A Paper Interface for Code Exploration

Quentin Bonnard
CRAFT - EPFL
Rolex Learning Center
Station 20
CH-1015 Lausanne
quentin.bonnard@epfl.ch

Frédéric Kaplan
CRAFT - EPFL
Rolex Learning Center
Station 20
CH-1015 Lausanne
frederic.kaplan@epfl.ch

Pierre Dillenbourg
CRAFT - EPFL
Rolex Learning Center
Station 20
CH-1015 Lausanne
pierre.dillenbourg@epfl.ch

ABSTRACT

We describe Paper Code Explorer, a paper based interface for code exploration. This augmented reality system is designed to offer active exploration tools for programmers confronted with the problem of getting familiar with a large codebase. We first present an initial qualitative study that proved to be useful for informing the design of this system and then describe its main characteristics. As discussed in the conclusion, paper has many intrinsic advantages for our application.

Author Keywords

Paper interface, code exploration, augmented paper, tangible interface

ACM Classification Keywords

H.5.1 Information Interfaces and Presentation: Artificial, augmented, and virtual realities; H.5.2 Information Interfaces and Presentation: Training, help, and documentation

General Terms

Design, Human Factors.

CONTEXT AND MOTIVATION

Programmers are often confronted with the problem of getting familiar with a large codebase. In a typical scenario, new programmers coming to an institution have to learn about a project in order to start their own contribution. This is usually a difficult challenge. Different strategies are commonly used. The new programmer can browse the documentation, when it exists. This gives a broader overview, but does not allow for in-depth exploration. In a complementary manner, he can follow goal-oriented tutorials. He may also actively learn about the code by fixing bugs or developing unit tests. While usually more motivating for a programmer and useful for the project, this solution does not give a good overview of the overall architecture.

In this article, we present a novel tool for code exploration, based on a paper interface. Paper interfaces intend to provide computing abilities to paper, while trying to keep its simplicity and advantages over electronic devices. They can be used to augment traditional documents, but also create music [2], design warehouses [7], etc. However, programming appears as one of the tasks least adapted for paper interfaces, as it is one of the few intellectual activities that did not exist before the digital age. Indeed, very mature software exist for coding, and text input only makes paper a very poor competitor to a keyboard/mouse/screen system. Nevertheless we believe that paper has a rich set of properties that fits particularly well in the specific scenario we just described.

We describe hereafter Paper Code Explorer, a system designed to take the most of paper for code exploration. This is a Digital Desk-like system [5] and follows in a long tradition of using tagged paper for interaction control [1]. This tool focuses on code understanding and is not meant to be used for testing, debugging or developing. Nevertheless, code exploration plays an important role in any of such programming activities. Several techniques are commonly used to navigate between the portions of code displayed on the rather limited screen real estate: bookmarks, hyperlinks between definitions and occurrences, hierarchical index of the components of the workspace, outline of the displayed resource, tabs, etc. The contexts in which such active exploration has positive learning outcomes have been the subjects of many studies. One findings is the crucial need of a global map/representation to make the best use of hypertextual navigation: allowing flexible in-depth exploration while not getting lost in the process [4]. We believe that paper interfaces are good candidates to offer alternative solution to this classical problem.

In the next section, we describe an initial study involving a new programmer getting familiar with a code library using a mock-up of Paper Code Explorer. Informed by this initial study, we then present the main characteristics and components of our code exploration system.

AN INITIAL STUDY

Before designing the system, we set up a mock up of Paper Code Explorer and used the opportunity of a new colleague joining our team. A member of the team (the expert) was to explain him (the novice) how to start working in our codebase. Naturally, the expert would have walked through the code, and the novice would have asked question as they ar-

UbiComp '10, Sep 26-Sep 29, 2010, Copenhagen, Denmark.
ACM 978-1-60558-843-8/10/09.

Figure 1. Paper sheets and a computer mocked up Paper Code Explorer.

rive. We asked them to use a simple mock-up of Paper Code Explorer, to identify some flaws and validate some ideas. The mockup, shown on Figure 1, consists of small sheets of paper for each of the classes selected beforehand by the expert. Each class was represented by a box containing three boxes: one for the name of the class, one for the list of the name of the fields of the class, and the list of the name of the methods of the class. Both lists were ordered by decreasing visibility of the member (public, then protected, then private). Common paper related tools were placed on the table: sticky notes, stapler, pens, scissors, tape and paper clips. A large sheet of paper was placed as a support. The screen of the expert's computer was projected in front of them, and a regular keyboard/mouse controlled the computer. We interrupted the session several time to ask the novice some questions. This experience was not exactly what Paper Code Explorer is intended for: the expert guided the novice similarly to a tutorial documentation, as opposed to the novice exploring the code alone to build his own representation. This is linked to the fact that the novice only needed to know a small subset of the framework, as he would only work on a restricted aspect (improving a path finding algorithm); the novice does not need to learn everything about the software, only where to contribute.

This informal experiment is not an evaluation, but a first step in the design, which made possible following observations:

- *Paper representation of classes supports large overview.* Before even starting to discuss the architecture, the novice identified a design flow almost immediately when looking at the sheets in front of him: "There is a GetInstance() method everywhere so it's probably useless". (In fact, it corresponds to an abuse of the singleton design pattern for convenience reasons.)

- *Paper interfaces for code exploration should support a flexible navigation system adapted to the different granularity levels of the codebase.* The vertical navigation (from package to line of code and vice-versa) is at least as important as the horizontal navigation (from a class to another). Most notably, the expert started by drawing main components of the software and their relationship to each other, or walked through the main method block of code by block of code. Moreover, the expert navigated mostly using `show definition of` commands.

- *Paper representation of classes should include the visual characteristics of the corresponding source code.* The novice noted that the plain list of members does not give a feeling of the size of the file (in number of lines) as the size of the scrollbar cursor does on the computer. Furthermore, the list of methods does not show how big each method is, which is an important data. The novice further suggested that the ordering of the members by visibility is not very helpful; it would be more interesting to group private functions with the public function calling them. These comments inspired the design of the flash cards shown on Figure 2.

- *Code understanding tools should support visual, active exploration.* An interview with the novice the day after the experiment revealed that he was remembering the size of the classes better than their names. Even if short questions during the experiment showed that the expert's explanations were clear, the novice did not remember most of them the day after, which is another example that passive learning as in a walk through is not effective.

DESIGNING PAPER CODE EXPLORER

This section describes the design choices for the on-going implementation of Paper Code Explorer.

Base Components

Paper Code Explorer is a software meant to be used with an *augmented lamp*, i.e. a projector and a camera above a desk. It uses the ARTag fiducial markers system [1] to track and project an augmentation paper sheets of various sizes and forms. It is integrated in the Eclipse environment[2], which provides a very mature framework to handle code, and is easily extensible and customizable. Java is hence a good candidate as the language of the codebase to explore, as it is a broadly used language, and big open source projects in Java are not hard to find.

Paper Classes for a Broad Overview

Paper Code Explorer uses papers sheets in two ways: as support for objects and as ways of triggering contextual commands. We print flash cards containing the name of the class, the list of its members, and a tag allowing Paper Code Explorer to map the paper to its logical counterpart. Using classes as the unit of paper objects is a good compromise for our code understanding objective: printing line does not scale to big codebases, which is our target, and packages are too coarse for a deep enough understanding. These flash cards could be made out of cardboard or paper, depending on the relation we want to build between the user and the objects. We prefer using cheap, easily duplicable paper flashcards which can be cut, annotated or thrown away without

[1] `http://www.artag.net/`
[2] `http://www.eclipse.org/`

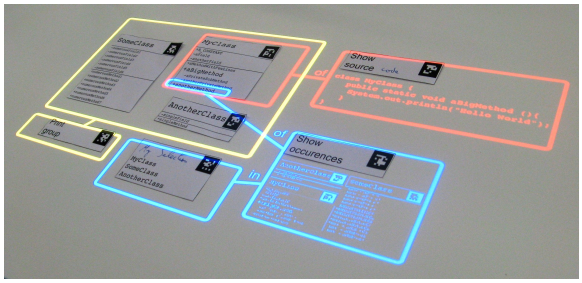

Figure 2. Three classes (MyClass, SomeClass and AnotherClass) are used for this illustration. The paper command `show source` is applied on MyClass, and projects its result in red. The paper command `print group` has been used to produce the paper object representing the three classes circled by the yellow box. The surface taken into account is delimited by a yellow projection. The paper paper command `show occurrences` is applied on one of the methods of MyClass, and the resulting set of class is restricted to the previously mentioned group. The results are projected in blue.

consequence and therefore should allow for more flexible usages.

Using paper classes on a desktop rather than a class diagram on a screen has the main advantage of providing a clearly bigger surface, which is important for such a layout based visualization. This allows for a navigation in the layout that does not require scrolling, which takes full advantage of spatial memory. Paper is tangible, so it is more natural to manipulate than a window with a cursor: moving is easy, several objects (close to each other) can be selected and moved at once. The motivation here is to allow the user to layout the classes according to relationships that make sense to her, and offload the working memory of these relationship onto the spatial organization.

Augmenting the Paper for In-depth Exploration

The list of the members of a class is usually not sufficient to understand it; we need to be able to show the code of the class or its documentation. The lamp can provide for such an augmentation of the objects, as shown on Figure 2 The scenario is comparable to a menu-based interface: the user selects selects an object and a command to apply on it. In our case, we put a command paper sheet corresponding for instance to `show the documentation of` or `show the source code of` a nearby object. A pop-up is then projected with the corresponding content, which can be scrolled by moving up and down the object paper relatively to a fixed, projected reference point.

On this aspect, the interaction zone is limited to the area covered by the lamp, which is comparable to the one of a screen. However, the advantage of a paper interface is that the menu can exist outside of this area. The user does not have to find a compromise between accessibility of various actions in menu and the space allocated to objects. More over, the same manipulability advantage of paper applies to actions: they can be organized and grabbed easily. Command results being linked spatially to them can be moved in an equally easy manner.

Manipulable Queries

The chosen granularity of the classes does not mean that command inputs can not have a finer granularity. Let us consider the command `show occurrences of` a member of a class (a method or field). This command is augmented with a pointer projected at a fixed position relatively to the paper. This precise pointer can be manipulated as easily as the paper, and allows a communication with Paper Code Explorer as precise as a mouse. It can also be interesting to apply actions on augmented content, e.g. `show definition of` a variable in the projected source code of a class.

Actions are not limited to unary operators. For example, it is important in our scenario to show the relationships between several classes: inheritance, aggregation, function calls, etc. Code can be considered as a semi-structured database which can be queried [3], for example on private methods returning a String and using a given member. We make such queries tangible and manipulable: the user can modify them easily (changing the input or the parameters) and observe the changes in real time. For example, a query on all occurrences of a method can give too many results, so the user can restrict them to occurrences within a given class. If this is too restrictive, the occurrences can be restricted to a wider set of classes. All these modifications on the query happen by incrementally adding and removing paper objects as a feedback to the result.

Also, some queries can have a result mentioning classes that are not in the augmented area. In this case, we use the fact that printed paper have a fixed text layout which can be easily remember. To be more concrete, let us consider the command `find classes using` a given member. When used, it projects thumbnails of the corresponding flashcards rather than the full flashcard in order to spare the display area. The user is maintaining a spatial arrangement of the printed classes on the side. This way she can match the form of the projected thumbnails with the one of the classes displayed on the side. Of course, other techniques can replace thumbnails if they are not adapted to the size of the codebase. The goal is not to remember the fixed layout of the whole codebase, but rather offload the working memory using features of the human vision.

Active Reading Using the Paper Interface as Information Support

A musician or a writer, for example, annotate heavily the document they are working on. This behavior can be found in most, if not all processes involving documents. On this topic, source code does not appear as a document: it is the same before and after spending time to understand it. Code can be commented, but these comments are usually not personal understanding notes.

Compared to screen-based exploration, it is clear that paper interfaces offer a much larger variety of tools to read in an active manner. In our scenario it is very easy to annotate the printed code in much the same way one would annotate a printed article. The flash cards corresponding the classes can be underlined or highlighted in any color in a natural

way, free text can be written simply (consider for example writing the mathematical formula computed by a method), free forms can be drawn, etc. In addition, a reader can bookmark, highlight, link elements, underline or circle. The issue of extracting annotations have been addressed already [6], and the extracted data can easily be associated to the digital content used to generate the paper object.

To go one step further, one can save a layout by pasting the papers on a sheet. This creates a new composite object, that in turn can be annotated, named for a faster recall, linked with one another, etc. Alternatively, the paper classes can be stacked, folded, cut or teared apart.

A Printer on the Desktop for Memory Cycles

There are 3 layers on Paper Code Explorer. The printed layer forms the base of the interface, the augmentation displays digital information on it, and the user writes on the printed layer, possibly using the augmented information. The difference between the printed and written information is that the printed information can be duplicated easily. The interesting point is that these three layers can be merged into a printed or an augmented layer. This allows for a physical or virtual snapshot of the interface/memory, respectively. Such snapshots are useful for versioning the exploration work: they allow the user to save the state of the interface, e.g. save a grouping of the paper classes in case the new one is not as good. They are also useful for recovering from interruption. Moreover, the physical snapshot are paper objects themselves, and can be annotated too: giving it a title for example helps the interruption recovery furthermore.

To integrate the Print-Augment-Write (PAW) iterations in the workflow, it should be as easy as possible to create a new paper object. These new objects have to be usable by the user and by the Paper Code Explorer. To do so, they are assigned a tag so that the system can map the paper object to its information. A printer allows such a controlled creation process. Receipt printers are very adapted to our case: they are relatively small, and can be placed on the desktop, making them reachable but not too invasive. They can achieve sufficient speed (e.g. 7 inch per second) and receipts are not valuable per se (they are valuable if they prove the payment of something expensive, but are discarded in all other cases), removing the restraints a user could have to print and use temporary documents.

Practically, Paper Code Explorer allows to create a paper object representing a group of other paper objects. This group can be used as an alias of its content in a manipulable query (e.g. `show the relationships between` this group), or as a manipulable abstraction (e.g. a module to relate to other modules). The creation of new paper objects can be triggered with paper commands, such as `print` one of the classes shown in the result set displayed by a manipulable query. Paper commands can be duplicated too, or even combined, allowing a user defined menu of commands. The printer also allows to give more freedom in the starting point of the exploration: printing all the classes would not be very scalable for example; it is more interesting to start with an overview in which the user *zooms* by printing the details of chosen elements.

RELEVANCE OF PAPER-BASED INTERFACE

Although the definitive implementation of the Paper Code Explorer is still on going, we can already discuss the relevance of paper-based interfaces in this context. The manipulability of paper is an excellent way to navigate in a complex system such as a software architecture. The fact that paper remains visible outside the interaction zone eases the access and organization of commands. Complex objects such as queries can be built and modified intuitively. Code can be annotated freely with a pen, which, among other things, helps a lot the activity of reading for understanding.

One of the main objectives of paper and tangibles interfaces consists of augmenting the functionalities of a physical object without reducing its simplicity of usage and original advantages. In Paper Code Explorer, most of the fundamental advantages of paper are preserved. However, the augmentations do not offer the same level of interactivity than a traditional computer interface. Our system focuses on code exploration and does not support the input of code. The resolution of the projected augmentation or of the pointers is not as high as those of a screen or mouse, respectively. Nevertheless, we believe that the intrinsic benefits of a paper-based interface for learning justify this compromise at the interactivity level. In order to evaluate this choice more thoroughly, we intend to perform a comparative study with a tabletop display interface.

REFERENCES

1. T. Arai, D. Aust, and S. Hudson. PaperLink: a technique for hyperlinking from real paper to electronic content. In *Proc. SIGCHI conference on Human factors in computing systems*, pages 327–334. ACM, 1997.

2. E. Costanza, M. Giaccone, O. Kueng, S. Shelley, and J. Huang. Tangible interfaces for download: initial observations from users' everyday environments. In *Proc. CHI EA 2010*, pages 2765–2774.

3. E. McCormick and K. De Volder. Jquery: finding your way through tangled code. In *Proc. OOPSLA 2004*.

4. J. Rouet and A. Tricot. Task and activity models in hypertext usage. In *Cognitive aspects of electronic text processing*, pages 239–264. 1996.

5. P. Wellner. Interacting with paper on the DigitalDesk. *Communications of the ACM*, 36(7):87–96, 1993.

6. Y. Zheng, H. Li, and D. Doermann. Machine printed text and handwriting identification in noisy document images. *IEEE transactions on pattern analysis and machine intelligence*, 26(3):337–353, 2004.

7. G. Zufferey, P. Jermann, A. Lucchi, and P. Dillenbourg. Tinkersheets: using paper forms to control and visualize tangible simulations. In *Proc. TEI 2009*, pages 377–384.