Design Issues of Visual Languages for Supporting Software Evolution

Etsuya Shibayama
Graduate School of Information Science and Engineering
Tokyo Institute of Technology
etsuya@is.titech.ac.jp

Masashi Toyoda
Institute of Industrial Science
The University of Tokyo
mtoyoda@acm.org

Buntarou Shizuki
Institute of Information Sciences and Electronics
University of Tsukuba
shizuki@is.tsukuba.ac.jp

Shin Takahashi
Graduate School of Information Science and Engineering
Tokyo Institute of Technology
shin@is.titech.ac.jp

Abstract

We describe issues in design of visual languages that can provide a support for software evolution and lessons learned from our experience in development of a visual language environment KLIEG. In this paper, we put emphasis on general frameworks for visual syntax, interaction supports, and aspect visualization. We also describe several specific techniques and design decisions of KLIEG.

1. Introduction

As its name represents, software is inherently *soft*. This very nature is without doubt prerequisite for evolution. However, solely being soft is not sufficient for successful evolution. Legacy software cannot evolve not because it is *hard* but because it is not comprehensible and thus it is not modifiable in a favorable direction. Everyone knows that it is often difficult to design and implement flexible and evolvable software, even though evolution is the *raison d'être* of software. We certainly need some technologies to fill the gap between ideals and reality.

Studies of Darwinian evolutions teach us some lessons of evolutions. First of all, living lives on the earth have stable structures that are immutable during a series of evolutions. Even though they have astonishing varieties, most of them have the same genetic coding systems. This means that their core architecture is incredibly stable and has not changed for a few billions of years. Some structures are less stable but still persist over millions of years. Skeletal structures of animals are such examples. Computer software also requires stable structures that will guarantee the internal consistency in some level. Solely being soft almost always brings about chaotic results. Another lesson is that a small change in genotypes can be the cause of a drastic change in phenotypes (e.g., [5]). Software should also be constructed so that small changes in implementation can provide new and rich functionality. Changing many portions at a time in a consistent manner is always a hard job.

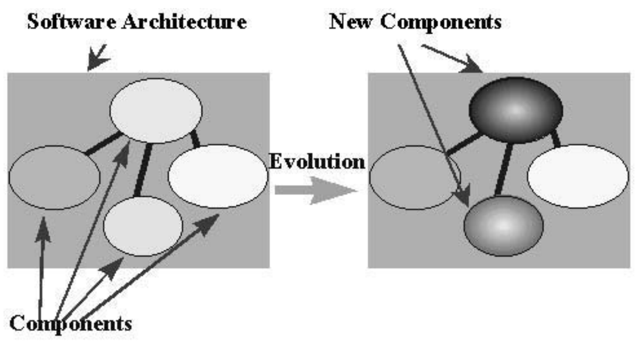

Figure 1. Software evolution by replacing components

We assume in this paper that software that can evolve has its own stable architecture and replaceable components (Figure 1). Software evolution will proceed by the following manner.

1. recognition of the environmental change,
2. comprehension of software and spotting the components to be replaced, and
3. replacement of the components.

Note that our model is hierarchical in the sense that a component can have its internal structure consisting of finer architecture and replaceable components. We have been concentrating most of our efforts on support of the second point. Proposing a mechanism or interactive support for the third point is easy and the first point is rather difficult.

We have been developing a visual language environment KLIEG[18, 13, 14, 15] that can provide a support for evolution of software in several phases of development cycles. In this paper we will describe philosophical background in design of the KLIEG language and also more general issues and problems in supporting software evolutions by visualization and interaction techniques.

Our discussion in this paper covers visual language syntax, browsing and editing support, the importance of aspect visualization, and incorporation of designers' intensions. We present our general framework based upon hierarchical visual language syntax and interactive zooming interfaces. Our visual syntax is general and expressive enough for representing programs, designs, and execution in a comprehensible manner. Our interaction techniques with zooming mechanisms provide not only scalable solutions but also support for visual abstractions. We consider it important to provide standard notations with standard interaction techniques for protecting users' investment in learning languages and tools. In addition, in this paper we would like to shed light on supporting aspects of programs. Aspects are important in some particular situations but less ubiquitous than language supported abstractions. We describe our approaches to visualization of ad hoc aspects.

In the following, we begin our discussion with syntax and presentation issues of (visual) programming languages in Section 2. We describe the importance of visualization and interaction techniques for providing and presenting foci and aspects of software in Section 3. We also discuss a way to represent extra-linguistic design information such as designer's intensions.

2 Syntax and presentation issues

People may consider that syntax and presentation styles of programming languages are merely a surface issue and more significant problems are concealed in a deeper level, e.g., in semantics. However, a presentational structure itself is important for human designers and/or programmers. Suppose for instance that your team members wrote codes in C or Java with exotic indentations and ciphering names of identifiers. Even if a computer would perfectly recognize and execute their codes, you would sooner or later have a nightmare.

In this section, we describe general syntax and interaction framework of visual languages. As B. Harrison pointed out in [2], tool skills are significant. It is often easier to change programming languages than to editors. This is the very reason why we consider that general interaction framework covering a wide range of visual languages is necessary. Though our background knowledge and experience heavily depend on design and development of KLIEG, we believe that our discussion is valid in more general settings and informative for those people who will design scalable visual notations or languages.

2.1 Strings, trees, and hyperlinks

In theory, a formal language is a set of (one-dimensional or linear) strings. This linear representation is useful for data exchange by computers but is not necessarily appropriate for human to understand programs. For human beings, more comprehensible representations are necessary.

Looking back to the structured programming era, the standard programming notation was a two-dimensional character layout with indentations, in which trees or hierarchical structures can naturally be represented in a reasonably comprehensible manner. As far as data and control structures are naturally regarded as trees, this presentation style is effective. Even today this old wisdom is still there.

Everyone knows, however, that today's object-oriented software consists of an organized collection of interrelated objects. Its natural shape is a directed graph and cannot be reduced to linear strings or trees without loss of comprehensibility. Coping with this trend, in the software engineering community, UML and other visual notations have been proposed for representing various aspects of object-oriented software. An essential advantage of a visual notation is that it can provide more freedom of layout in 2D or 3D spaces and is expressive enough to represent relationships among interrelated objects.

Our language syntax follows this visual line. However, we do not employ general directed graphs as the basic structures. Instead, we put more attention on hierarchical structures with hyperlinks by the following reasons:

- They are expressive. A hyperlink can certainly be a substitute of a directed edge of a graph. Hierarchical structures can naturally represent layered abstractions, which are essential in today's programming and/or design languages.
- Hierarchical structures are ubiquitous and thus shall be

treated with its own worth. Note that they are often the key to scalable solutions in various domains.

- Hierarchical structures and hyperlinks can be easily mapped into XML, the standard exchange format in the foreseeable future.

Summing up, they are not only expressive but also structured and at least potentially interoperable. Since we know the limit of human cognitive power, we often sacrifice expressiveness for order and simplicity. We are also interested in protection of users' investment.

2.2 Visual syntax

In the literature, various visualization techniques have been proposed to depict hierarchical structures on a computer screen. Examples include ConeTree[12], Information Cube[11], hyperbolic browser[8], the continuous zoom[1], and so on. Those techniques more or less tackled on the screen real estate problem. ConeTree depends on the extent of 3D space and the other on zooming techniques.

Since our visual syntax shall be expressive enough to represent software design, programs, and execution, we put strong emphasis on visualizing abstractions. "Exposing everything if possible" is not our goal. Abstractions are more related to the notions of hiding and encapsulation. Fortunately, zooming techniques are promising not only for scalable visualization but also for representing *hiding*. That is, by zooming, while some portions are magnified, the others are cut down and effectively hidden.

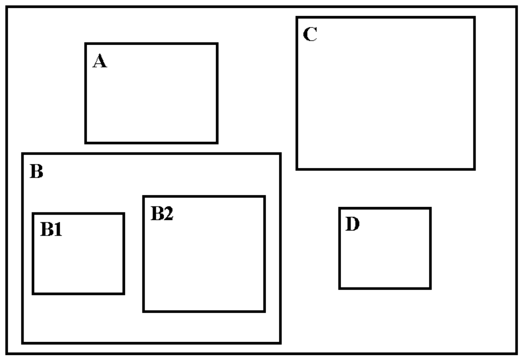

Figure 2. Depicting a hierarchy by nested rectangles

After careful consideration, we have followed the continuous zoom and have employed nested rectangles to represent a hierarchy (Figure 2). Each rectangle corresponds to a node in the hierarchy and the parent-child relation is represented by geometrical inclusion. For instance, in Figure 2, B has two children B1 and B2 and three siblings A, C, and D. Hyperlinks would be illustrated as arrows, though they are omitted in this figure.

Major uses of hierarchies in our problem domain are representing:

- syntax trees,
- call graphs,
- composite or aggregate objects, and
- directories.

In any case, a parent node is representative of the sub-hierarchy consisting of its descendants. This means that outline editing, in which a sub-hierarchy may be folded into its root node, is promising. In fact, outline editing has proven to be useful by many years of experiences, and is employed by text editors, e.g., Emacs, and word processors, e.g., MS Word. We consider that nested rectangles (or circles, triangles, etc.) are most suitable for outlining in 2D spaces. We did not employ single focus zooming techniques including hyperbolic browser and Information Cube since in outline editing two or more focal points are often necessary.

2.3 Browsing and layout supports

Again in theory, a language has been considered as a set of static entities. However, visual languages are interactive. This is another source of the power. We can incorporate syntax-directed supports, e.g., those for browsing and editing, into a language itself. In addition, providing not only general syntax framework but also general interaction methods for syntax-directed manipulations, we hope the burden to change languages are reduced.

Since our target hierarchies are too large to fit in a computer screen, browsing supports are inevitable. In a focus+context view like ours, browsing in a general sense is to create a view, in which nodes of interest are magnified, and possibly to adjust its layout for comprehensibility.

Obviously we need methods to specify the nodes of interest. The following are popular ways to specify the destination(s) of browsing.

- Navigation to the parent or a child
- Navigation via a hyperlink
- Search or query

The first two are navigation via a link. If the reader is too familiar with an HTML browser such as Internet Explorer or Netscape Navigator, she or he may consider that there are no differences between those two types of link navigation. However, they are significantly different in our context.

The most fundamental and important requirement is that every node be specifiable, either directly or indirectly. Otherwise we would have unreachable nodes. This means that the target structures should be strongly connected. Thanks to the hierarchical structures, the first sort of navigation is sufficient to reach any node. The other two might be considered as short cuts.

Navigation to the parent is very simple in our visual syntax. As is obvious from Figure 2, whenever the source node is visible (or sufficiently large), its parent is also visible (or larger than any one of its children including the source node). Therefore, we have nothing to do. Navigation to any ancestor is easy, too. Navigation to a child is also simple. It is just to magnify a node whose enclosing parent is already visible.

Navigation via a hyperlink is considerably different. In general, after navigation, the current focus on the source node may or may not disappear. Suppose for instance that a hyperlink from a function call expression to the definition of the function is created. Upon navigation of the hyperlink it seems reasonable to keep the focus on the source node since a user often would like to see both the definition and a use of a function during editing a program. However, if the screen is already full of foci, the user may hope to clean up obsolete ones. With an extremely flexible and expressive zooming interface like ours, it is often tedious to specify the foci and their degrees of interest by manually adjusting layout. Unfortunately, more freedom always means more responsibility.

In order to alleviate this problem, we proposed a technique, in which the runtime system manages a three-state automaton for each node and predicts whether or not a focus is on the node. This simple predictor is effective partly because its prediction accuracy for novice users is about 80%[16]. In addition, the accuracy for expert users is almost 100%. Any human user can soon learn the behavior of simple three-state automata perfectly by experience. Mutual understanding between a user and a supportive environment is often significant.

Keyword search is another means to specify destinations and popular in text editing. Structured queries specifying link structures may be useful in visual languages.

2.4 Implementation and applications

We have implemented a zooming library HyperMochiSheet[16] for building applications that provide a support for browsing and editing hierarchical structures with hyperlinks in the visual syntax illustrated as in Figure 2. The most intriguing feature of this library is its focus prediction and a support for layout management mentioned in the previous subsection.

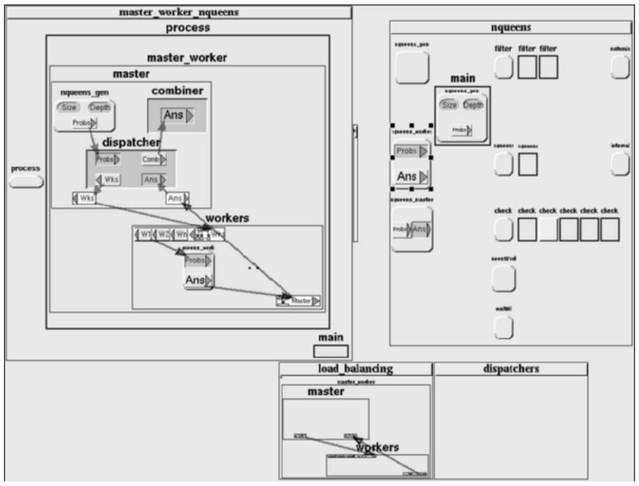

Figure 3. Klieg

On top of HyperMochiSheet, we have developed several tools. The most noticeable is the KLIEG environment. Figure 3 is a snapshot of the KLIEG program editor. Each top-level rectangle represents a module, in which object definitions are included. In this figure, there are five modules of various display sizes, one on the top left, another on the top right, two on the bottom, and a thin one right adjacent to the top left one. It might be difficult to capture the last one.

Each module is naturally regarded as a hierarchy, which corresponds to a syntax tree of a textual programming language. We omit the details of KLIEG visual syntax since they are irrelevant in this section.

KLIEG is a dataflow visual language and so the arrows in the figure represent dataflow links, which are implemented by hyperlinks in our framework. Also a number of invisible hyperlinks are automatically generated, e.g., between the uses and the definition of each object. Note that an object definition in KLIEG corresponds to a class definition of mainstream object-oriented languages. Objects in our sense are closed to those in prototype-based languages[9, 19].

One interesting point in representing abstractions by zooming is that they can be continuous. In contrast, in a traditional setting, each entity in a hierarchy is either exposed or hidden. With continuous representations of abstractions, a property of an analog value can be mapped to an appearance of a node. For instance, in response to a search request, our directory browser[17] maps the relevance factor of a node to its display size. For instance, the view in Figure 4 represents the directory hierarchy of Java 2 SE library source files, with magnifying every file in the java package that contains occurrences of the word "EventListener." This browser is also implemented on top of HyperMochiSheet.

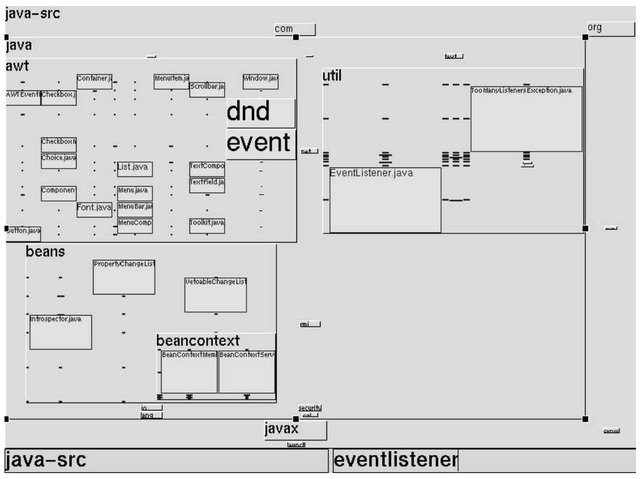

Figure 4. A directory browser

In this example, the more a node is promising, the more it is exposed. On the other side of the coin, less important nodes may be aggressively cut down. A translucent display technique[7] may have similar continuous effects by mapping the degree of exposure to the degree of transparency. Without zooming, however, a translucent technique is not scalable.

3 Aspect visualization

Next to visual syntax and browsing supports, we propose a general framework for visualization of aspects. Since both screen sizes and our cognitive abilities are severely limited, we cannot look at all the details of design, implementation, or execution at a time, unless the target software is exceptionally small. Therefore, for software comprehension, we need appropriate aspects that are small enough to fit in a computer screen.

There are various aspects. Some may consist of organized collections of interrelated objects, methods, variables, and so on that are scattered among design, implementation, or execution. Program slicing is an example and the method definitions of a same method name may be another. A tracer of Lisp or a functional language often produces information about the calls and returns of a function. The collection of the corresponding portions of the call graph is yet another aspect. An instance of a design pattern[4, 10] can also be regarded as an aspect. However, a design pattern is not merely a collection of syntactic entities. It includes design information or designer's intensions. Visualizing such information is still a challenge.

3.1 A framework for aspect visualization

Corresponding to various sorts of aspects, there are various techniques for exploiting and visualizing them. However, the major issue of this subsection is a general framework for aspect visualization. We do not mention any domain specific techniques for exploitation of aspects.

Our visualization framework is based upon the visual syntax and interaction techniques proposed in the previous section. We assume that an aspect in this section consists of nodes in a hierarchy that are possibly scattered to separate sub-hierarchies. Since in our syntactic framework a node is usually a unit of language abstraction, an aspect in this sense is not an abstraction directly supported by the language but a collection of interrelated language abstractions.

On the one hand, language abstractions are persistent and directly supported by language processors and/or environments. However, they are limited in numbers and sorts. On the other hand, some aspects are short-lived but very useful in limited situations. The language designer cannot accept every possible aspect as a built-in abstraction. With a general framework for manipulating aspects, it should be possible to visualize an ad hoc aspect as if it were a first-class abstraction. We should provide an environment that can cooperate with a user and create and visualize personal abstractions. This is the ultimate goal of our proposal.

Proposing a concrete solution, we assume that the following are the only parameters for visualizing an individual node:

- Size
- Location
- Texture

As far as visual syntax like Figure 2 is concerned, the size, location, and label are the only parameters of a node. The first two parameters determine the layout of a view and the last one the appearance of each node. Controlling layout and appearance with those parameters is the key idea of this model and it is sufficiently general.

In the multi-focus zooming interface like ours, the size of a node is influenced by its degree of interest (DOI), which is usually supplied by an exploitation algorithm of aspects. The way to determine its location is less clear. HyperMochiSheet calculates the geometry of each node under the assumption that each node has its own home position relative to its siblings and within its parent. This assumption is valid in visual languages since descriptions in visual languages include layout information, which are usually provided by human designers and/or programmers. In sum, if we use visual languages from the beginning, the only extra information necessary for layout is DOIs.

The zooming algorithms in an ordinary sense are irrelevant to the last parameter. Still, however, with the mechanism of semantic zooming[3], the appearance of a node may be controlled somehow. For the purposes, of course, we have to prepare mapping rules and so this part might be rather ad hoc.

3.2 Design patterns and aspects

It is a difficult question whether design patterns should be language abstractions or mere aspects. They are not language abstractions in today's typical implementation languages such as C++ and Java. The core abstractions of those languages are classes and objects. An inter-class or inter-object relation is extra-linguistic. Design languages like UML capture more about inter-class and inter-object relations. Still, however, they do not provide every possible sort of pattern as a built-in abstraction.

In any case, effective visualization of a design pattern shall display the following information:

- Structural information among classes or objects in the pattern, and
- Design information or designer's intensions.

In these days, language designers sometimes propose new abstractions for representing the former information explicitly but the real problem is lack of the latter information.

Taking account of this point and the nature of design patterns, we have proposed a mixed solution. That is, a design pattern is an inter-object abstraction explicitly supported by a language and one pattern may have more than one design related aspect[18].

If design patterns were mere aspects and were not supported by languages, we would need pattern-mining algorithms. However, exploitation of design information without any help of the pattern designer is hopelessly difficult. We consider that the only feasible solution is to ask designers to provide necessary information. For the purpose, programming environments should be able to recognize patterns as persistent abstractions and explicitly keep track of the provided information.

We restrict the object structure of a pattern to the visual syntax introduced in Section 2. Objects in a pattern may or may not be abstract. An abstract object must be instantiated with concrete objects before runtime.

Figure 5 is an example definition of a pattern in KLIEG. This figure represents a master-workers object network, in which a single master dispatches tasks to multiple worker objects and gathers the results of the workers' computations. The master part consists of three objects, generator, combiner, and dispatcher. The workers part consists of multiple workers.

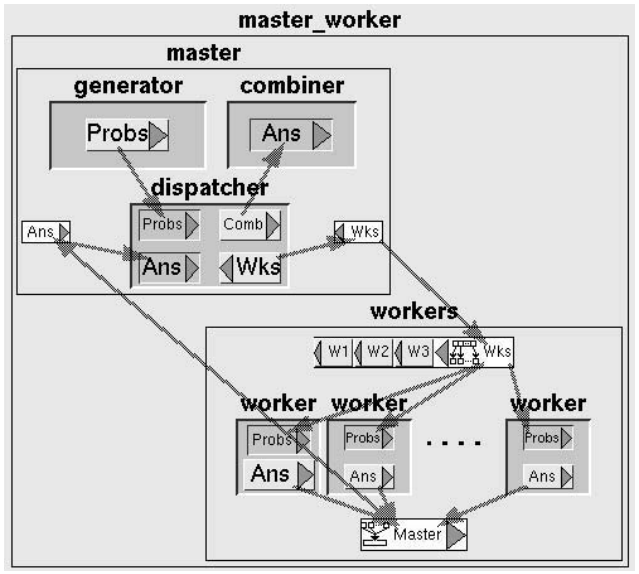

Figure 5. Master-workers pattern

In visual languages, it is not difficult to illustrate this sort of skeletal structure. We have also implemented interaction mechanisms to replace components[18] but they are not an issue of this paper.

Upon reuse of software that is constructed from patterns, the following information is desirable:

- Which components are replaced?
- What are their alternatives?

The answers depend on the functionality or behavior that changes via evolution. We consider that different views or aspects are necessary for different purposes. That is, (an instance of) a design pattern is not an aspect but may have multiple aspects, corresponding to multiple types of evolution.

Figure 6 depicts two aspects of the master-worker pattern. The Top view, which puts emphasis on the generator and workers, presents an aspect for answering the first question when the problem domain will change. In such a case, both the way to partition a given problem and the way to solve sub-problems should change but other portions, e.g., the dispatching algorithm, can survive. In the figure, those portions that should be replaced are magnified. As for the second question, we propose to create hyperlinks between abstract object and concrete candidates. Similar to the top view, the bottom one emphasizes the dispatcher and combiner[1].

Currently, pattern designers must provide this sort of layout information for each aspect. As is already described, as-

[1]"Combiner" is a formal parameter name. In Figure 6, it is replaced with an actual object name "pass_answers."

pects of design patterns include designers' intensions and it seems difficult (even if it is not impossible) to exploit those in an automatic manner.

3.3 Concerns and Aspects

Figure 6 reveals another debatable question: Is a meta-level an aspect? The master-workers pattern is designed to provide an automatic load balancing mechanism. It might be natural to put such a mechanism on a meta-level. In some sense, the two views in Figure 6 emphasize base level and meta-level, respectively. On the top view, the generator and workers, which are problem dependent and thus can be regarded as base level objects, are emphasized. On the bottom view, the dispatcher, which is problem independent and thus can be regarded as a meta-level object, is emphasized.

The essential role of a meta-level is to implement *separation of concerns*. The two views in Figure 6 represent different concerns and these concerns are visually separated into two figures. However, they are two sides of the same coin. Even if we can separate objects, we cannot separate the architecture in our model. In addition, one may prefer a mixed view like Figure 5 or some other views that share visible portions with the top and/or bottom view in Figure 6.

In this respect, our aspects are different from those of AOP (Aspect-Oriented Programming)[6]. Aspects, in our sense, of the same software inevitably have shared portions. Since our motivation is to provide better views for software comprehension, this is not the problem. As mentioned in Section 1, with referring to biological evolution, we believe that just a small number of hotspots can change during a successful evolution. Our mission is to create a comprehensible view including such hotspots and any relevant information.

4 Conclusion

In this paper, we propose general frameworks of the following:

- Visual syntax that is expressive, structured, and interoperable
- Multi-focus zooming interface and its browsing supports
- Visualization of various aspects

These frameworks are designed by our experience in design and implementation of the visual language environment KLIEG. We consider that they are useful for development of a wide range of visual language systems that are scalable and provide a support for building comprehensible software.

References

[1] L. Bartram, A. Ho, J. Dill, and F. Henigman. The continuous zoom: A constrained fisheye technique for viewing and navigating large information spaces. In *Proceedings of the 8th ACM symposium on User Interface Software and Technology*, pages 207–215, 1995.

[2] C. Chambers, B. Harrison, and J. Vlissides. A debate on language and tool support for design patterns. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 277–289, 2000.

[3] K. M. Fairchild, S. E. Poltrock, and G. W. Furnas. Semnet: Three-dimensional graphic representation of large knowledge bases. In *Cognitive Science and Its Applications for Human-Computer Interaction*, pages 201–233. Lawrence Erlbaum Associates, 1988.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[5] S. J. Gould. *The Panda's Thumb: More Reflections in Natural History*. W.W. Norton & Company, 1980.

[6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–244. Springer-Verlag, 1997.

[7] A. Kramer. Translucent patches - dissolving windows -. In *Proceedings of User Interface Software and Technology*, pages 121–130. ACM, 1994.

[8] J. Lamping, R. Rao, and P. Pirolli. A focus+context technique based on hyperbolic geometry for visualizing large hierarchies. In *Conference Proceedings on Human factors in Computing Systems*, pages 401–408. ACM, 1995.

[9] H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 214–223, 1986.

[10] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1994.

[11] J. Rekimoto and M. Green. The information cube: Using transparency in 3D information visualization. In *Proceedings of the Third Annual Workshop on Information Technologies & Systems*, pages 125–132, 1993.

[12] G. G. Robertson, J. D. Mackinlay, and S. K. Card. Cone trees: animated 3D visualizations of hierarchical information. In *Proceedings of Conference on Human Factors and Computing Systems*, pages 189–194. ACM, 1991.

[13] E. Shibayama, M. Toyoda, B. Shizuki, and S. Takahashi. Visual abstractions for object-based parallel computing. In *Object-Oriented Parallel and Distributed Programming*, pages 113–132. Hermes Science Publications, 2000.

[14] B. Shizuki, M. Toyoda, E. Shibayama, and S. Takahashi. Visual patterns + multi-focus fisheye view: An automatic scalable visualization technique of data-flow visual program execution. In *Proceedings of IEEE Symposium on Visual Languages*, pages 270–279, 1998.

[15] B. Shizuki, M. Toyoda, E. Shibayama, and S. Takahashi. Smart browsing amount multiple aspects of data-flow visual program execution, using visual patterns and multi-focus fisheye views. *Journal of Visual Languages & Computing*, 11(5):529–548, 2000.

[16] M. Toyoda and E. Shibayama. Hyper mochi sheet: A predictive focusing interface for navigating and editing nested networks through a multi-focus distortion-oriented view. In *Proceedings of ACM CHI'99 Conference on Human Factors in Computing Systems*, pages 504–511, 1999.

[17] M. Toyoda and E. Shibayama. Hishimochi: A zooming browser for hierarchically clustered documents. In *ACM CHI2000 Extended Abstracts*, pages 28–29, 2000.

[18] M. Toyoda, B. Shizuki, S. Takahashi, S. Matsuoka, and E. Shibayama. Supporting design patterns in a visual parallel data-flow programming environment. In *Proceedings of IEEE Symposium on Visual Languages*, pages 76–83, 1997.

[19] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 227–242, 1987.

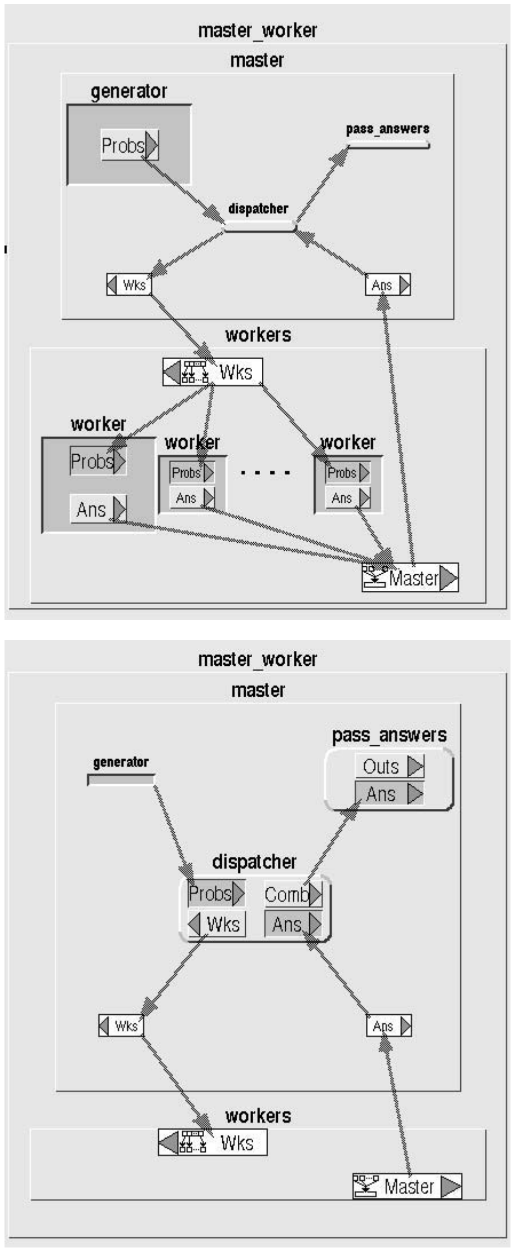

Figure 6. Two aspects of the master-workers pattern