Two-Dimensional Positioning as Visual Thinking

Shingo Takada[1], Yasuhiro Yamamoto[2], and Kumiyo Nakakoji[234]

[1] Keio University, Yokohama, 223-8522, Japan
michigan@ics.keio.ac.jp
[2] Nara Institute of Science and Technology, Nara, 630-0101, Japan
{yasuhi-y, kumiyo}@is.aist-nara.ac.jp
[3] SRA, Inc
[4] PRESTO, JST

Abstract. People depend on various external representations in various design situations. These external representations are necessary at the time of creation in early stages of a design task, as they help the designer visualize what they are thinking and continue with their task in the process of reflection-in-action. Designers in domains such as architecture have drawn diagrams, or sketches, as the external representations. We take writing and programming as two example domains, and argue that *two-dimensional positioning* serve the same purpose for these domains as *diagrams* do for architectural design. We describe two tools, ART for writing and RemBoard for component-based programming, which help writers or programmers visualize what they are thinking through positioning parts of writing or software components on a two-dimensional space. We examine the issues that are necessary for this, and explore how they were handled in the two tools.

1 Introduction

Design occurs in the first stages of any construction of artifacts. The design task has been characterized as ill-structured [9], and the problem and solution co-evolve, going through a process of "reflection-in-action"[7]. Rather than pre-plan what to do during a design task, the designer is engaged in a cycle of producing a representation (e.g. a sketch) and reflecting on them. The representation serves as a "situation" that talks back to the designer, through which the designer asks:

- what "parts" are missing;
- how much the designer is "sure" about a newly created part;
- what the role of this newly created part is in terms of the whole design;
- what the role of this newly created part is in terms of the other parts; and
- which direction the whole design is moving toward and whether the direction is in accordance with the intention behind the design.

The representation is useful only for the designer him/herself at the time of its creation [4]. The designer is free to make conventions that only has meaning to him/herself. Of course, in some cases, the representation may be intended for other people such as the designer's client. But while the actual work (design) is going on, the representation is only for the designer and not for other people, and as a result if a design-in-process is seen by other people, the representation may be interpreted differently from what the designer had intended [4].

This representation has been called an externalization of what the designer is thinking [1], or put another way the current situation of the design task. According to Bruner, the externalization of what the designer is thinking is important because it *"relieves us in some measure from the always difficult tasks of 'thinking about our own thoughts' while often accomplishing the same end. It embodies our thoughts and intentions in a form more accessible to reflective efforts"* [1]. This suggests that the form that the externalization takes is important. If the externalization does not take a suitable form, then it does not help with *'thinking about our own thoughts'*.

In many domains, such as architecture, externalization has often taken the form of sketches, which can be thought of as a type of diagram that is done roughly and will be elaborated later. Designers will make sketches of what they are thinking, reflect on it, and then make changes as necessary. The sketch becomes *"a sort of hypothesis or 'what if' tool"* [4].

In other domains or situations, it may be more or less difficult to conduct a conventional sketch. For example, consider the situation where someone is writing a document. The writer already has parts of the document typed into a computer and wants to use it. How can the writer "sketch" the document using those existing text fragments? Consider another situation where someone is writing a program using a vast library of software components to choose from. How can the programmer make a "sketch" of the program while keeping in mind the various alternatives that may arise while searching for software components to use?

We consider the two domains of writing and component-based programming, and argue that two-dimensional positioning can serve the same purpose for these domains as "sketches" do for architectural domains. We describe two tools – ART for writing and RemBoard for component-based programming – that help designers visualize what they are thinking by positioning parts of their tasks on a two-dimensional space.

The rest of this paper first begins with an overview of why writing and programming need support for a form of "sketching". We compare these with solving a jigsaw puzzle, and contend that two-dimensional positioning can serve as a "sketching" tool. Section 3 then describes the two tools (ART and RemBoard), and how they helped users in their tasks. Section 4 discusses issues concerning how visualization is supported in the two tools. Section 5 makes concluding remarks.

2 Writing, Programming, and Jigsaw Puzzle

We consider writing and component-based programming as two domains that need some form of "sketching". We compare it to solving jigsaw puzzles, and contend that two-dimensional positioning can serve as a “sketch”.

2.1 Writing

Designing in writing has conventionally taken place as making an outline of the document to be written. This takes the position that one first identifies a stable structure, then gradually elaborates it from chapter to section to paragraph to sentence.

With the proliferation of computer tools such as word processors, problems have arisen mainly resulting from the ease with which text can be changed [13]. For example, writers will start typing (writing) a paragraph, but because a word processor is used, they can easily cut-and-paste parts of that paragraph, or copy various notes from previous work. They may do this even though they may not understand how the parts can be connected, or how the parts can be put together.

The writers engage in a trial-and-error process of trying various combinations and organizations of written pieces, and see if they can make sense out of them. They will explore what position the current piece can have within the whole document, what pieces need to be modified, and what has and has not yet been written, etc. The document content itself is not the only information that is important, but information that pertain to the state of the document and its structure, which Kintsch and van Dijk called meta-comments, also play a role in writing [3].

Thus writers will need to be able to visualize various factors concerned with the current situation, i.e. they will need some method of “sketching”.

2.2 Component-based Programming

Component-based programming is a paradigm where programmers will develop software by “gluing together” various components (or program parts). The task basically consists of (1) finding the necessary components, and (2) making the “glues” necessary to put together the components. We focus on the former.

The search for necessary components in component-based programming is critical as they are the building blocks for the program. Programmers however may not necessarily know exactly what type of components are needed. Although they will at least have conceptually top-level requirements defined for the program, they may not know the details.

The programmers will engage in a trial-and-error process of searching for components that can be used in their program. This process is complicated because the programmers cannot decide which components to use unless they fully understand their detailed behavior. Furthermore, unlike conventional programming paradigms, the component package[1] has an affect on what the programmer

[1] A component package is a library that contains the software components from which the programmer chooses to build software.

will make at the detailed level. Components in a certain package may make certain requirements possible (or impossible) to implement, further complicating the decision of choosing a component. McKinney went as far as to say that the component package should "drive your requirements allocation" [5].

Thus, the programmers will need to remember intermediate search results, i.e. in order to continue searching, the programmer will need to be able to keep track of what he/she needs, what he/she currently has, and the relationships between what he/she has. In other words, the programmer needs some method of "sketching", allowing them to visualize the current situation.

2.3 Jigsaw Puzzle

We can view both writing and component-based programming as solving a jigsaw puzzle. A person solving a jigsaw puzzle needs to try various combinations of pieces, but needs to see at each instance the state of the task, e.g. what has been finished, what pieces need to be considered, etc. By being able to see the current state, the person can consider the next step to take. Through this process, he/she will obtain a coherent whole, i.e. a completed puzzle.

As we described above, a writer may have several written pieces in hand and need to explore how they can be combined, what needs to be modified, etc. The writer can only do this exploration if he/she can see the current state. Through seeing the current state, the writer can decide on for example what changes to make, ultimately resulting in a coherent whole, i.e. a document.

Similarly, a programmer may need to search through a library of components to develop a program. He/she will need to keep track of what type of component is needed, what he/she already has, and the relationships between what he/she already has. This will let the programmer continue searching for other components, until the goal is reached of obtaining a coherent whole, i.e. a program.

Of course there are differences between writing, programming, and solving jigsaw puzzles, and we will discuss the differences in Section 4.

2.4 Two-Dimensional Positioning

We characterized writing and programming to be similar to solving a jigsaw puzzle. Writers and programmers need to fit the various pieces together so that the document or program will become a coherent whole.

People solving a jigsaw puzzle will often use some space such as a table, and put pieces on it according to some meaning. For example, if a person thinks that a piece corresponds to a certain part of the picture, that person will likely place it at a corresponding place on the table. Other pieces may be grouped together according to its color, or whether it is a side piece. The positioning of the pieces has some meaning that will help the person solve the puzzle.

Much like positioning puzzle pieces on a table will help solve a jigsaw puzzle, we take the approach of positioning elements in a two-dimensional space as a means of visualizing the current situation. Elements correspond to the pieces in the jigsaw puzzle. For writing, elements are “parts of writing”: phrase, sentence, paragraph, section, etc. In component-based programming, the elements would be the software components themselves.

According to Shipman et al [8], people use the visual and spatial characteristics of graphical layouts to express relationships between icons and other visual symbols. Thus, people can position elements in a two-dimensional space to help visualize and understand their current situation, from which they can reflect and continue with their task, much like sketches in the architectural domain.

We next describe the two tools we have developed for writing and programming.

3 Two Tools – ART and RemBoard

We describe the two tools that we have developed for the writing and programming domains. We also describe how users were able to use them to visualize and understand the current situation.

3.1 ART

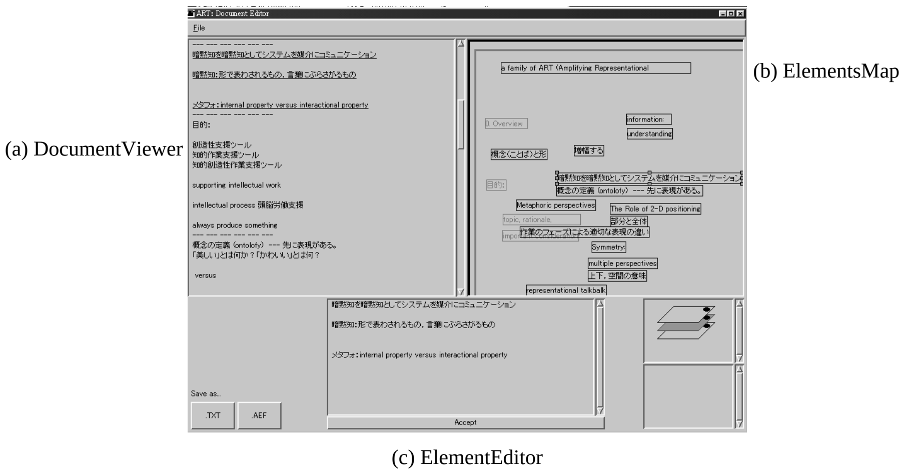

Fig. 1. ART

The ART system [12] [6] (Fig. 1) supports document construction as a design task allowing users to position segmented text as “elements” in a two-dimensional space. An element is any unit that writers choose to think of as one, such as a phrase, a sentence, a paragraph, or a longer piece of text.

System Overview The ART system consists of the following three main parts: ElementsMap, ElementEditor, and DocumentViewer.

ElementsMap (Fig. 1 (b)) is a two-dimensional space that graphically displays elements that comprise the document. Each element is represented as an icon. An icon does not show the entire content of the element, but only the first ten percent of the element's text; therefore, the size of the element box corresponds to the size of the actual element (unless the user resizes the box). A user can freely change the position of elements by pointing and dragging icons on ElementsMap.

Elements can be created and edited with ElementsMap and ElementEditor (Fig. 1 (c)). The ElementEditor component is a text editor for the contents of an element providing editing functions such as cut, copy, paste, and "spin off," which divides one element into two. Selection of an element in ElementsMap allows a user to modify the content of the element in the ElementEditor. When nothing is selected in ElementsMap, a user can edit text in ElementEditor and create a new element by positioning the newly created icon in ElementsMap. Two or more elements can be merged by selecting multiple elements on ElementsMap.

One point of note is that an element's vertical position in the ElementsMap is interpreted as corresponding to its position in the document sequence, and the DocumentViewer (Fig. 1 (a)) component displays the actual content of the document by sequentially scanning the elements displayed in the ElementsMap from top to bottom. Thus, a user can freely change the order of elements in the whole document by changing the vertical relationship of elements in ElementsMap. Positioning changes and content changes made in ElementsMap and ElementEditor are automatically reflected in all three parts.

Visualizing the Current Situation The ART system provides three types of "views" to understand the current situation. The ElementsMap provides an overview of the entire situation, while ElementEditor provides details of each element. The DocumentViewer displays the context of the element with details of neighborhood elements. The three views are integrated and changes made in one view are dynamically reflected on the other views.

User studies [6] have shown that positioning on the ElementsMap is the core of using the system. For example, one subject placed elements in the bottom right corner of the ElementsMap. This let the subject know that those elements needed further work. The subject took one of those unfinished elements, made modifications, and then placed them with other elements that already had some work done on them. The subject would then repeat this process. Another subject made a set of elements be the same size and carefully aligned them; this was to indicate that these elements were "finished". Still another subject made some elements particularly larger than others so that it would "call for attention" later in the task.

In one situation, a user felt that two elements "should be related to each other but I don't know how they are related". Instead of aligning them, she overlapped them to let her understand what she was thinking.

In this way, each user visualized various situations in their own way to help them understand their current situation, and let him/her proceed with the task.

3.2 RemBoard

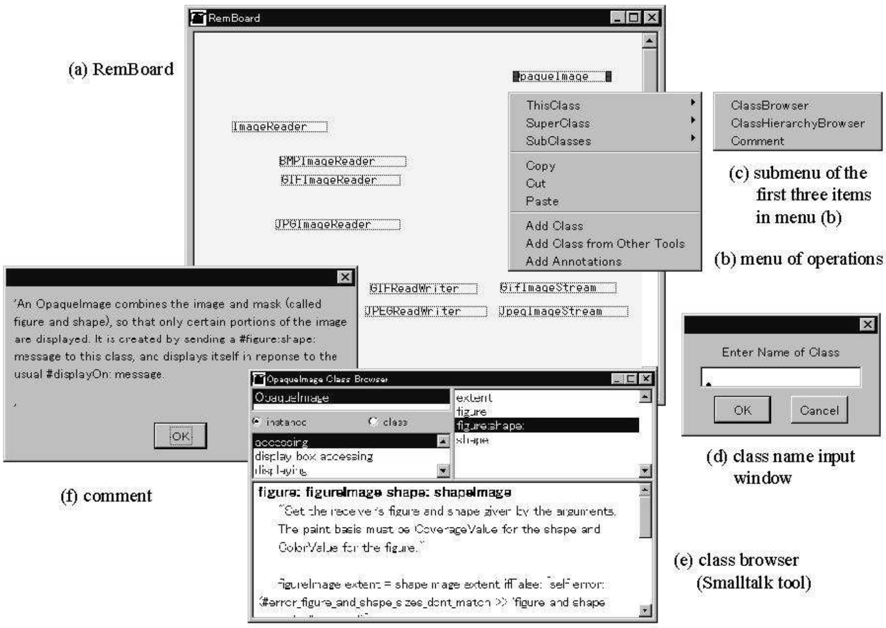

Fig. 2. RemBoard

RemBoard [10] (Fig. 2 (a)) is a tool for remembering software components that may become necessary while programming in the Smalltalk programming language. More precisely, the components that RemBoard handles are *classes*, although RemBoard can also handle annotations. Smalltalk, which is an object-oriented programming language, allows programmers to reuse classes from a large class library. For example, in the VisualWorks (Smalltalk) environment, programmers can exploit a library of more than 900 reusable classes, which helps increase program quality and productivity.

Finding “necessary” reusable classes from the large library, however, is challenging, making Smalltalk programming hard to learn for novice programmers. It is not easy to understand the whole class hierarchy, and having a hierarchy browser and keyword matching retrieval mechanisms are not enough for novices because it is difficult for them to understand what retrieved classes and methods "really mean" [11]. There is a need for a mechanism that allows Smalltalk programmers to remember intermediate search results as one cannot decide which classes and objects to use unless one fully understands their detailed behavior. RemBoard provides a representational medium for such programmers to remember what has been retrieved.

System Overview RemBoard is a tool that is added on top of the VisualWorks (Smalltalk) environment. RemBoard is a free two-dimensional space where programmers can place elements, which are classes. Various operations can be performed directly on elements displayed on RemBoard.

Users can put classes on RemBoard by using a window to directly input the class name (Fig. 2 (d)) or by copying from a Smalltalk tool, such as an editor (Fig. 2 (e)) or a tool showing search results. Recorded classes are shown as icons on the two-dimensional space with the class name as its label. Users can also place their own annotations on the two-dimensional space. Users can move, delete, or duplicate elements on RemBoard.

RemBoard also allows users to directly perform operations on iconized classes that are necessary for a programming task: for example, to open an editor (Fig. 2 (e)) on a class, or to show comments that were attached by the original programmer (Fig. 2 (f)).

Visualizing the Current Situation RemBoard uses a two-dimensional space to express relationships among the recorded classes. In the user studies [10], we have found that subjects used positioning of elements on RemBoard to visualize classes that are under consideration for the task, classes that were thought to be "related" to those classes, and the relationships among these classes.

In one situation, a user positioned a class below another class to show inheritance (the relationship between ImageReader and BMPImageReader in Fig. 2 (a)). This let the user visualize that BMPImageReader inherits various states and behavior from ImageReader, and could keep this in mind while continuing his task. In another situation, a class was placed far from other classes to show that it was not directly related to the other classes but was deemed important to remember (OpaqueImage in Fig. 2 (a)).

Interestingly, the above two situations were taking place simultaneously within a single two-dimensional space without causing any confusion for the subject. When the programmer is understanding the situation, the "meaning" of the positioning depends on what part of the space the programmer is looking at.

4 Discussion

We have compared writing and component-based programming with solving a jigsaw puzzle, and much like using a table to solve a puzzle, we used a two-dimensional space to visualize and understand the current situation, i.e. the two-dimensional space served as a means for "sketching".

There are a few important differences between an actual jigsaw puzzle and writing (or programming):

1. An actual jigsaw puzzle has all the pieces necessary to make the jigsaw puzzle. In writing and programming, not all pieces may exist. The writer (or programmer) will need to make new pieces as necessary.
2. An actual jigsaw puzzle will not have any unnecessary pieces. This is also not true with writing and programming. The writer (or programmer) may have pieces at hand, but they may actually not need it.
3. No changes are necessary for pieces in an actual jigsaw puzzle. Writers (or programmers) may need to make changes to parts of text (or software component) so that they will fit.
4. An actual jigsaw puzzle will have a picture of what the completed puzzle will look like. This also does not hold for writing and programming. There may be a conceptually top-level goal to work for, but there is rarely no detailed map from which to work with.
5. There will only be one solution for a jigsaw puzzle, but a document may be written in many ways, and there may be several components that can satisfy the requirements for a program.

In order to account for these, the writer or programmer will need to be able to:

- make new elements,
- delete elements,
- edit elements, and
- see what they currently have.

We further discuss these issues below. The discussion is augmented by results from user studies of the two tools.

4.1 Overview of the User Studies

In one ART user study [12], three subjects modified a collection of notes with no coherent structure that concerned the usage of instruments in a laboratory. The subjects' task was to make it into a coherent document. The initial "jigsaw puzzle piece" was one element holding all of the notes. In another ART user study [6], a subject wrote an actual grant proposal based on three email discussions. In this case, the initial jigsaw puzzle pieces were three elements, each corresponding to one email discussion.

In the RemBoard user study [10], three subjects modified a small address book application, so that the items displayed in the address book (such as names and addresses) would be aligned. The initial jigsaw puzzle pieces in the RemBoard user study was the four classes in the application. There was also the basic class library provided by the VisualWorks (Smalltalk) environment.

Overall, the two-dimensional space provided by ART was used more frequently than in RemBoard. This could be attributed to the fact that ART is a stand-alone tool while RemBoard is an add-on tool. The two-dimensional space in ART is the most integral part and the user has to use it to conduct his/her task[2]. On the other hand, RemBoard is an add-on tool and the programmer can write a program without using it.

4.2 Adding, deleting, and editing elements

The first three issues of adding, deleting, and editing elements were implemented in both ART and RemBoard. Although elements in both ART and RemBoard could be deleted by choosing the element, and then choosing "delete" or "cut" from a pop-up menu, the implementations of adding and editing elements were done differently.

ART elements are added and edited by using the ElementEditor and ElementsMap, i.e. these two functions are completely integrated within ART.

On the other hand, RemBoard elements are added by using a window to directly input the class name or by copying from a Smalltalk tool. Elements are edited by opening a Smalltalk editor. Since RemBoard is added on top of the VisualWorks (Smalltalk) environment, these two functions are done using the tools provided. This is also a reflection of RemBoard being a tool that supports the search for necessary components, and not a standalone programming environment.

There is another significant difference between ART and RemBoard. An ART element can be split into multiple elements, and multiple elements can be merged into one element. A corresponding operation does not exist in RemBoard. This is the result of the difference between the granularity of elements in ART and RemBoard. An ART element is a piece of text that the writer thinks of as "one", while a RemBoard element is a class. In the ART user studies, users were frequently found to split and merge elements into a granularity which they felt were more useful for their current situation.

4.3 Seeing what they have

The fourth issue of seeing what the writer or programmer has is most critical. The writer or programmer does not have a complete image of what the document or program will be like when finished. Furthermore, there is no one correct solution for a document or a program; a document may be written in different ways, and several different components may be available as options, each of which satisfies the same requirements. These points mean that understanding the current situation will have much more significance in writing and programming compared to solving a jigsaw puzzle; the writer or programmer will have to consider various options, each of which may be correct, but "correct" in different ways.

If the writer or programmer is not able to understand the current situation, he/she will not be able to reflect on it and the design process will not proceed

[2] It is of course possible to make one large element.

well. In other words, the externalization or visualization of the current situation did not go well, and the tool would not function as a "sketching" tool.

Expanding this further, the writer or programmer will need to be able to know the following three types of information:

- what elements they are currently considering,
- the characteristics of the elements they are considering, and
- the relationship between elements they are considering.

They will further need to have some freedom in being able to express the characteristics and relationships.

They should also be able to see the entirety. Because the designer needs to consider many disparate factors, they prefer to be able to view sketches without much head or eye movement [4]. This led to both ART and RemBoard having a "limited" two-dimensional space. It would have been possible to have the two-dimensional space be larger and have scroll bars, but this would have meant that the designer would need to search within the two-dimensional space.

We now consider each of the three types of information that the writer or programmer will need to know, i.e. those that need to be visualized.

Elements under consideration We first consider the need to be able to know what elements the writer or programmer is currently considering. We consider this from three viewpoints: labels, element size, and positioning.

Labelling is one way to differentiate elements, and thus be able to know which elements are under consideration. For example, labels of jigsaw puzzle pieces are fixed; it is the picture of the part it is supposed to show. Labels were handled differently for ART and RemBoard. Since each element in RemBoard is a class, the class name was used as a label. It is a definite precise name. In contrast, no such labelling is used in ART. What the writer sees is the first ten percent of the text in the element (unless the writer resizes the element). No explicit labelling mechanism was provided in ART because each element can be thought of differently according to the context. We have, however, observed some users making a pseudo-label by fitting the "label" in the first ten percent of the element's text.

Another way to know what elements are under consideration is by resizing the elements. Both ART elements and RemBoard elements are resizable. One ART user resized several elements so that they would "call for attention" later in the task. Another ART user had resized several elements to the same size to let him know that "those elements are finished". Interestingly, none of the RemBoard users in the user study resized an element for a similar purpose.

Finally, a third way to differentiate elements under consideration is through positioning. Users can position elements in a way so that elements placed in a certain area need work, and others not. One ART user put unelaborated elements at the bottom right corner of the ElementsMap. In RemBoard, a user put unnecessary elements in the top right corner[3].

[3] The user could have deleted the element but chose not to because he was "unsure if

Characteristics of elements Another type of information necessary to know the current situation is to know characteristics of the elements. Size and shape are two example characteristics of a jigsaw puzzle piece. These two characteristics can be used to figure out if one piece fits another. Unlike jigsaw puzzle pieces, ART and RemBoard elements may change, and it can be considered to be useful if such characteristics could be shown. We consider four types of characteristics: size, "last changed", complexity, and origin of element. Note that in a jigsaw puzzle, all four characteristics are fixed, i.e. puzzle pieces are of nearly the same size, they do not change, the complexity of pieces in the same puzzle are the same, and all pieces come from the same jigsaw puzzle package.

Both ART elements and RemBoard elements have *size* as element-specific characteristics. The type of size in ART is limited basically to number of characters and number of words. Although number of sentences is also possible, since an ART element does not necessarily need to be a sentence, it may have no meaning. Currently size in ART is visualized by having the element size be large enough to show the first ten percent of the element's text. In contrast, RemBoard elements have many types of size properties. They include number of methods, number of variables, and number of lines. These also could be visualized through element size.

The second characteristics *"last changed"* denote when that element was last modified. This can be implemented similarly for both ART and RemBoard. Color could be used to show which element was changed last, next to last, etc.

The third characteristics *complexity* can be used for RemBoard elements. Each RemBoard element, or Smalltalk class, belongs in an organized hierarchy, which lets classes inherit behaviors and states from other classes. Each class can also have a use (or used) relationship with other classes. These inheritance and use/used information can be used to compute the complexity of a class [2], and then visualized through means such as color or size. Although it may be possible to define a similar measure for ART elements, it would basically be a variation of the size characteristics.

The fourth characteristic *origin of element* concerns where the programmer (or writer) "found" the element. RemBoard is integrated within the Smalltalk programming environment which has an extensive class library, i.e. there is a large pool of components from which to choose. Although programmers will develop new classes, keeping track of what classes have been found and how it was found is important. Without such information, the programmer may not be able to, for example, redo part of a keyword search. Thus, RemBoard elements have a feature where the programmer can see how it was put on the RemBoard. For example, if keyword search was used, the list of keywords that were used to retrieve that class is shown as part of the label. Because programmers have this information, they can go back and try other combinations of keywords. On the other hand, there are no document database connected to ART, and no information is stored (or visualized) concerning how an element was produced. Although writers can use documents from previous documents that they have

---

it really was unnecessary".

used, much of the text is written anew, especially when compared to RemBoard elements. In fact, no ART users have ever expressed an interest in a feature that would let them know how a certain element was produced.

The above characteristics are of course not exhaustive and other characteristics could be beneficial. For example, Yacoub et. al have proposed a set of features characterizing software components [14].

Relationship between elements Users will want to view many types of relationships between elements. In jigsaw puzzles, the key relationship is its location compared to the final product. Thus, people will often put puzzle pieces where they think that piece will go, especially if it is a side or corner piece.

Relationships between elements in ART and RemBoard are shown through the positioning in the two-dimensional space. We have already given two examples of global relationships that were observed for ART and RemBoard users: an ART user put unelaborated elements at the bottom right corner of the ElementsMap, and a RemBoard user put unnecessary elements in the top right corner. We now elaborate on other types of relationships.

Between ART and RemBoard, there is only one fixed relationship: the vertical (or top-bottom) orientation in ART. The top-to-bottom ordering corresponds with the order that the elements are concatenated to obtain the document (which is shown in the DocumentViewer). This can be considered to be a natural constraint, as a written document is read from top to bottom. None of the ART users have ever expressed a dissatisfaction with this fixed orientation.

Because of this top-bottom constraint, ART users are basically left with the horizontal (or left-right) orientation to show various relationships. Some of the relationships that were observed were:

- enumeration/itemization: A user positioned elements at the same vertical position, but used horizontal positioning to show that they were items of a group. Interestingly, the same user used the vertical relation to show the same thing in a different situation. She used a different way of visualizing the same relationship.
- need for other elements: A user put some space between two elements to show that he "needed to write something".
- "closeness": All users tended to close the space between elements if he/she thought that those elements were closely related in some way.

RemBoard users expressed relationships such as the following:

- inheritance: A user placed one element below and to the right of another class; for example, BMPImageReader and ImageReader in Fig 2 (a).
- use/used: A user placed elements next to each other to show that one class used the other class. Interestingly, this was not fixed to just vertical or horizontal positioning. A user would use vertical positioning for one task but horizontal for another. Also, the user was able to remember right away which class was using the other class, without their being an explicit annotation describing it.

- alternative: A user aligned alternatives; for example, GIFReadWriter and JPEGReadWriter is one alternative and GifImageStream and JpegImageStream is another alternative in Fig 2 (a).
- important but not related: When a user found a class that he thought to be important but could not define its relation with other classes already on RemBoard, he placed it far away from those already on RemBoard; for example, OpaqueImage in Fig. 2 (a).

Although this was not observed, other relationships could also be possible, such as aggregation which in object-oriented programming means part-whole relationship.

4.4 Sketching and Two-Dimensional Positioning

As we discussed above, the two tools served as a means for the writer or programmer to visualize their current situation. Two-dimensional positioning was a critical component as it allowed the users to express relationships and characteristics of the elements. As with sketching, the users had freedom in how they could express or visualize various situations – except for the vertical relationship in ART, they were free to position elements in any manner they want.

Through visualizing the current situation, the users were able to reflect and consider what the next step should be. For example, the following situations were seen in the user studies:

- An ART user continually moved an element until it reached a state that the user thought "correctly" showed the relationship between it and other elements, i.e. she used the visualization to reflect on the design [6].
- A RemBoard user put on RemBoard any classes which he thought might be of use later. This helped him visualize what possible combinations might be possible in his programming task [10].

These observations from the user studies suggest that the two-dimensional positioning served the same purpose as sketches in the architectural domain.

5 Conclusion

We took writing and programming as two example domains, and described two tools – ART for writing and RemBoard for component-based programming – that supports the writing or programming task. The key idea is that two-dimensional positioning can serve the same purpose in these two domains, much like sketches do in the architectural domain, i.e. they serve to visualize what the designer is thinking. We made references to their similarity with solving a jigsaw puzzle, and discussed how visualization occurred.

There are various ways that a user can visualize a certain train of thought. We need to keep this in mind: certain visualization may actually hinder the user's task. A mapping that is "natural" from the user's viewpoint is critical in any type of sketching tool. From this viewpoint, the two tools let users freely position and resize elements; the users are the ones that map a certain situation or thought to a certain representation. The power of attaching meaning to a representation is a powerful one and should be exploited to its fullest, just like sketching in architecture, ART for writing and RemBoard for component-based programming.

References

1. Bruner, J.: The Culture of Education. Harvard University Press, Cambridge, MA. (1996).
2. Chidamber, S. and Kemerer, C.: A Metrics Suite for Object Oriented Design. IEEE Trans. on Software Engineering. 20(6) (1994) 476–493.
3. Kintsch, W. and van Dijk, T.: Towards a model of text comprehension and production. Psychological Review. 85(5) (1978) 363–394.
4. Lawson, B.: Designing with drawings. How Designers Think: The Design Process Demystified. Architectural Press, Oxford, MA. Chapter 14 (1997) 241–259.
5. McKinney, D.: Impact of Commercial Off-The-Shelf (COTS) Software on the Interface Between Systems and Software Engineering. Proc. of 1999 International Conference on Software Engineering. (1999) 627-628.
6. Nakakoji, K., Yamamoto, Y., Takada, S., Reeves, B.: Two-Dimensional Positioning as a Means for Reflection in Design. Conference on Designing Interactive Systems (accepted).
7. Schoen, D.A.: The Reflective Practitioner: How Professionals Think in Action. Basic Books, New York (1983).
8. Shipman, F.M., Marshall, C.C., and Moran, T.P.: Finding and Using Implicit Structure in Human-Organized Spatial Layouts of Information. Human Factors in Computing System (CHI'95). (1995) 346–353.
9. Simon, H.A.: The Sciences of the Artificial (Third ed.). MIT Press, Cambridge, MA (1996).
10. Takada, S., Nakakoji, K., and Torii, K.: Using 2D Space for Understanding What "Search Options" We Have Taken in Exploring a Class Library. Technical Report ccc-98-9, Cognitive Science Lab, NAIST (1998).
11. Takada, S., Otsuka, Y., Nakakoji, K. and Torii, K.: Strategies for Seeking Reusable Components in Smalltalk. Proc. of 5th International Conf. on Software Reuse. (1998) 66–74.
12. Yamamoto, Y., Takada, S., and Nakakoji, K.: Representational Talkback: An Approach to Support Writing as Design. Proc. of 3rd Asia-Pacific Computer Human Interaction Conf. (1998) 125–131.
13. Williams, N.: New Technology, New Writing, New Problems? Intellect Books, England (1992).
14. Yacoub, S., Ammar, H. and Mili, A.: Characterizing a Software Component. Proc of International Workshop on Component-Based Software Engineering. (1999) 151–158.