documentation are available at http://www.ptidej.net/research/taupe/. This work has been partially funded by the Guéhéneuc's Canada Research Chair on Software Patterns and Patterns of Software and NSERC Discovery Grant.

* Corresponding author.
E-mail address: yann-gael.gueheneuc@polymtl.ca (Y.-G. Guéhéneuc).

0167-6423/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2012.01.004

comprehension activities, developers build mental models of the program, which should allow them to perform changes without introducing bugs [50] or degrading the program design [38].

The activity of program comprehension has been studied by many researchers, for example to devise models of program comprehension, *e.g.*, [7,61,54], or tools to ease this activity, *e.g.*, [3]. Recently, researchers have introduced the use of eye-tracking devices to further improve our understanding of the developers' cognitive processes taking place during program comprehension [20,21,46,48,63] by studying their use of visualized data, such as source code and identifiers [47], diagrams [55], and others. In particular, we proposed a theory [21] to unify previous theories of program comprehension with current theories in vision science [37]. The proposed theory aims to model and explain in detail the developers' process of acquiring the necessary data using their vision to understand a program.

Eye-tracking devices have been used for nearly 30 years in cognitive science for the study of human–computer interfaces, for marketing, medical research, and so on. An eye-tracker records the coordinates of a subject's gaze when looking at a computer screen. It provides a new perspective on a subject's comprehension process because it shows the areas attracting the subject's attention as well as the visual path of her gaze on the screen [11]. The subject's attention and visual path together form a window on her cognitive processes [43]. Thus, analyzing the data recorded using an eye-tracking device allows understanding in detail a subject's process of acquiring data, for example during program comprehension.

However, there are still some major obstacles to the widespread adoption of eye-tracking devices. Besides the costs of such devices, the accompanying analysis software systems are not open-source and often not extensible, preventing the development and seamless integration of new sophisticated analyses [27]. Consequently, we undertook the development of the Taupe system[1] (Thoroughly Analyzing the Understanding of Programs through Eyesight) to visualize, analyze, and edit eye-tracking data. The two main objectives of Taupe are compatibility and extensibility so that researchers can easily: (1) apply the system on any eye-tracking data and (2) extend the system with their own analyses. To meet our objectives, we base the development of Taupe: (1) on well-known good practices, such as design patterns and a plug-in architecture using reflection, (2) on a thorough documentation, validation, and verification process, and (3) on lessons learned from existing analysis software systems.

In Section 2, this paper describes the context of development of Taupe. In Section 3, we discuss related works. In Section 4, it details the architectural and design choices made during its development, and its documentation, validation, and verification process. In Section 5, it illustrates the application of Taupe in three experiments on the use of design patterns by developers during program comprehension. Finally, in Section 6, it summarizes the contributions of Taupe and concludes with future work.

2. Context

In this section, we first introduce definitions needed to understand the rest of the paper. We then discuss previous work on program comprehension and on using eye-tracking data. Finally, we describe two eye-trackers to show their usage and describe the recorded data that they provide.

2.1. Definitions

In vision science, a *fixation* is the position of the eye during a gaze and a *saccade* is a movement of the eye between two fixations [37]. In the field of empirical software engineering [24] and in particular in this paper, an *experiment* is a set of subjects who answer a set of questions (*e.g.*, comprehension questions) using some visual data (*e.g.*, a UML class diagram).

An *area of interest* (AOI) is an area of some visual data with a specified relevance. An area can thus be *relevant* to the subject to answer a question, *irrelevant*, or *ignorable*. The fixations in an ignorable area of interest should not be taken into account in any subsequent analyses.

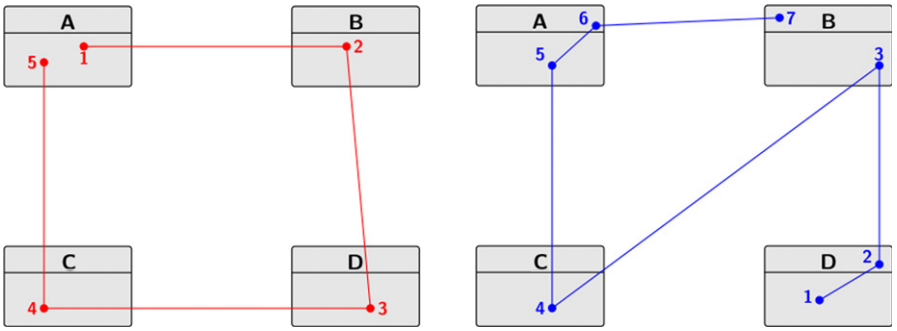
For example, if we are interested to observe how subjects read a class diagram and relate relevant parts to answer a question that is displayed on top of the diagram, the area in which the question is written does not provide any interesting data about the subjects' cognitive process. We can consider this area as an *ignorable area of interest*.

A *visual path* is a series of visited areas of interest, sorted by chronological order. An area is visited if there is a fixation in it. The same area of interest that is visited twice consecutively is considered only once. For example, Fig. 1 shows two distinct visual paths on a same diagram, with four areas of interest: `A`, `B`, `C`, and `D`. One visual path, on the sub-figure of the left, is `ABDCA` while the other, on the sub-figure of the right, is `DBCAB`.

2.2. State of the art

The Taupe system is used to edit, visualize, and analyze data that are gathered using eye-trackers during empirical studies on program comprehension activities. We summarize the theories in vision science, report some works on empirical studies on software engineering, and detail studies in program comprehension, emphasizing on studies of the developers' use of UML class diagrams, which form the main bulk of the eye-tracking studies in software engineering.

[1] "Taupe" means "mole" in French and is pronounced 'tOp.



(a) ABDCA. (b) DBCAB.

Fig. 1. Two distinct visual paths.

2.2.1. Theories in vision science

We summarize theories in vision science to explain the theoretical background behind usability of eye-tracking systems. Vision science is an interdisciplinary domain of cognitive science interested in the understanding of people's vision system. Vision science collects facts on vision, formulates laws from these facts, and devises theories explaining these laws and facts. With these theories, vision scientists have been able to predict new facts successfully and to refine their theories.

Vision science possesses many theories to explain color vision, spatial vision, perception of motion and events, as well as eye movements, visual memory, and visual awareness. To the best of our knowledge, Palmer's book presents a complete and in-depth coverage of vision theories, cast in the information processing paradigm [37].

2.2.2. Theories in program comprehension

The domain of software engineering possesses theories, which are an invaluable help in setting up experiments to observe facts (dis)proving laws and theories [13]. A theory helps in understanding a domain by explaining "Why is it so?" questions [13, p. 7] [36].

Empirical studies are essential to understand phenomena with which software engineering deals, thanks to the work of precursors, such as Basili [5]. Empirical studies are based on the classical cycle: observations (facts), laws, theories. Many facts have been recorded through empirical studies and some laws have been proposed [33].

Few theories of program comprehension have been proposed in the literature. One of the first theory of program comprehension, proposed by Brooks [8], describes program comprehension as a process of building a sequence of knowledge domains, bridging the gap between problem domain and program execution. A succession of knowledge domains describes a software engineer's comprehension of a program.

Another theory, developed by von Mayrhauser [34], is an integrated theory describing the processes taking place in a software engineer's mind during program comprehension, as a combination of top-down and bottom-up comprehension processes, working with a common knowledge base. This integrated theory accounts for the dynamics of forming and of abstracting a mental representation of a program.

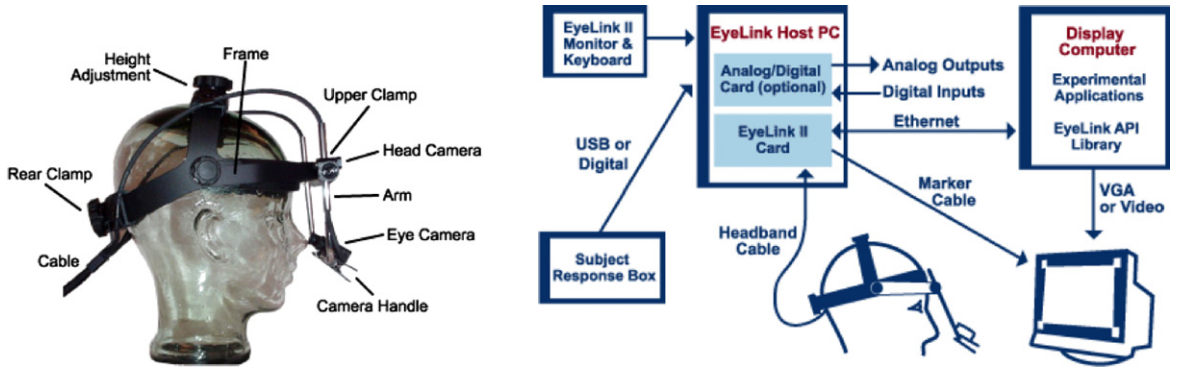
Existing theories of program comprehension describe different processes deployed by software engineers to understand a program and to analyze available information. However, existing theories do not explain well how software engineers acquire this information. Guéhéneuc [21] proposed a theoretical framework by joining vision science and program comprehension. This framework could help explaining the processes of program comprehension, describing known facts, extracting new facts, and designing new experiments in program comprehension.

2.2.3. Studies of program comprehension

The rich literature on program comprehension focused on the problems of obtaining data from software artifacts (static and dynamic data, features, documentation, and other repositories), see for example [1,52]. It also tackles the means to represent and to communicate this data, using various techniques from text-based editors to 3D interactive dynamic environments, such as [49]. This literature is essential to understand what kind of data software engineers have at their disposal to understand programs.

Some works also studied the contexts in which program comprehension takes place. Murphy et al. [35] distinguished, described, and identified recurring patterns in software engineers' daily activities. Although not related to program comprehension explicitly, their work brings insight in the program comprehension activity, because this activity is part of all but the most basic software engineering activities. Thus, this line of research is important to generalize claims in program comprehension.

All software engineers use diagrams as means to convey information to other developers or to better understand programs. Diagrams reduce comprehension and learning effort by omitting irrelevant details and highlighting pertinent information. The closer the information presented on diagrams is to a developer's mental representation, the easier it is to understand [9].



(a) Headband. (b) Setting.

Fig. 2. Eye-link II [53].

Class diagrams have been extensively studied in the literature on program comprehension. They represent the structure and global behavior [26] of programs, showing classes, interfaces, and their relationships. They are often used by software engineers during development and maintenance to abstract implementation details and to present an easier-to-grasp clustered view of the program source code [19,42].

2.3. Devices

There are two main classes of eye-tracking devices: intrusive ones, which requires subjects to wear some gears and non-intrusive ones. We now present three eye-tracking devices (one intrusive device and two non-intrusive devices) and the data visualization software that they use for visualizing and exporting their data.

2.3.1. Intrusive eye-tracking device

1. Eye-link II: this system consists of three miniature cameras mounted on a padded headband (see Fig. 2a). Two eye cameras are used to track the eyes' movements. An optical head-tracking camera integrated into the headband allows an accurate tracking of the subject's point of gaze.
   The Eye-link II has the highest resolution (noise limited at $< 0.01°$) and fastest data rate (500 samples per second) of any head mounted eye-tracker.
   Fig. 2b from *SR-Research*[2] [53] describes the configuration of this eye-tracking system, which uses two separate computers: the *Host PC* and the *Display PC*. The *Host PC* performs real-time eye-tracking at 250 or 500 samples per second and also computes the subject's true gaze position on the display. The *Desktop PC* provides displays for experiment and calibrating targets during eye-tracker calibration.
   The major problem of this system is its instability. If the subject moves too far away from the calibration point, the calibration step must be done again. An offset (a difference between the real place of the fixation and the fixation recorded by the eye-tracker device) is often present.
   The Eye-link II system comes with the *Eye-link II Windows Developer Kit* that provides sample display programs, C source code, and instructions for creating experimental programs. It uses `Data Viewer`[3] to display eye-fixation and visualization path on top of the presented stimulus. It also provides the output files that are stored in the *Host PC*. One output file contains eye fixations, saccades and the information related to the interest area. The `Data Viewer` is neither *open-source* nor *free* and cannot be easily extended with new analyses.

2.3.2. Non-intrusive eye-tracking device

1. *FaceLAB* from *Seeing Machine*[4]: it is a more recent eye-tracking device than the Eye-link II system. It consists of one computer (either a desktop PC or a laptop) and two miniature cameras, as shown in Fig. 3. It tracks first the position of the subject's head using her eye-brows, nose, and lips and, then, uses this data to track the subject's gaze. It works in pair with *GazeTracker* from *EyeResponse*,[5] which is a more recent [15] data visualization software than the `Eye-link II Data Viewer` but which is also neither *open source* nor *free*. Multiple eye-trackers can interact with one another: such a configuration is used in cockpits or in cars to track the eyes of the subject even when she turns her head. It is also possible

2 http://www.sr-research.com/accessories_ELII_dv.html.
3 http://www.sr-research.com/accessories_EL1000_dv.html.
4 http://www.seeingmachines.com/.
5 http://www.eyeresponse.com/.

Fig. 3. The FaceLAB eye-tracker [45].

to run *FaceLAB* and GazeTracker on a single computer with a *scene camera* that records a video of the whole scene. *FaceLAB* interacts with the eye-tracking device and transmits the data to *GazeTracker* via network. *GazeTracker* simply saves the data associated with the displayed image. *GazeTracker* provides a more advanced interface and more visualization tools than `Eye-link II Data Viewer` but it cannot be readily extended with new analyses. A screen cast of the use of *FaceLAB* and *GazeTracker* is available on-line.[6]

2. *ITU Gaze Tracker*: it is a non-commercial eye-tracking device by the university of Copenhagen(ITU).[7] The *ITU Gaze Tracker* is an open-source, low-cost software that allows the use of a web-cam as an eye-tracker [44].

TAUPE can import and analyze eye-tracking data provided by Eye-link II's `Data Viewer` and *FaceLAB*'s *GazeTracker* from *Seeing Machine*.

3. Related work

We only recall here some of the main lines of research on program comprehension using class diagrams.

Purchase et al. [10] reported the results of experiments on the effect of aesthetics criteria on the preferences of users of UML class diagrams. They performed several experiments with subjects to assess the subjects' preferences over several pairs of class diagrams, each diagram in a pair conforming to different (but related) aesthetics criteria. They collected quantitative data in the form of percentages of subjects preferring one diagram over another in each pair and qualitative assessments of each diagram. They reported the most important aesthetic criteria for UML class diagrams: joined inheritance arcs or directional indicators.

Eichelberger [12] studied the relation between the UML notation for class diagrams, principles of human–computer interactions, and principles of object-oriented design and programming. The author then suggested changes to the UML notation and aesthetics criteria to lay out class diagrams. These changes and aesthetic criteria are implemented in a tool, SUGIBIB, to lay out UML-like class diagrams. The author claimed that laying out class diagrams conforming to aesthetics criteria improve the readability of the diagrams but only qualitative arguments were provided.

Hadar and Hazzan [23] presented results from a study on the strategies applied by software engineers in the process of comprehending visual models of programs. They use visual models that were described using the UML notation and included use case, activity, class, sequence, collaboration, state chart, object, package, and deployment diagrams. The subjects were senior students majoring in computer science from several universities. The subjects were divided in two groups to collect both qualitative and quantitative data. The authors concluded on the usefulness of multifaceted descriptions of programs provided by the UML and that no one type of diagram was more important than the other. However, further studies should be performed to confirm these findings.

Sun and Wong [56] evaluated the layout algorithms for class diagrams of two industrial tools, Rational Rose and Borland Together, according to criteria from previous work, including the cited works by Purchase [10] and Eichelberger [12]. Using laws from the Gestalt theory of visual perception [37, p. 50–53], they retained and justified 14 criteria to assess the visual quality of the layouts of class diagrams. They applied these criteria on a Thermometer program and on JUNIT [17]. They concluded on the good quality of both industrial tools, on the relevance of their criteria, and on the difficulty of satisfying all criteria.

Guéhéneuc conducted an experiment with eye-trackers to study how software engineers acquire and use information from UML class diagrams [20]. He concluded on the importance of classes and interfaces and reported that developers seem to barely use binary class relationships, such as inheritance or composition. Yusuf et al. [63] conducted a similar study to analyze the utilization of specific characteristics of UML class diagrams (*e.g.*, layout, color, and stereotypes) during program comprehension. They concluded on the efficiency of layouts with additional information as colors or stereotypes to improve program comprehension.

Sharif et al. [46] performed a controlled experiment with eye-trackers to assess the effect of different layouts on the comprehension of UML class diagrams. They reported that the multi-cluster layout obtains higher level of accuracy and

6 http://www.ptidej.net/research/taupe/videos/.

7 http://www.gazegroup.org/downloads/23-gazetracker.

takes less time than the orthogonal layout. In another work, Sharif et al. [48] also conducted an eye-tracking experiment to investigate the impact of layout on the comprehension of four design patterns (Strategy, Observer, Composite, and Singleton) in UML class diagrams. They reported the positive impact of multi-cluster layout on speed for all four design patterns. They also concluded that the multi-layer layout has positive impact on accuracy and visual effort for Strategy and Observer design patterns.

Bednarik and Tukiainen [2] proposed an approach to study trends on repeated measures of sparse data over a small data set of program comprehension activities captured with eye-trackers. Using this approach, they characterized program comprehension strategies using different program representations (code lecture and program execution). Sharif et al. [47] similarly studied whether the Camel Case convention of creating identifiers was more efficient than using underscores. They replicated a previous study by Binkley et al. [4] that showed that the Camel Case convention leads to a higher accuracy in reading. Interestingly, they found opposite results: although the data indicated no difference in accuracy between the Camel Case and underscore conventions, subjects recognized identifiers in the underscores style more efficiently.

Uwano et al. [59] studied the code-reading habits of developers and identified typical "patterns" that distinguishes "efficient" readers from others. They first implemented an integrated environment to measure and record code reviewers' eye movements and, based on their fixations, identify the lines of the source code that the reviewers are reading. They then conducted an empirical study of 30 review processes of six programs by five reviewers. They reported that all reviewer "scanned" the source code following a similar pattern and that reviewers who did not spend enough time during the "scan" tended to take more time for finding defects in the code.

4. Taupe

The Taupe system is a software program designed by the *Ptidej Team*[8] to import data from eye-trackers and to enable the execution of various algorithms on the imported data. Its first version has been developed since 2005 with the contribution of many students. Its current version, v2.0, was developed by the first three authors by reusing some code from the first version but revising entirely the program architecture and design. The users' and developers' guides[9] provide more details about versions of Taupe.

4.1. Motivation

Taupe was originally designed to compare the ways subjects' read and understand UML class diagrams. A subject was asked different questions about her understanding of some diagrams when performing some maintenance tasks. The subjects' eye movements were recorded using an eye-tracking device and then analyzed using Taupe.

Taupe is an open-source and free software system that can parse, process, and analyze eye-tracker's data stored in the output files provided by eye-trackers software systems. Taupe not only visualizes eye fixations and the areas of interest on top of the stimulus but also provides a set of algorithms to analyze the data.

Taupe is compatible so it can handle eye-tracking data from different devices. The first version of Taupe worked with the Eye-link II `Data Viewer`'s output file. Then, we extended Taupe to support GazeTracker's data. Unlike GazeTracker and Eye-link II `Data Viewer`, Taupe is not specific to any eye-tracking system. Moreover, Taupe is extensible and therefore it can be extended with new parsers and analyses for eye-tracking data.

Proprietary systems such as `Eye-link II Data Viewer` and GazeTracker are provided by the manufacturers of the corresponding eye-tracking devices and packaged with their eye-trackers. These systems are neither open-source nor free and they only analyze the eye-tracking data that are provided by their own integrated eye-tracking devices, offering a limited, not extensible set of analyses.

4.2. Architecture

Fig. 4 shows the architecture of Taupe v2.0. The core of Taupe consists of the data collected about an experiment: fixations, saccades, questions, and their answers. Some of this data is provided by eye-tracking devices. Other data is collected by the experimenters through questionnaires or other means.

The data is organized depending on the answers to the questions. The questions are represented by a set of images and their corresponding area of interest. Each file that contains the areas of interest related to a question can be manually written and Taupe also allows the user to create this kind of files using a graphical user interface. Different types of parsers are used to extract the information from the files provided by the eye-tracking devices.

The possibility to link a subject to other subjects is also available in Taupe, through the concept of *group*. A group is a set of subjects who have an attribute in common. For example, such an attribute could be the level of study (B.Sc., M.Sc., Ph.D., and so on), their gender (male or female), their UML knowledge (low, average, high, and so on). All these variables can be measured by some external means (questionnaires, interviews, and so on).

[8] http://www.ptidej.net/.

[9] http://www.ptidej.net/research/taupe/downloads/.

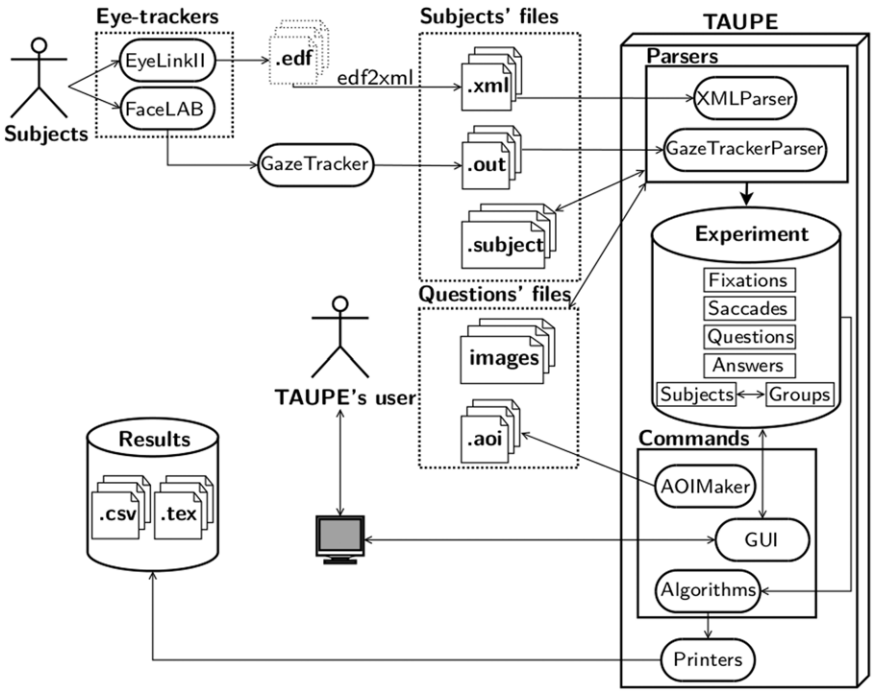


Fig. 4. Architecture.

TAUPE handles one experiment at a time. The system's main features are implemented as *commands*. For example, the *AOI Maker* command helps to graphically designate and create areas of interest for a given image. Other commands provide other essential features of the system, for example the set of algorithms that produce analysis results. These results may be written in a set of files using some *printers* selected by the user and they can be subsequently studied or analyzed using any data mining software. TAUPE also implements a graphical user interface to visualize data and their characteristics.

4.3. Inspirations

The development of TAUPE has been inspired by many sources, such as our experiences from our previous experiments using different eye-tracker devices. In this section, we explain the necessary requirements of the system that has been developed to visualize and analyze eye-tracking data. In addition, we explain how TAUPE satisfies these requirements.

4.3.1. From the eye-trackers

The three main features of TAUPE result from our past uses of eye-tracker devices and their analysis software systems. First, TAUPE must implement some parsers to use the data from multiple eye-tracking devices. Second, TAUPE must graphically display the fixations, saccades, and visual path over a diagram. Third, TAUPE must allow its users to correct the data recorded by the devices using some offsets, as shown in Fig. 5, which is an example of a diagram with a clearly visible static offset. Every cloud of points must be moved to the same direction and the same distance to match with the diagram.

Three types of offset were encountered during our previous experiments: *static* offsets, *non-static* offsets, and *chaotic* offsets. *Static* offset can be easily seen and a constant translation can be applied to all the fixations to fix the offset. *Non-static* offsets are less easily seen and the translations are different for each group of fixations according to their position on the screen. *Chaotic* offsets are random offsets due to vagaries from the eye-tracking devices and–or the subject and must be corrected manually for each fixation.

4.3.2. From the field of study

The field of study using eye-tracking devices led to several useful metrics to assess a subject's browsing effort, for example. Metrics based on fixations are the most common; for example, the total number of fixations is pointed out by Goldberg et al. [18]: "The number of fixations overall is thought to be negatively correlated with search efficiency". The number of fixations per areas of interest indicate that certain areas are more noticeable or more important than other areas [39]. According to Just et al. [31], the duration of the fixations on a specific area can have two different meanings:

1. The subject has a hard time to extract the information [16].
2. The subject is "more engag[ed] in some way" by the object [40].

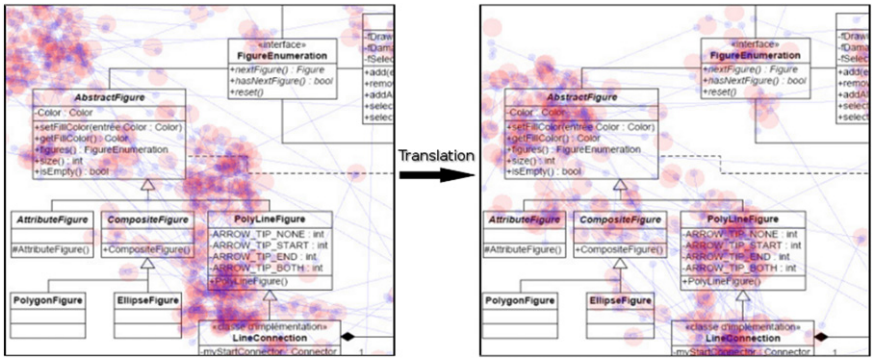


Fig. 5. An example of a static offset.

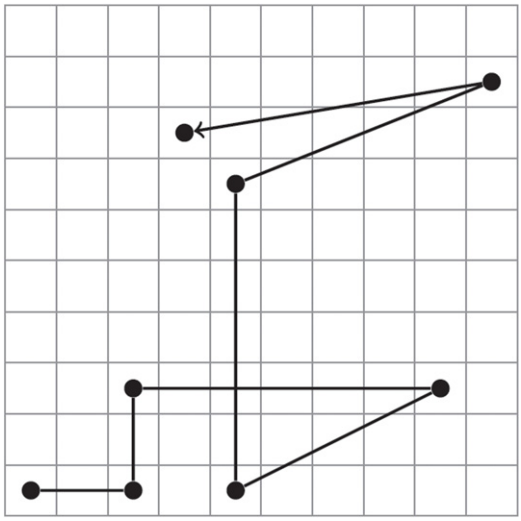


Fig. 6. Example of spatial density of 8%.

The *spatial density* of fixations is a widely used metric: the "coverage of an area" due to search and processing may be captured by the spatial distribution of gazepoint samples. [...] The area can be divided into grid areas either representing specific objects or physical screen area. [...] The *spatial density* index was equal to the number of cells containing at least one sample, divided by the total number of grid cells. [...] A smaller spatial density indicated more directed search, regardless of the temporal gazepoint sampling order" [18]. The spatial density of a visual path is presented as an example in Fig. 6. In this example, we have $10 \times 10 = 100$ cells while eight distinct grid cells are traversed visually so the spatial density is 8%. The *spatial density* is identified by the variable SD and it is formulated as follows:

$$SD = \frac{\sum_{i=1}^{n} c_i}{n}$$

where n is the number of fixation in the specific area (one cell) and c_i is equal to 1 if the area number i visited, otherwise it is equal to 0.

Some metrics use saccades. However, some eye-tracking devices do not provide the required raw data. For example, *FaceLAB* does not provide the *amplitude* of a subject's saccades. Yet, other saccade-based metrics are implemented in Taupe, for example using a *transitional matrix* and its *density*: "also known as link analysis, frequent transitions from one region of a display to another indicates inefficient scanning with extensive search. The transition matrix is a tabular representation of the number of transitions to and from each defined area" [18]. A cell is filled if a saccade starts or ends in its area. The *transitional matrix* is computed as follows:

$$TM = \frac{\sum_{i=1}^{n} \sum_{j=1}^{n} c_i, j}{n.n}$$

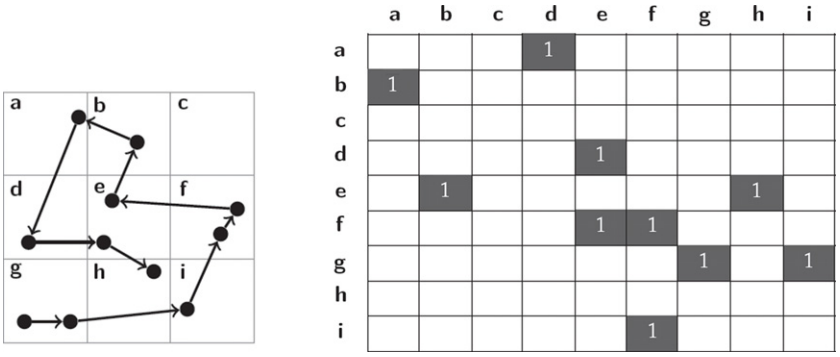


Fig. 7. Example of the visual path and the corresponding transition matrix.

where n is the number of fixation in the specific area (one cell) and $c_{i,j}$ is equal to 1, if there is a saccade from cell i to cell j, otherwise it is equal to 0.

The density of the *transitional matrix* can be computed as:

$$TRANSITION\_DENSITY = \frac{\sum_{x \in C} isFilled(x)}{\#C}$$

where C is the set of cells in the *transitional matrix* and $isFilled : C \rightarrow \{0, 1\}$ returns 1 if the specified cell is filled and 0 otherwise. Two visual paths can have the same *convex hull* and the same *spatial density* but different *transitional densities*.

Fig. 7 shows an example of a visual path on the display grid and table in the figure represents its transition matrix. A cell at position (x, y) that contains the value 1 means that a transition is from the cell x to the cell y.

4.3.3. From previous experiments

Taupe can compute several statistics about the time spent on each question for each group of subjects. These metrics do not come from the field of eye-tracking but are useful to compare the subjects' performance. Thus, it provides a command to obtain statistics about the fixations' duration, the subjects' time spent in a specified area of interest, and the transitional matrix. The whole set of metrics and algorithms implemented and available currently in Taupe are explained in its users' guide (see footnote 8).

As illustrated in Fig. 1, two subjects can have two different visual paths on the same diagram while answering the same questions. These two visual paths show that the two subjects understood the diagram differently. The difference between two *visual paths* is computed in Taupe using an *edit distance* algorithm, the *Levenshtein algorithm* [22], which compares two strings. Using Taupe, the user can also choose a specified percentage of kept fixations to generate these *visual paths*. A *merged fixation* is all the consecutive fixations that are in an area of interest without leaving this area (the duration of this resulting fixation is the sum of all fixations' durations). The kept percentage of fixations is related to the longer *merged fixations*.

Jeanmart [28] suggested the metric *Normalized Fixations per Area of Interest* that is the ratio between the normalized number of fixations in an area of relevant interest and the normalized number of fixations in an area of irrelevant interest. The $NORM\_RATE_i$ metric can be used to assess a subject's effort:

$$NORM\_RATE_i = \frac{\frac{\#FAORI_i}{\#AORI_j}}{\frac{\#FAOII_i}{\#AOII_j}}$$

where A_j is the set of answers related to the question j, $FAORI_i$ is the set of fixations contained in an area of relevant interest for the subject's answer i, $AORI_j$ is the set of areas of relevant interest in the question j, $FAOII_i$ is the set of fixations contained in an area of irrelevant interest for the subject's answer i, $AOII_j$ is the set of areas of irrelevant interest in the question j, and # returns the cardinality of a set.

4.4. Taupe *use*

Fig. 8 shows the main user interface of Taupe. The *AOIMaker* (Area of Interest Maker) command allows users to create a set of areas of interest for a specific image. The *Results* command executes a selected algorithm on eye-trackers' data while the *visualization Tool* command displays fixations, saccades, areas of interest, and visual paths.

4.4.1. Areas of interest

As mentioned in Section 4.2, each question is related to a file that contains its set of areas of interest. Although a user can create such a file using Taupe's *AOIMaker* command, shown in Fig. 9; they can also write such a file manually by specifying a list of coordinates for each AIO. They must respect the EBNF grammar [25] in Fig. 10.

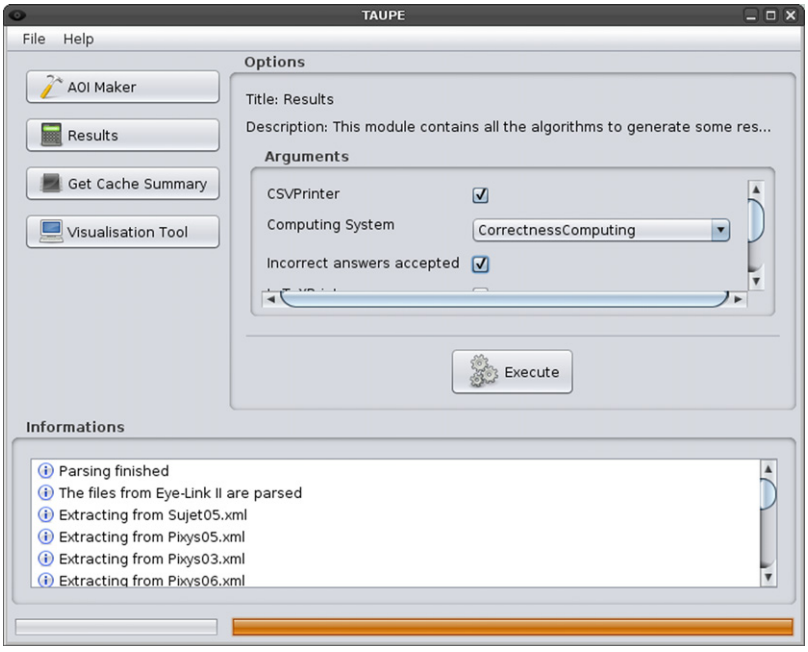


Fig. 8. The TAUPE main interface.

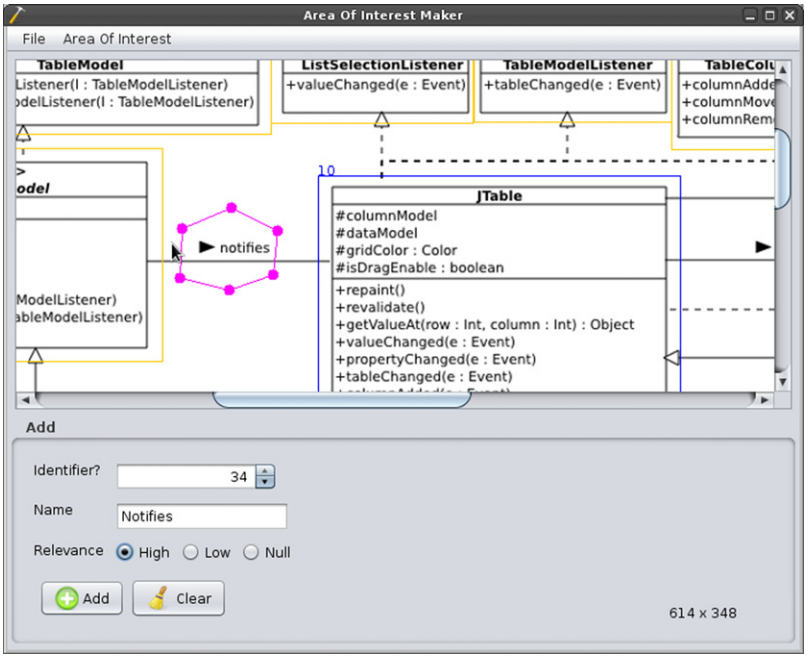


Fig. 9. Areas of interest maker.

```
1  [<id> <type> <name> <coordinate> <coordinate> <coordinate>+ <EOL>]*
2  <id> ::= integer
3  <type> ::= NULL | AOI | AORI
4  <coordinate> ::= "("integer "," integer")"
5  <EOL> ::= EndOfLine
```

Fig. 10. The EBNF grammar of the question file.

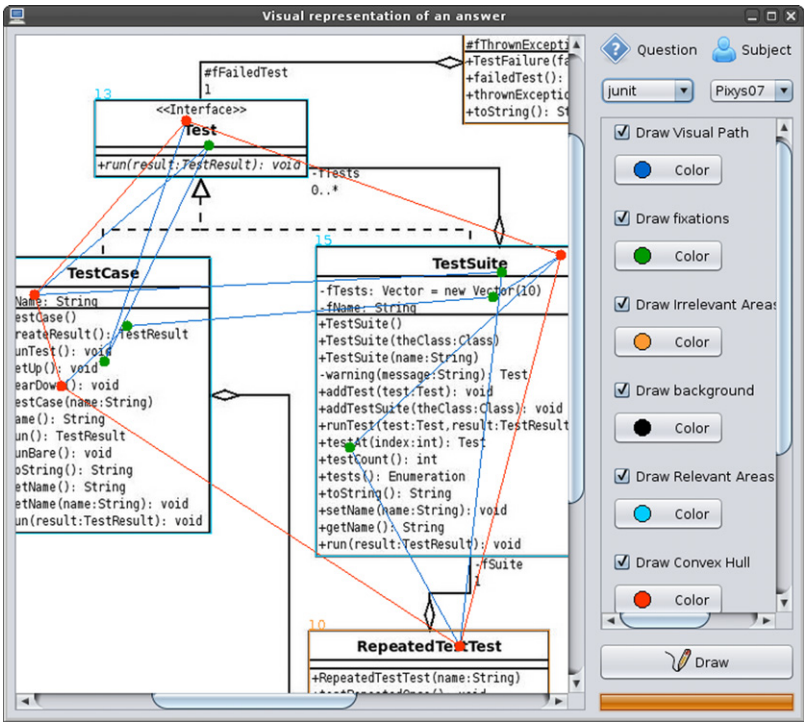


Fig. 11. The visualization tool.

4.4.2. *Visualization*

Although the other analysis software systems in the field of eye-tracking allow the users to visualize a set of data recorded by these devices, TAUPE brings extensibility in the visualization of the data and of the analyses performed on the data, for example, the visualization interface can display, in addition to fixations and saccades, the convex hull of a chosen percentage of fixations.

The convex hull of a set of fixations is the smallest polygon that includes all fixations. TAUPE generates convex hulls for a series of percentages on the range of [5% . . . 100%]. The percentage corresponds to the percentage of fixations that are considered by TAUPE per AOI. For example, 50% means that only half of the fixations (the longest ones) are considered per AOI. As shown in Fig. 11, a convex hull is drawn using orange lines that encompass the whole fixations that are available for this images.

4.4.3. *Input files*

An input file to TAUPE is simply a set of fixations and saccades listed in a text file. Different eye-tracking devices provide different input formats but with essentially the same content: a time (sometimes a *timestamp* in milliseconds), a type (fixation or saccade), a set of coordinates (x and y), and a duration. Other pieces of data, such as the subjects' pupil sizes or the saccades' peak velocity, can also appear in an input file.

Eye-link II: The *SR-Research*'s software system generates `.edf` files that TAUPE cannot read directly. A tool named `edf2xml` (provided by the same company) can be used to generate `.xml` files from the `.edf` files. Then, these `.xml` files can be loaded in TAUPE, which include the saccades' *peak velocity* and *amplitude*.

GazeTracker: The GazeTracker software system can generate `.out` files that can be loaded in TAUPE using the appropriate parser. A screencast on how to export data from GazeTracker is available on-line (see footnote 6).

4.5. TAUPE *development*

The main objectives of the TAUPE software system are compatibility and extensibility. During the development of its version 2.0, decisions were taken to satisfy these two objectives.

4.5.1. *Compatibility*

The compatibility of the TAUPE system with respect to the eye-tracking devices is achieved using an open architecture in which each core component of the system, including parsers, experiments, commands, and printers, as shown in Fig. 4,

can be specialized or new, such component can be added to the system. We achieve such an open architecture using several design patterns and Java reflection mechanism.

First, TAUPE implements the architectural pattern *Model View Controller* (MVC) in its implementation for its user interface, as described by Gamma et al. [14]. The MVC uses the *Observer* design pattern to facilitate the communication between the *model* and its *views*.

The *Composite* and *Visitor* design patterns are implemented to represent and visit the data in TAUPE: fixations, saccades, questions, and so on. Results from commands also implement the *Composite* and *Visitor* design patterns to allow hierarchical output and the combination of the outputs of several commands. The printers, which generate output files that contain the results, are a set of visitors, which define how to print each result (using some `visit` methods) while the results describe how to navigate their hierarchy (using some `accept` methods).

The TAUPE system handles one experiment at a time and, therefore, the `Experiment` class implements the *Singleton* design pattern. It also only returns *Iterator*s on lists and sets as a defensive measure against user code modifying unwillingly or maliciously the content of the experiment or of the results.

4.5.2. Extensibility

Extensibility is one of the two main objectives of the TAUPE system. It pertains to maintainability [58]: "the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment". Kienle et al. [32] also mentioned extensibility as on of the important requirements that a tool builder must consider. We achieve extensibility in TAUPE by following two complementary directions: one related to its implementation and the use of the Java reflection mechanism, another related to its documentation.

First, TAUPE makes extensive use of the Java reflection mechanism to increase its extensibility by allowing other developers to easily add their own parsers, commands, and printers through the implementation of the appropriate interfaces and the integration of their implementation in specific folders. New commands (such as new algorithms, new parsers …) can be added to TAUPE simply by extending the correct interface (or the correct abstract class) with a new class by putting it in the right package. TAUPE will dynamically load these classes without the need for further modifications in TAUPE's code. For example, to create a new group of subjects in the system, a software engineer only must create a new class that extends the abstract class `Group` and add this new class in the package `laigle.taupe.viewer.utils.group.data`. There are five abstract methods to be implemented to make the group effective:

[getName() : String] This method returns the name of the group according to its members' characteristics. For example, "Gender" would be the name for a group whose members are organized according to their gender.

[getPrefixName() : String] This methods returns the short name of the group and is used as an alternative when, for example, outputting groups in a CSV file.

[getAvailableValues() : Iterator<Object>] This method returns the possible values of the characteristics of the group members. Typically, the `GenderGroup` would have three available values: *female*, *male*, and *unknown*.

[newInstance(Object o) : Group] This method plays the role of constructor with an argument that represents the common characteristic's value of the group members.

[isEligible(s : SubjectData) : Boolean] This method checks whether or not a subject `s` can be added to the group according to its characteristics.

Second, the extensibility of TAUPE also depends on the ease of extending it by developers. Complete documentation is available online (see footnote 8): both a user's guide and a developer's guide are available. The developer's guide includes explanations to extend eight different kinds of core components of TAUPE. It was written with the objective to ensure that all future developers of TAUPE clearly understand how TAUPE works and to describe the means to add new and–or modify existing components. The system source code is extensively documented using the *Javadoc* mechanism, with more than 820 methods of the system described. Consequently, new components can be easily added to TAUPE by other researchers.

4.5.3. Validation and verification

The development of the TAUPE system has included explicit validation and verification phases. As the TAUPE system is used in scientific research, it was crucial that all of its results are correct. The validation process [58] for TAUPE led to the development of several test cases using the *JUnit framework*.[10] The package `laigle.taupe.tests` contains all of the current 50 test cases. These test cases use as oracles values computed by hand for the different implemented algorithms. For example, they include test cases for the computation of the convex hull of a set of fixations or the computation of the *edit distance* between two areas of interest.

The verification process [58] of TAUPE also led to the development of test cases that target the inner working of the system. In particular, several test cases target the core components of the system and the implementations of the various design patterns.

10 http://www.junit.org/.

Table 1
Metric values computed on TAUPE source code.

	v0.1	v1.0	v2.0
Total number of lines	3193	8036	55,698
Total lines of code	812	4912	12,488
Number of packages	3	11	25
Number of classes	10	71	150
Number of interfaces	0	10	7
Number of methods	40	490	820
Number of attributes	30	248	338
Lack of cohesion of methods	0.329	0.370	0.224

Finally, to give an overall idea of the quality of the current version, Table 1 reports several metrics computed using Eclipse's *Metrics* plug-in[11] on the three available versions of the system.

4.6. Tool builders' issues

While building Taupe, we encountered several issues and learnt many lessons for future development. First, understanding the domain is of utmost importance, in particular when dealing with highly-specific domains, such as eye-tracking. Indeed, in the design and implementation of the first versions of Taupe, we overlooked certain concepts that we had to integrate into the latest version. Such concepts include that of ignorable areas of interest. We learnt the lesson that a thorough and systematic domain analysis is always necessary, even if experts are available, to formally describe all the concepts. We thus recommend developers to produce an exhaustive list of concepts and their relations before any implementation.

Second, understanding the expected use of the tools is necessary, again in particular when dealing with highly-specific domains. Indeed, as researchers used Taupe and came up with novel experiments and analyses to perform, we realized that the first versions of Taupe were missing certain concepts and features to help researchers during their experiments. Such features includes the ability to add new exporters. We learnt the lesson that, in research prototypes such as Taupe, it is important to provide access points to all the features of the tool so that researchers can customize the tools to their particular needs. We thus recommend developers to carefully design their tools with extensibility in mind.

Third, making explicit the process of installation, use, and extension of the tools greatly increase its ease of use by researchers. Indeed, it is important to ensure as much as possible that there is one and only one means to extend the tool to avoid confusing researchers when they want to implement a new analysis. We learnt the lesson that extensibility is a double-edge sword and that having one and a single access point only for each concept manipulated in the tool to ease the implementation of new analyses and to ensure that different researchers do not use different paths to access the same concepts, thus possibly leading to conflicts or problem of consistency.

5. Controlled experiments

We now present three case studies that show the use and the relevance of TAUPE to edit, visualize, and analyze eye-tracking data to further our understanding of program comprehension. The first case study was conducted by Jeanmart et al. [30] on the impact of the Visitor design pattern on comprehension and modification tasks; the second study was performed by van den Plas et al. [60] on the impact of the Composite and Observer design patterns on comprehension and modification tasks; and the third study was realized by De Smet, Lempereur et al. on the impact of the MVC architectural style on comprehension and modification tasks. For each case study, we succinctly recall its goal, null hypothesis, design, and results; then we describe the use of TAUPE in the study. We summarize the three controlled experiments in Table 2.

5.1. Impact of the visitor design pattern

Goal. The goal of this study is to analyze the impact of the Visitor design pattern [14] on comprehension and modification tasks in the context of the maintenance of programs. In comprehension tasks, subjects were asked about different functionalities provided by the program. In modification tasks, subjects were asked to specify classes, methods, or attributes that should be added or modified to add new features to the program or to modify an existing feature.

Hypotheses. The study hypotheses were:

- HC_{0_1}: A class diagram with the Visitor does not reduce the subjects' efforts during program comprehension when compared to a class diagram without it.

11 http://metrics.sourceforge.net/.

Table 2
The summary of the controlled experiments.

	Goal	Eye-tracker	Independent variable	Dependent variable	Measures
5.1	Impact of visitor design pattern on comprehension and modification	Eye-link II	Visitor design pattern	Average number of fixation Average duration of fixation effort	Norm-Rate ADRF NRRF
5.2	Impact of Composite and Observer design pattern comprehension and modification	Eye-link II	Composite pattern Observer pattern	Spatial density Transitional matrix Average fixation's duration ON target / All target	SD TM ADRF IN_All_i
5.3	Impact of MVC architecture style on maintenance tasks.	FaceLAB	Different variants of MVC design pattern	Subject's speed Subject's accuracy	Average time Correctness

- HC_{0_2}: A class diagram using the canonical representation of the Visitor does not reduce the subjects' efforts during program comprehension when compared to a class diagram using the canonical representation of Visitor with another layout.
- HM_{0_1}: A class diagram with the Visitor does not reduce the subjects' efforts during program modification when compared to a class diagram without.
- HM_{0_2}: A class diagram using the canonical representation of the Visitor does not reduce the subjects' efforts during program modification when compared to a class diagram using the canonical representation of Visitor with another layout.

Design. Three open-source projects were used in this experiment: *JHotDraw*,[12] *JRefactory*,[13] and *PADL*.[14] *JHotDraw* is a framework to implement technical and structured drawings, it provides support for the creation of geometric and user-defined shapes. *JRefactory* is a code refactoring tool for Java programs. *PADL* is a meta-model for describing object-oriented programs, it is similar to the UML meta-model.

The dependent variables chosen in the experiment were the *Average Number of Relevant Fixations* and the *Average Duration of Relevant Fixations*.

The *Average Number of Relevant Fixations* is used in the $NORM\_RATE_i$ measure as described in Section 4.3.3. In addition, the *Average Duration of Relevant Fixations* is denoted by the variable *ADRF* where *d(c)* is a function that gives the total duration of the fixations made to a class *c*.

$$ADRF = \frac{\sum_{c \in \{Rel.Classes\}} d(c)}{\#\{Rel.Classes\}}$$

The authors also defined a variable called *NRRF* to calculate the amount of a subjects' effort while performing a task on a diagram.

$$NRRF = \frac{\frac{\sum_{c \in \{Rel.Classes\}} f(c)}{\#\{Rel.Classes\}}}{\frac{\sum_{c \in \#\{Rel.Classes\} \cup \{NonRel.Classes\}} f(c)}{\#\{Rel.Classes\} + \{NonRel.Classes\}}} \quad (1)$$

The eye-tracking device used in this study was the Eye-link II; 24 subjects participated in the study.

Results. The results of this study were that the presence of the Visitor design pattern as well as its layout do not have a significant impact on the comprehension of UML class diagrams when the subjects must perform comprehension tasks but it has a statistically significant impact on modifications tasks [29].

Relevance of Taupe. Before collecting the eye-tracking data, they decided to use the *NORM_RATE* measure to analyze the data and statistically test the various null hypotheses. Jeanmart first implemented the formula in an Excel sheet, which he planned to use by copying/pasting the eye-tracking data into the sheet. It become quickly painfully apparent that copying/pasting thousands of pieces of data collected for each subject was a daunting and error-prone task. Then, Jeanmart implemented the formula as a command in the Taupe system. After less than a week of implementation and validation and verification, they were able to analyze all the collected data and statistically test their null hypotheses using Taupe user interface. The measure and statistical analyses are now available in Taupe for future studies.

12 http://www.jhotdraw.org.
13 http://jrefactory.sourceforge.net/.
14 http://wiki.ptidej.net/doku.php?id=padl.

Table 3
Average edit distance between group of subjects per program and ratios.

	ArgoUML	JUnit	QuickUML
Beginners	75.3	84.5	95.5
Experts	106	151.3	88.9
Experts/Beginners (%)	140.8	179.0	107.4

5.2. Impact of the composite and observer design patterns

Goal. The goal of this experiment was to analyze the impact of the Composite and Observer design patterns [14] during comprehension and modification tasks.

Hypotheses. The null hypotheses of the study were:

- H_{0_1}: The impact of the Composite design pattern on the average effort of subjects to perform comprehension and modification tasks is the same for beginners and for experts.
- H_{0_2}: The impact of the Observer design pattern on the average effort of subjects to perform comprehension and modification tasks is the same for beginners and for experts.

Design. Three open-source programs were selected to compose the questions of the study: *JUnit*, *QuickUML*,[15] and *ArgoUML*.[16] JUnit is a unit test framework for the Java programs. QuickUML is a diagramming program to draw UML class diagrams. ArgoUML is also a diagramming program that supports all UML diagrams and provides reverse-engineering facilities as well as exporting in various format.

Four metrics were selected as independent variables in this study to assess the subjects' effort: the spatial density, the transitional matrix, the average fixation's duration, and the ON-target/ ALL-target measure. The *ON-target/ ALL-target* measure is defined using the number of fixation within the specific area of interest, divided by the number of all fixations. Complete descriptions of the implementations of these in Taupe are available in Taupe user's guide (see footnote 8).

$$IN\_ALL_i = \frac{\#fix(i,j)}{\#F_j}$$

The level of expertise of the subjects was assessed using their employment. A subset of the subjects were experts from the Pyxis software company[17] while other subjects were students in the Ptidej Team (see footnote 7) and the Soccer Laboratory.[18]

The eye-tracker used to conduct this study was the Eye-link II; 24 subjects took part in the study.

Results. It was not possible to reject the null hypotheses using the *ON-target/ ALL-target* measure. However, the analysis of the subjects visual paths showed significant commonalities among experts on the one hand and beginners on the other and significant differences between the experts' visual paths and the beginners'. Table 3 reports that the average edit distance between the beginners' visual paths is always lower than that of experts' visual paths, independently of the considered program. This observation shows that beginners systematically browse a diagram while experts use their expertise to quickly gather the important information from the diagram.

Relevance of Taupe. To the best of our knowledge, the Taupe system is the only such system providing analyses of the visual path and able to compute the edit distance among a set of visual paths. Thus, it was instrumental in showing the commonalities among beginners and experts and the differences between beginners and experts.

5.3. Impact of different forms of the MVC design pattern

Goal. The goal of this experiment was to analyze the impact of different variants of the Model View Controller (MVC) [14] architectural style during maintenance tasks.

Hypotheses. The null hypothesis was:

- H_{0_1}: The different variants of the MVC architectural style are all equivalent during the maintenance of a program. The time and the visual path needed to complete the maintenance tasks on UML diagrams are the same, no matter the variant of the MVC.

15 http://sourceforge.net/projects/quj/.
16 http://argouml.tigris.org/.
17 http://www.ptidej.net/research/taupe/downloads/.
18 http://web.soccerlab.polymtl.ca/.

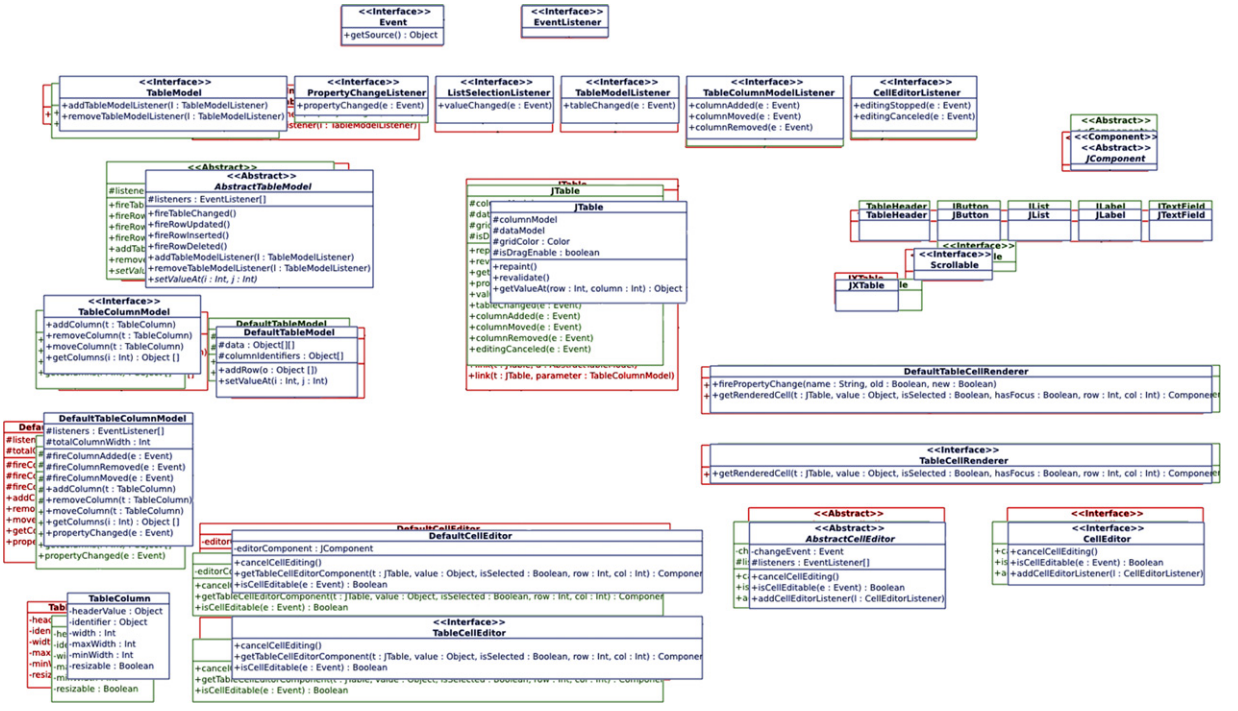


Fig. 12. Superposition of all the studied forms of the MVC style.

Table 4
Results collected for the different forms of the MVC architectural style.

	Comprehension			Modification		
	MD	MVC	MVP	MD	MVC	MVP
Average time (s)	66	87	47	40	64	41
Correctness (%)	54	77	60	54	77	60

Design. The class diagrams of two different programs were obtained by reverse-engineering and studied: the diagram of the `JTable` from *Java's Swing GUI* widget toolkit[19] and that of *JFreeChart*,[20] an open-source framework for Java programming language, which allows the creation of complex charts.

Different variations of the MVC were then implemented from these two diagrams and used in the study: the canonical form of the MVC style [14], the *Model-Delegate* style [57] (also called UI-Model or Document-View, MD), and the *Model View Presenter* style [41] (MVP). These variations are shown in Fig. 12.

The eye-tracking system used in this study was *FaceLAB* as described in 2.3.2. A total of 23 subjects participated to the study. However, only 18 subjects out of the 23 were valid. The "mortality" was due to the instability of *GazeTracker*, which sometimes fails to save the data on the hard drive and also to the poor quality of the recorded data at first.

Results. Table 4 reports the main data collected during this study. The analysis of this data showed that subjects spent less time on the MVP variant and that their answer were less correct with the MD style. Average times showed that the MVP variant is usually easier to understand than the two other variants. The correctness of the subjects' answers showed that the MVC variant is easier to understand than the others variants.

Relevance of TAUPE. Again, TAUPE eased the process of analyzing the recorded eye-tracking data by allowing us to integrate the algorithms necessary to compute the various measures as commands of the system. Moreover, we also used its visual path algorithm to further analyze the collected data, which results we plan to release in the near future.

6. Summary

We now summarize our contributions and future improvements.

19 http://java.sun.com/products/jfc/tsc/articles/architecture/.
20 http://www.jfree.org/jfreechart/.

6.1. Contributions

To the best of our knowledge, TAUPE is the first open-source analysis system for eye-tracking data built with compatibility and extensibility as main objectives. Compatibility is realized through the implementations of parsers for the data collected by two eye-tracking devices and the ability to add new such parsers. Extensibility is achieved by providing clear sets of application programming interfaces (APIs) to the various core components of the system and by using reflection to load automatically new implementations. TAUPE is released under the GPL license.

TAUPE is designed to accept data from multiple eye-tracking systems as input because eye-tracking techniques evolve every day and some new devices appear on the market. The fact that TAUPE is easy to evolve can contribute to the development of the field of eye-tracking.

TAUPE offers a set of algorithms of analyses in the field of eye-tracking. Although some of these algorithms were well-known and widely used in the field, others are offered for the first time in such a system, for example the analyses of the visual paths and their edit distances.

6.2. Improvements

In the next versions of TAUPE, we plan to add a feature to manually correct fixations with some constraints based on their relative distances. These constraints could be used when there is no static offset but an offset that is different in different areas of interest.

We also plan to extend the *AOIMaker* so that it can be used to manipulate fixations and saccades recorded by an eye-tracking device, for example to offset the data through drag and drop of the mouse to ease the users' analysis tasks.

We will also improve the performances of the TAUPE system in general and of its parsers in particular, which can be time consuming when thousands of fixations and saccades have been recorded, typically during long experimental sessions.

The usability of the TAUPE system could be improved by the introduction of graphical user interfaces allowing the users to interact directly with all of TAUPE input files, for example to input some information about the subjects instead of editing manually the `.subject` files.

The TAUPE system's feedback to the users could also be improved by showing more information though the progress bar and the list in the main window. A more thorough usage of the progress bar would improve the user interface. Data and results could also be displayed using charts to improve their visual analyses. Merging the fixations of a subject and coloring them according to their durations would allow TAUPE to generate heatmaps [62], which could further help researchers in identifying relevant area of interests.

Obviously, users can use the files generated with TAUPE in some external data-mining software systems, such as R.[21] Therefore, we also will create a specific *printer* for such systems. We will also study the feasibility of generating directly `.xls` or `.ods` files.

The current version of TAUPE only uses fixations and saccades collected by the eye-tracker devices but new devices provide new kind of data, such as pupil size, head position, blinking rate, and so on. Future versions should use this data to allow the development of even more sophisticated measures and analyses.

7. Conclusion

The activity of program comprehension has been studied by many researchers but only recently visual data have been gathered using eye-tracking devices to further improve our understanding of program comprehension processes. An eye-tracker records the coordinates of a subject's gaze when looking at a computer screen. It provides a new perspective on a subject's comprehension processes because it shows the areas attracting the subject's attention as well as the visual path of her gaze on the screen [11]. A subject's attention and visual path together form a window on her cognitive processes [43]. Thus, analyzing the data recorded using an eye-tracking device allows understanding in detail a subject's process of acquiring data, for example during program comprehension.

However, there were still some major obstacles to the widespread adoption of eye-tracking devices. Besides the costs of such devices, the provided analysis software systems were not open-source and often not extensible, preventing the development and seamless integration of new sophisticated analyses [27].

Consequently, we undertook the development of the TAUPE system to visualize, analyze and edit eye-tracking data with the main objectives of compatibility and extensibility. We presented the TAUPE system: its context, implementation (based on good practices and a thorough documentation, validation, and verification process), and three case studies using TAUPE.

TAUPE is being developed by the Ptitdej Team and released under the GPL and its development will continue in the future to improve its architecture, design, user interface, and to provide more sophisticated measures and analyses. We encourage researchers and practitioners to download its source code (see footnote 8) and to contribute with measures and analyses of their own.

21 http://www.r-project.org/.

Appendix. Supplementary data

Supplementary data to this article can be found online at http://dx.doi.org/10.1016/j.scico.2012.01.004.

References

[1] A.V. Aho, J.E. Hopcroft, J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.
[2] R. Bednarik, M. Tukiainen, An eye-tracking methodology for characterizing program comprehension processes, in: Proceedings of 5th symposium on Eye Tracking Research & Applications, ACM Press, 2006, pp. 125–132.
[3] B. Bellay, H. Gall, A comparison of four reverse engineering tools, in: I. Baxter, A. Quilici (Eds.), Proceedings of the 4th Working Conference on Reverse Engineering, IEEE Computer Society Press, 1997, pp. 2–11.
[4] D. Binkley, M. Davis, D. Lawrie, C. Morrell, To camelcase or under_score, in: Proceedings of the 17th International Conference on Program Comprehension, 2009, pp. 158–167.
[5] B. Boehm, H.D. Rombach, M.V. Zelkowitz, Foundations of Empirical Software Engineering: The Legacy of Victor R. Basili, 1st ed., Springer-Verlag, 2005.
[6] B.W. Boehm, Software Engineering Economics, Springer-Verlag New York, Inc., New York, USA, 2002, pp. 641–686.
[7] R. Brooks, Using a Behavioral Theory of Program Comprehension in Software Engineering, IEEE Press, Piscataway, NJ, USA, 1978.
[8] R. Brooks, Using a behavioral theory of program comprehension in software engineering, in: M.V. Wilkes, L. Belady, Y.H. Su, H. Hayman, P. Enslow (Eds.), Proceedings of the 3rd International Conference on Software Engineering, IEEE Computer Society Press, 1978, pp. 196–201.
[9] C.F. Chabris, S.M. Kosslyn, Representational correspondence as a basic principle of diagram design, in: Knowledge and Information Visualization, Springer-Verlag, 2005, pp. 36–57.
[10] H.C. Purchase, J.A. Allder, D. Carrington, Graph layout aesthetics in UML diagrams: user preferences, Journal of Graph Algorithms and Applications 6 (2002) 255–279.
[11] A.T. Duchowski, Eye tracking methodology, Theory and Practice 328 (2007).
[12] H. Eichelberger, Nice class diagrams admit good design? in: J.T. Stasko (Ed.), Proceedings of the 1st Symposium on Software Visualization, ACM Press, 2003, pp. 159–168.
[13] A. Endres, D. Rombach, A Handbook of Software and Systems Engineering, 1st ed., Addison-Wesley, 2003.
[14] Gamma Erich, R.J. Richard Helm, J. Vlissides, Design Patterns Elements of Reusable Object-Oriented Software, Addison-Wesley Pub. Co., 1995.
[15] Eye Response Technologies, 2009. GazeTracker Reference Manual. Eye Response Technologies Inc.
[16] P.M. Fitts, R.E. Jones, J.L. Milton, Eye movements of aircraft pilots during instrument-landing approaches, Aeronautical Engineering Review 9 (1950) 24–29.
[17] E. Gamma, K. Beck, Test infected: programmers love writing tests, Java Report 3 (1998) 37–50.
[18] J.H. Goldberg, X.P. Kotval, Computer interface evaluation using eye movements: methods and constructs, International Journal of Industrial Ergonomics 24 (1999) 631–645.
[19] Y.G. Guéhéneuc, A reverse engineering tool for precise class diagrams, in: J. Singer, H. Lutfiyya (Eds.), Proceedings of the 14th IBM Centers for Advanced Studies Conference, CASCON, ACM Press, 2004, pp. 28–41. 14 pages.
[20] Y.G. Guéhéneuc, Taupe: towards understanding program comprehension, in: H. Erdogmus, E. Stroulia (Eds.), Proceedings of the 16th IBM Centers for Advanced Studies Conference, CASCON, ACM Press, 2006, pp. 1–13. 13 pages.
[21] Y.G. Guéhéneuc, A theory of program comprehension—joining vision science and program comprehension, International Journal of Software Science and Computational Intelligence 1 (2009) 47 pages.
[22] D. Gusfield, Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, Cambridge University Press, New York, NY, USA, 1997.
[23] I. Hadar, O. Hazzan, On the contribution of UML diagrams to software system comprehension, Journal of Object Technology 3 (2004) 143–156.
[24] D. Hamlet, Foundations of Empirical Software Engineering, vol. 19, Springer, Berlin Heidelberg, 2005.
[25] ISO/IEC, 1996. EBNF Grammar Specification. Technical Report ISO/IEC 14977.
[26] D. Jackson, A. Waingold, Lightweight extraction of object models from bytecode, in: D. Garlan, J. Kramer (Eds.), Proceedings of the 21st International Conference on Software Engineering, ACM Press, 1999, pp. 194–202.
[27] R.J.K. Jacob, K.S. Karn, Commentary on section 4: eye tracking in human–computer interaction and usability research: ready to deliver the promises, 2002.
[28] S. Jeanmart, Evaluation de l'impact d'un patron de conception sur la comprehension et la maintenance de programmes — une experimentation par un systeme d'eye-tracking, Master's Thesis. Facultes Universitaires Notre-Dame de la Paix. Namur, Belgium, 2008.
[29] S. Jeanmart, A study of the impact of design patterns on program comprehension and maintenance activities, ACM SIGSOFT 2008/FSE 16, 2008.
[30] S. Jeanmart, Y.G. Guéhéneuc, H. Sahraoui, N. Habra, Impact of the visitor pattern on program comprehension and maintenance, in: J. Miller, R. Selby (Eds.), Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement, ESEM, IEEE Computer Society Press, 2009, 10 pages.
[31] M. Just, P.A. Carpenter, Eye fixations and cognitive processes, Cognitive Psychology 8 (1976) 441–480.
[32] H.M. Kienle, H.A. Muller, The tools perspective on software reverse engineering: requirements, construction, and evaluation, Advances in Computers (2010) 189–290.
[33] M.M. Lehman, Programs life cycles and laws of software evolution, Proceedings of the IEEE 68 (1980) 1060–1076.
[34] A. von Mayrhauser, Program comprehension during software maintenance and evolution, IEEE Computer 28 (1995) 44–55.
[35] G.C. Murphy, M. Kersten, M.P. Robillard, D. Čubranić, The emergent structure of development tasks, in: A.P. Black (Ed.), Proceedings of the 19th European Conference on Object-Oriented Programming, Springer-Verlag, 2005, pp. 33–48.
[36] A. Newell, You can't play 20 questions with nature and win, in: W. Chase (Ed.), Visual Information Processing, Academic Press, 1973.
[37] S.E. Palmer, Vision Science: Photons to Phenomenology, 1st ed., The MIT Press, 1999.
[38] D.E. Perry, A.L. Wolf, Foundations for the study of software architecture, SIGSOFT Software Engineering Notes 17 (1992) 40–52.
[39] A. Poole, L.J. Ball, In search of salience: a response time and eye movement analysis of bookmark recognition, People and Computers XVIII-Design for Life: Proceedings of HCI 2004, 2004.
[40] A. Poole, L.J. Ball, Eye tracking in human–computer interaction and usability research: current status and future prospects, in: Claude Ghaoui (Ed.), Encyclopedia of Human Computer Interaction, 2006.
[41] M. Potel, MVP: Model-view-presenter — the taligent programming model for C++ and java. Taligent, Inc., 1996.
[42] V. Rajlich, Program comprehension as a learning process, in: Y. Wang (Ed.), Proceedings of the 1st International Conference on Cognitive Informatics, IEEE Computer Society Press, 2002, pp. 343–347.
[43] K. Rayner, Eye movements in reading and information processing: 20 years of research, Psychological Bulletin 124 (1998) 372–422.
[44] J. San Agustin, H. Skovsgaard, E. Mollenbach, M. Barret, M. Tall, D.W. Hansen, J.P. Hansen, Evaluation of a low-cost open-source gaze tracker, in: Proceedings of the 2010 Symposium on Eye-Tracking Research #38; Applications, ACM, New York, NY, USA, 2010, pp. 77–80.
[45] Seeing Machine, 2010. Seeing Machine's website — FaceLAB. http://www.seeingmachines.com/product/facelab/ (accessed 23.12.2010).
[46] B. Sharif, J. Maletic, The effect of layout on the comprehension of uml class diagrams: a controlled experiment, in: 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT 2009, IEEE, 2009, pp. 11–18.

[47] B. Sharif, J.I. Maletic, An eye tracking study on camelcase and under_score identifier styles, in: Proceedings of the 18th International Conference on Program Comprehension, 2010, pp. 196–205.
[48] B. Sharif, J.I. Maletic, An eye tracking study on the effects of layout in understanding the role of design patterns, in: Proceedings of the 2010 IEEE International Conference on Software Maintenance, IEEE Computer Society, Washington, DC, USA, 2010, pp. 1–10.
[49] F. Simon, F. Steinbrückner, C. Lewerentz, Metrics based refactoring, in: P. Sousa, J. Ebert (Eds.), Proceedings of the 5th Conference on Software Maintenance and Reengineering, IEEE Computer Society Press, 2001, pp. 30–38.
[50] E. Soloway, Learning to program = learning to construct mechanisms and explanations, Communications of ACM 29 (1986) 850–858.
[51] I. Sommerville, Software Engineering — Fifth Edition, Addison-Wesley Publishing Company, Reading, MA, 1996.
[52] D. Spinellis, Code Reading: The Open Source Perspective, 1st ed., Addison Wesley, 2003.
[53] SR Research Ltd., 2006. EyeLink II User Manual version (07/02/2006). SR Research Ltd.
[54] M.A.D. Storey, F.D. Fracchia, H.A. Müller, Cognitive design elements to support the construction of a mental model during software exploration, Journal of Systems and Software 44 (1999) 171–185.
[55] Cepeda Porras Gerardo, Y.G. Guéhéneuc, An empirical study on the efficiency of different design pattern representations in UML class diagrams, Empirical Software Engineering 15 (2010) 27 pages.
[56] D. Sun, K. Wong, On evaluating the layout of UML class diagrams for program comprehension, in: J.R. Cordy, H. Gall (Eds.), Proceedings of the 13th International Workshop on Program Comprehension, IEEE Computer Society Press, 2005, pp. 317–326.
[57] F. Swartz, Ui-model structure, 2004. http://www.leepoint.net/notes-java/GUI/structure/30presentation-model.html (accessed 18.10.2010).
[58] The Institute of Electrical and Electronics Engineers, 1990. IEEE standard glossary of software engineering terminology. IEEE Standard.
[59] H. Uwano, M. Nakamura, A. Monden, K. ichi Matsumoto, Analyzing individual performance of source code review using reviewers' eye movement, in: Proceedings of the 2006 symposium on Eye Tracking Research & Applications, 2006, pp. 133–140.
[60] B. Van Den Plas, La theorie "Vision-Comprehension" appliquee aux patrons de conception, Master's Thesis, Facultes Universitaires Notre-Dame de la Paix, Namur, Belgium, 2009.
[61] A. Von Mayrhauser, A.M. Vans, Program Comprehension During Software Maintenance and Evolution, IEEE Computer Society Press, Los Alamitos, CA, USA, 1995, volume 28, pp. 44–55.
[62] L. Wilkinson, The history of the cluster heat map, American Statistician 63 (2009) 179–184.
[63] S. Yusuf, H. Kagdi, J.I. Maletic, Assessing the comprehension of UML class diagrams via eye tracking, in: E. Stroulia, P. Tonella (Eds.), Proceedings of the 15th International Conference on Program Comprehension, IEEE Computer Society Press, 2007, pp. 113–122.

View publication stats