Aspect Browser: Tool Support for Managing Dispersed Aspects*

William G. Griswold[+] Yoshikiyo Kato[†] Jimmy J. Yuan[+]

[+]Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114
{wgg,jyuan}@cs.ucsd.edu

[†]Department of Aeronautics and Astronautics
University of Tokyo
7-3-1 Hongo Bunkyo-ku, Tokyo 113-8656, Japan
yoshi@ai.rcast.u-tokyo.ac.jp

1 Introduction

Over time, requirements for a software system evolve to meet changing user needs or compete in the marketplace. Consequently, software maintenance and evolution are the dominant activities in the software lifecycle. Designing for change—a major tenet of software design—has been cast as a problem of designing software so that it can be extended, replaced in small pieces, or locally changed, rather than globally changed [5, 6]. Locality of change through information hiding is pursued because global changes are hard to reason about and can require the coordination of all the developers who have expertise with the software involved.

Although modularization, if used properly, separates the concerns of primary design decisions, it often fails to cost-effectively separate lower-order design decisions [8]. These lower-order decisions may cross-cut the primary module structure for at least two reasons. One, the programming language used provides behavioral abstraction (e.g., procedures and objects), whereas many lower-order design decisions are not describable as executable behaviors [3]. Two, the lower-order decisions may be emergent—that is, they were not recognized as being important during the design of the original system. Either way, changes to these cross-cutting design decisions tend to be more costly since they are dispersed throughout the system and tangled with the primary design decisions and each other.

New programming language features have been proposed for explicitly separating secondary design decisions (here called *aspects*), via new module-like constructs. Proposed features include reflective capabilities, event mechanisms, and customizable composition rules [3, 4, 8]. With the introduction of such features, two questions arise. One, how can aspects be managed in older programs written in languages that do not have aspect features? Two, since aspects emerge over time as requirements evolve, how can existing code be evolved to use aspect features?

We have directly experienced the emergence of cross-cutting aspects in the construction of several programming tools, despite the use of modern techniques and languages. However, we found that these breakdowns in modularity were not especially disruptive. We overcame these breakdowns in modular design by exploiting the textual and syntactic similarity of the pieces of code that had to be modified. The similarity permitted the programmer to use software searching tools to quickly and precisely identify the dispersed elements and to quickly visit them, with one or more passes to plan out the change and then one or more passes to perform the change.

These successes suggest a complement to information hiding modularity called *information transparency*. When the code relating to a particular change is not localized to a module, an information-transparent software design allows a programmer to use available software tools to economically identify and quickly view the related code, easing the change. That is, the "signature" of the changing design decision can be used to approximate the benefits of locality, in particular providing a way to quickly view and compare the elements of the cross-cutting aspect without distraction from inessential details. This signature is one or more shared characteristics of the code to be changed, such as the use of particular variables, data structures, language features, or system resources. Since the intrinsic characteristics of a

*This research was supported in part by NSF grants CCR-9508745 and CCR-9970985, and in part by UC MICRO grant 98-054 with Raytheon Systems Company, and took place in part while the first author was on sabbatical with the AOP group at Xerox PARC.

design decision can be incidentally shared by unrelated code, it is helpful if the programmer has adopted distinguishing conventions such as stylized naming of identifiers.

Given the ability to use design decision signatures to denote aspects in code, a software designer can effectively describe non-hierarchical, interwoven, cross-cutting module structures. When combined with information-hiding modularity, then, a designer can effectively represent more complex modular designs and manage a greater number of design decisions, thereby increasing the maintainability of the software.

Describing and changing systems using these techniques currently requires considerable discipline and concentration on the part of the programmer, in large part because most current tools provide basic mechanisms for finding and visiting code the relevant code, but do not address all the problems that modularity addresses. In particular, a well-designed module localizes code to well-defined boundaries possessing an abstract interface. Localization provides virtually instantaneous inspection of any piece of code relevant to the module's purpose, avoiding programming mistakes relating to failure of short-term memory; the boundary makes it clear what code is part of the module and which is not, ensuring that all and only the code of concern is considered; the abstract interface provides a clean separation of the module's code and its use elsewhere in the system, reducing the chance of rippling changes. Together these provide superior support for achieving complete and consistent changes in the absence of cross-cuts.

In contract, the ability of tools to rapidly identify and visit cross-cutting code provides a level of instantaneity and completeness, but: the programmer has to explicitly scope the activity through identification and visitation, the related elements cannot be viewed both together and in their context of use, and ripple effects are likely. Additionally, in the programmer's attempt to scope the activity and construct a plan of change, a large amount of information may need to be examined, organized, and later recalled, usually without much help from the programming environment. These added responsibilities tax the programmer and can (for example) slow the effort, leading to the introduction of bugs or abandonment of the effort.

2 Managing Aspects with Information Transparency

To test our hypothesis that secondary design decisions have a signature that can be exploited by tools and to overcome the shortcomings of current toolsets, we undertook the design of a set of tools collectively called the *Aspect Browser* (*AB*). Our goal was to create a programming environment that (a) provides and integrates tools to assist a programmer's reasoning and recall with regard to cross-cutting aspects, and (b) encourages programmers to think in terms of aspects by creating the perception that aspects are real entities, much like classes and other language constructs.

As suggested by the information transparency principle, AB is heavily dependent on pattern matching technology—initially textual regular expressions to permit quick develop of an experiment-worthy prototype that could handle whatever programming languages our experimental subjects might be using.

As a starting point, we assumed that the operations one could perform on modules or classes should be performable on aspects: they can be named, annotated, visited, edited, visualized, etc. Because the aspects we wish to manipulate are dispersed in the code, some additional features to aid warranted to compensate: highlighting each aspect with a unique color wherever it appears in the code, and collecting it into a single module-like view. Finally, because aspects emerge as requirements evolve, it is also helpful if they can be discovered for the programmer. These features are realized through two integrated tools (and a few helpers), described below.

Aspect Emacs. Aspect Emacs (AE), an Emacs-Lisp extension to GNU Emacs, provides the core browsing capabilities of AB. An aspect in AE is defined as a pair of a regular expression and a color. When an aspect is activated, AE highlights the matching text in any displayed buffers with the aspect's corresponding color (See Figure 1). AE manages the aspects through a browser, shown in Figure 3. The browser shows the the pattern, color, match count, and annotation of each aspect. The user can perform a number of operations on aspects: create, delete, hide, show, show in a compressed view, change color, and edit annotation.

AE uses two simple external tools for aspect discovery, `redundancyfinder` and `aspectfinder`. The `redundancyfinder` tool searches the entire project to find significant redundancies in the code, (currently) reporting any line that appears more than once. This approach is effective at identifying code written with copy-paste programming, a common programming tactic that accelerates development without decreasing reliability [7]. The `aspectfinder` tool extracts fragments of identifier names (tags) from source code according to a programmer-specified naming convention. For example the name `delete_source_file` contains three tags, `delete`, `source`, and `file`, separated by underscores. A program written in this style can have its tags extracted by specifying underscore as the separator to `aspectfinder`. Both tools report their results as a list of candidates and associated

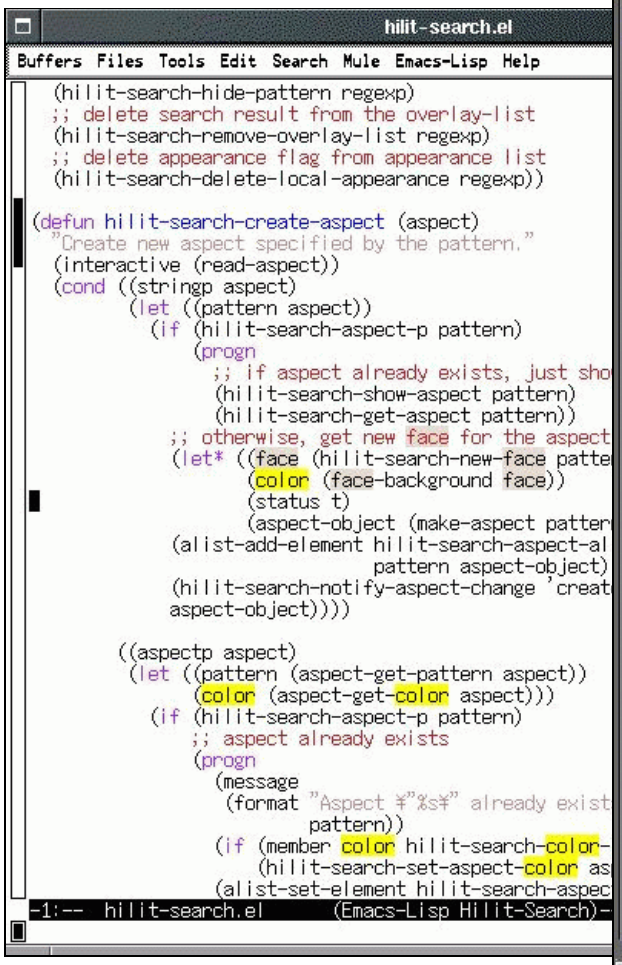

Figure 1: A screen shot of Emacs with highlights.

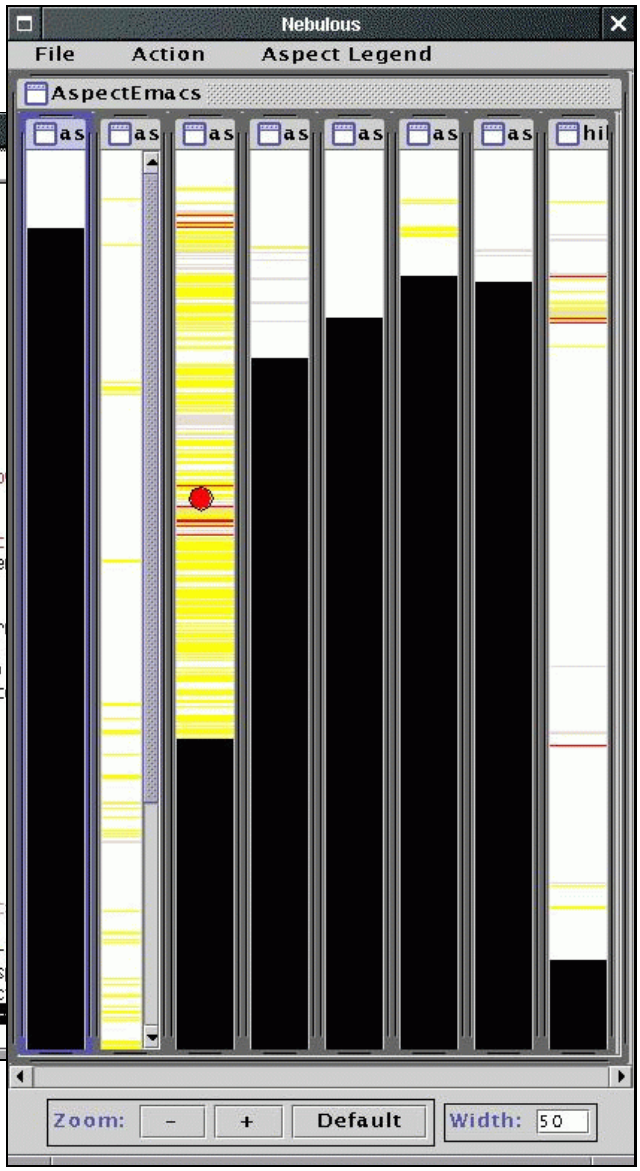

Figure 2: A screen shot of Nebulous.

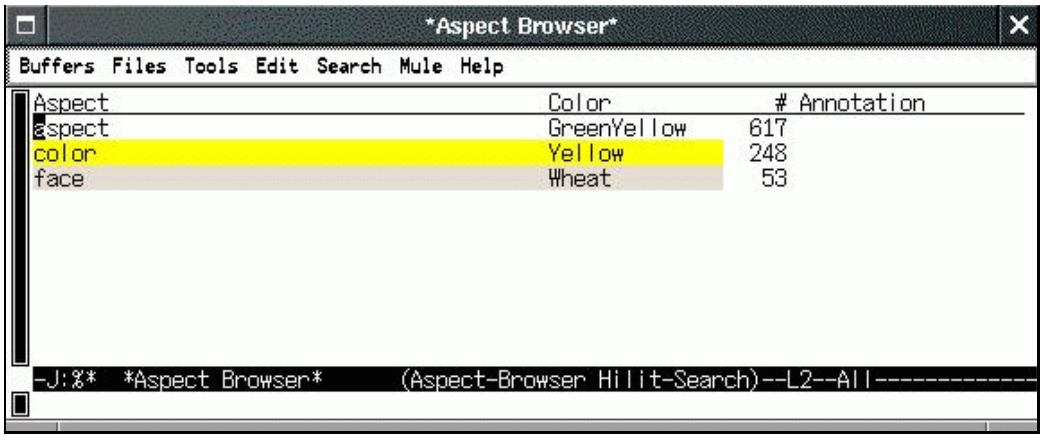

Figure 3: A screen shot of Aspect Browser.

occurrence counts; a candidate is included in the aspect browser with a mouse click.

Nebulous. Nebulous provides a global view of how various aspect entries cross-cut the source code using a view based on the Seesoft concept [1]. Each file is represented as a vertical strip, where the first row of pixels in the strip represents the first line of code, and so forth (Figure 2). Consequently, each line of code is represented in the view so that enabled aspects are highlighted in the view much as they are in AE, but since many files are shown, the true nature of the cross-cutting can be readily perceived. If more than one aspect resides in the same line, however, Nebulous shows the line in red to indicate that there is an "aspect collision". This is necessary in part because of the low resolution of the view, but it also proves to be useful in software evolution tasks, as described later.

To preserve consistency, AE sends a message to Nebulous for each aspect operation. Nebulous also provides hyperlinking to the source code: a double-click on a strip in Nebulous window causes Emacs to open the file and display the portion of the file where the user has clicked. Zooming in this way leaves a red round cursor in the Nebulous view, helping the programmer stay oriented when returning to Nebulous.

3 Experiment

Methods. As an initial test of the information transparency hypothesis and the Aspect Browser prototype, we conducted a preliminary experiment with two of the authors as the subjects—a programmer adding some functionality to an existing system with the help of a partner. The activity was recorded by videotaping the screen and recording audio of the two subjects. The target program was `wine2html`, an HTML generator that takes plain-text wine reviews and generates reviews in HTML along with accompanying views (e.g., lists by winery and varietal) that link into the reviews and vice versa. The program was written both in a highly modular fashion and in an information-transparent style. Transparency was achieved by consistently naming identifiers after their multifaceted roles. The program is written in the Icon programming language [2].

Before the modification, `wine2html` output all the reviews in a single file. If the number of reviews is large, the time to load the review page can be unacceptably slow. The selected task, then, was to modify `wine2html` so that it writes the reviews into multiple pages by limiting the number of reviews allowed in one page. This task was complicated by the cross-linking between the main reviews page and the other views: the change would involve not only changes that result in opening and closing many files, but also changes to the URL's that are generated to track the currently open file. The subjects were given two hours to complete the task.

Observations. Despite the highly modular and information-transparent style of the program, this change had not been anticipated by the subject. The programmer began an exploratory phase by noting which modules, features, or aspects of the system might be impacted by the change, observing for example that reviews, files, and URL's would likely be affected. He then created aspect entries in the aspect browser for these, with patterns `review`, `file`, `url` and so forth, as these were the names of roles that he had encoded into identifier names. For example, there is a routine name `review_open_file`, which opens the file to which the HTML reviews are written. As these aspects were created, the programmer noted both where the highlighting appeared in the nebulous view and how the highlighting was clustered or showed up in red due to two aspect hits on the same line. He hypothesized that these clusters indicated where these roles interacted (via the execution flow of the program), and hence where changes to the output file could impact review production and URL generation.

Most of the clustered highlighting turned out to be in two modules (i.e., files), the main routine and the review generator, with scattered highlighting elsewhere. The programmer focussed his energy on these two, delaying consideration of the other aspect hits until necessary. He used the double-click hyperlink feature of Nebulous to bounce back and forth between the two files, which permitted him to work in terms of geographical features (coloring) and the location of those features, rather than in terms of files and positioning in those files. When the programmer started making changes, he became aware of the (unhighlighted) tags embedded in the identifiers he was editing, realizing that some of the roles suggested in the names might be affected by the change as well. This caused him to perform further exploratory activities to ensure he had properly scoped his activity.

Analysis. The most important result from the experiment is that AB, centered around Nebulous, operated as a *map* of the program through which the programmer planned and then carried out the activity, much like a long car trip. The programmer used location in the Nebulous view as a means of orientation in the system, and the highlighting of aspects as a means of identifying salient "features" of the landscape. The implication that dispersed aspects might be

productively represented as map features rather than locations suggests that treating aspects differently than modules (which are metaphorically locations on a map, not features) may be a fruitful research direction.

The other positive observation is that the programmer came to think of the program not only in terms of modules, functions, etc., but also in terms of cross-cutting aspects. This thinking was not only observed during visualization and navigation, but also during editing, as when new concerns arose by observing names of roles in identifier names.

The programmer did not use the tools entirely as anticipated, as there was not just one aspect pattern that the programmer used to identify the relevant code. The programmer identified several aspects and observed their interactions in Nebulous to triangulate on just the "review file handling" aspect. There may be two reasons for this. One, for this particular change, the programmer was only interested in the file role as it pertained to writing out reviews, not other I/O activities. This required some kind of aspect intersection, which the programmer naturally achieved with the map metaphor by creating two or more aspects and observing the proximity of the aspect highlighting in Nebulous. Strict adherence to role-based naming conventions provides the possibility of direct conjunctive pattern matching, but still does not enable multi-line matching or conjunctive patterns, neither of which is supported well by current tool sets. Two, because dispersed aspects are not wrapped in abstractions, changes to them can impact other code. Thus, due to the role of files in forming HTML URL's, the URL role got pulled into the change as well. Thus, we observe that the aspect management mechanisms as defined in AB can be used to both narrow the scope of an activity *and* selectively expand it as new issues arise. Due to the lack of strict modularity arising from evolving requirements, both kinds of support might be crucial to managing aspects in evolving systems.

On the negative side, the programmer observed that when too many aspects were highlighted, he became overloaded by the number of colors in the view, and he could no longer easily focus on individual aspects in Nebulous. He compensated for this by turning off aspects that were not relevant to the immediate activity.

Other scalability issues were not touched upon because the program is rather small and the task limited in scope; all files were able to be displayed in the Nebulous window without left-right scrolling, and only one file required modest up-down scrolling. Scale and generality issues are being explored in a second study of a programmer removing a feature from a 500,000 Fortran program, but the data has not been fully analyzed.

4 Current Work and Conclusions

The main question that arose out of the study was how to scale up AB without losing the power of the map metaphor. Consequently, we looked to techniques used in map design and map usage to manage scale. For example, we have designed a mechanism for hiding files without aspect highlighting that is analogous to the way a map of the US might get folded in order to put California next to New Jersey: the unhighlighted files are hidden, but there is a thin line left that indicates that something is folded underneath. We are also adding the concept of *itinerary pins* to AB to permit the programmer to mark places that should be revisited later, and providing traversal features that permit conveniently walking through an itinerary or the matches of an aspect.

From the observations, we came to the conclusion that programmers probably cannot view too many aspects at a time because of the overwhelming number of colors. Also, on a bigger project, the number of aspects will increase, and a way to organize aspects will be necessary. This led to a concept of *workspaces*. A workspace is a set of aspects that can contain other workspaces as well. Workspaces could work as disjunctive aspects: turning on a workspace causes all aspects in it to be turned on. By using workspaces, programmers can organize aspects according to their tasks, turning on the workspace that is related to the next task and turning off those that are no longer relevant.

Our initial results are promising, but many questions remain: Is the map metaphor appropriate for managing cross-cutting aspects? Does the approach scale up? What kind of pattern matching technology will provide the kind of support programmers' really need? What additional kinds of aspect management do programmers need? And finally, can or should aspects be forced into module-like representations, or can they benefit from being treated differently?

AB is available from `http://www-cse.ucsd.edu/users/wgg/Software/`.

Acknowledgments. An early version of Nebulous was implemented by Eric Lundberg. Gregor Kiczales suggested the idea of highlighting in program text to show cross-cutting.

References

[1] S. G. Eick, J. L. Steffen, and Jr. E. E. Sumner. Seesoft—a tool for visualizing line-oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, November 1992.

[2] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, second edition, 1990.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. In *11th European Conference on Object-Oriented Programming*, pages 220–242. Springer-Verlag, June 1997.

[4] C. V. Lopes and G. Kiczales. Recent developments in aspectj. In *ECOOP'98 Workshop Reader*. Springer-Verlag LNCS 1543, 1998.

[5] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[6] D. L. Parnas, G. Handzel, and H. Wurges. Design and specification of the minimal subset of an operating system family. *IEEE Transactions on Software Engineering*, 2(4):301–307, December 1976.

[7] M. B. Rosson and J. M. Carroll. Active programming strategies in reuse. In *ECOOP '93, 7th European Conference on Object-Oriented Programming*, pages 4–20, 1993.

[8] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 107–119, May 1999.