# Exploiting the Map Metaphor in a Tool for Software Evolution*

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

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

[†]Department of Advanced Interdisciplinary Studies
University of Tokyo
4-6-1 Komaba, Meguro-ku, Tokyo 153-8904, Japan
yoshi@ai.rcast.u-tokyo.ac.jp

## Abstract

*Software maintenance and evolution are the dominant activities in the software lifecycle. Modularization can separate design decisions and allow them to be independently evolved, but modularization often breaks down and complicated global changes are required. Tool support can reduce the costs of these unfortunate changes, but current tools are limited in their ability to manage information for large-scale software evolution. In this paper we argue that the map metaphor can serve as an organizing principle for the design of effective tools for performing global software changes. We describe the design of Aspect Browser, developed around the map metaphor, and discuss a case study of removing a feature from a 500,000 line program written in Fortran and C.*

## 1 Introduction

The cost of software maintenance and evolution are often disproportionately large relative to the amount of code changed or the resulting change in software behavior. One cause is inadequate modularity: a change that might have otherwise been local is dispersed across several components. Because the programmer making the change does not have all the code of interest in a single place, extra effort must be invested in identifying all the places requiring change (*completeness*), and changing those places such that each change is consistent with the others (*consistency*) [3]. Otherwise, the change will be made incorrectly, requiring even higher correction costs. As a simplistic example, renaming a global variable requires locating the variable's definition and its uses, *and* changing them all to the same name. A misspelling or the choice of a name that is the same as a local variable can inject a hard-to-find software fault.

When a change is limited to a module's file, the code of interest is concisely enumerable by scanning all the lines visually with an editor or by using a simple search. Conflicts with other modules are precluded or minimized by encapsulation. This inherent *visibility* is what gives modularity its cognitive leverage: consistency and completeness can be achieved through a process of reliable visual *recognition*, with a minimal dependence on potentially faulty *recall* from memory [13, pp. 118–121]. This reduces the likelihood of injecting faults or the need to double-check for completeness and consistency of the change.

Unfortunately, achieving long-term, stable modularity of a useful software system is impossible [2]. For one, the software designers must anticipate all future changes—added features, performance improvements, and interoperability requirements—and localize each anticipated change to its own module. The whims of users and competitive markets inevitably create a need for software changes that could not have been anticipated. Consequently, designers make a best-effort estimate of what changes are most likely in the foreseeable future and focus on isolating them to modules. Changes outside this scope will be dispersed throughout the source code.

A natural question, then, is how can tool support assist in making these global changes? The tool of choice today is the text matching tool `grep` [1] or one of its many cousins, due to its ease of use, speed, and integration with the editing environment. `Grep` takes a regular expression and a list of files and lists the lines of those files that match the pattern. However, these tools have several limitations: relatively weak pattern matching capabilities; insufficient information about both a match's relationship to the module in which it appears, to the other matches, and to the results of previous searches; and incomplete management of information relating to the change, such as progress in the overall task. These can complicate achieving completeness and consistency by increasing the programmer's dependence on recall to make the changes correctly. A tool like Seesoft, which enables visualizing lines of code that match a criterion (like a textual regular expression) in a global view of files [4], can complement a tool like `grep`, as it enables seeing how such matching lines relate both to each other and to the system at large. This was our starting point for *Aspect Browser (AB)*, a tool for assisting evolutionary changes by making the code related to a global change (a kind of *aspect* [6])[1] feel like a unified entity, even like a module [5]. However, a pilot study

*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 AspectJ group at Xerox PARC.

[1] An *aspect* is a program property or behavior whose implementation naturally *crosscuts* the behavioral decomposition (e.g., procedures and objects) of a system. Consequently, it may be difficult or impossible to effectively separate an aspect from other program entities. Prototypical aspects are security, caching, memory management, and complex protocols, although the aspects arising out of software evolution are often due to the addition of new functionality after the basic architecture of the software is established.

revealed that AB's simple combination of the two technologies would inadequately relieve the programmer of recall demands when evolving large programs.

This experience led us to the insight that, instead of AB providing support so that aspect code can be treated as a module, AB could adopt a metaphor based on maps [7, 14], which would support the manipulation of both modules and aspects as first-class entities in their own right. Maps are a highly evolved human artifact for planning and way-finding based on spatio-visual abstraction. They abstract away unneeded detail to compress presentation to human-sized proportions and leverage the human visual system to recognize entities and organize complex information.

Symbols and a host of other standard map features elevate physically dispersed entities to first-class status. For example, the location of an entity on a map (placement on a 2-dimensional plane) is readily discernible by humans. Likewise, a symbol representing a class of entities (e.g., shape, texture, or color) that is visually distinguishable from others permits quick recognition and classification. To apply the map metaphor to programs, spatial and symbolic qualities must be ascribed to otherwise non-spatial and non-symbolic entities. A seemingly natural choice is for delineation of regions to denote modules and for symbols to denote members in aspects: a module is a scoped entity and its code is logically contained in it; aspects, lacking such containment, are naturally assigned to symbols that are recognized by appearance rather than location. Using location relationships—proximity and containment—the relationships amongst modules, amongst aspects (e.g., by proximity of aspect symbols), and between modules and aspects can be quickly recognized. This is a property exploited by Seesoft [4]. AB completes the map metaphor, including indexing (*ala* street indexes), a cursor ("you are here"), customization (e.g., zooming), annotation, and folding (i.e., as a street map is folded to make it smaller). Some of these features fall into the realm of *animated maps* (support for automated customization), and others are implementations of informal ways that maps are used to make them more useful. These are essential in our domain both because of the extraordinary size of the programs being mapped and the wide variety of uses (tasks) to which the map is applied: there is no one-map-fits-all solution. Animation also permits the map to remain up-to-date, as of course the program is being changed as a part of the overall task.

The next two sections discuss the relevance of the map metaphor to software evolution and AB. To gain insight on the relevance of the map metaphor to evolution, we then describe a case study of AB applied to removing a feature from the 500,000 line Fortran/C finite element solver system. Before closing, we discuss related work.

## 2 The Map Metaphor

In addition to putting modules and crosscutting aspects on an equal footing by overlaying aspect symbols on module regions, adopting the map metaphor addresses the challenge of scale. Evolving and maintaining large systems involves crosscutting aspects of enormous size and complexity, and could result in global views that are beyond the capacity of normal computer monitors or the cognitive capabilities of a programmer. Likewise, a change to an aspect can be spread over a considerable span of time, stressing a programmer's ability to recall what had been accomplished so far and what remains to be done.

Maps have evolved over time to address scale issues [14]. Modern maps provide magnified insets (zooming) to show needed detail in small, critical regions, thus allowing the main map to be rendered at a smaller scale; they provide indexes of special entities (e.g., roads, parks, schools) to permit locating them by alphabetic search rather than scanning the entire map; they are creased to permit folding to fit in a small space, while at the same time allowing two far-away locations to be placed next to each other; they can be marked, annotated, and stuck with pins to record long, complex routes and mark one's current location on that route; and the color scheme can be "dimmed" on parts of the map to indicate they are not of (current) interest (e.g., the periphery of a map of the U.S. will show Mexico and Canada in greyish tones). Animated maps allow the map user to dynamically choose what is zoomed and how much, what is dimmed, and what features are displayed on the map, permitting a higher level of customization than informal actions like folding and marking. Animation also ensures that the current state of the entity is being mapped, which is an essential property for software evolution. Likewise, changes to the map over time can reflect progress in completing a task, for example conveying which places have already been visited and which have been changed. With animation, a map really comes to represent a map—or plan—of the software evolution *task* being carried out, not just the (static) program.

We used the map metaphor, then, to instruct us on how to manage scale in AB. The metaphor also revealed that seemingly independent features in AB could be related through the metaphor. For example, when we cast the list of aspects as an *index* of aspects, we recognized that the list of redundancies that the tool infers for the user (Section 3) is an aspect index as well, and should be managed through a common mechanism. Such unification, combined with the programmer's familiarity with the features of maps and how they work, eases use of the tool.

## 3 Aspect Browser

An aspect in AB is defined as a pair consisting of a pattern (a grep-like regular expression) and a color. When an aspect is enabled, the display of any program text matched by the pattern is highlighted with the aspect's corresponding color. By using enclosed regions to represent modules (e.g., files) and swatches of color within regions to denote the location of a portion of an aspect, both modules and aspects can be given first-class representation in the environment. This functionality is achieved with two integrated tools, Nebulous and Aspect Emacs.

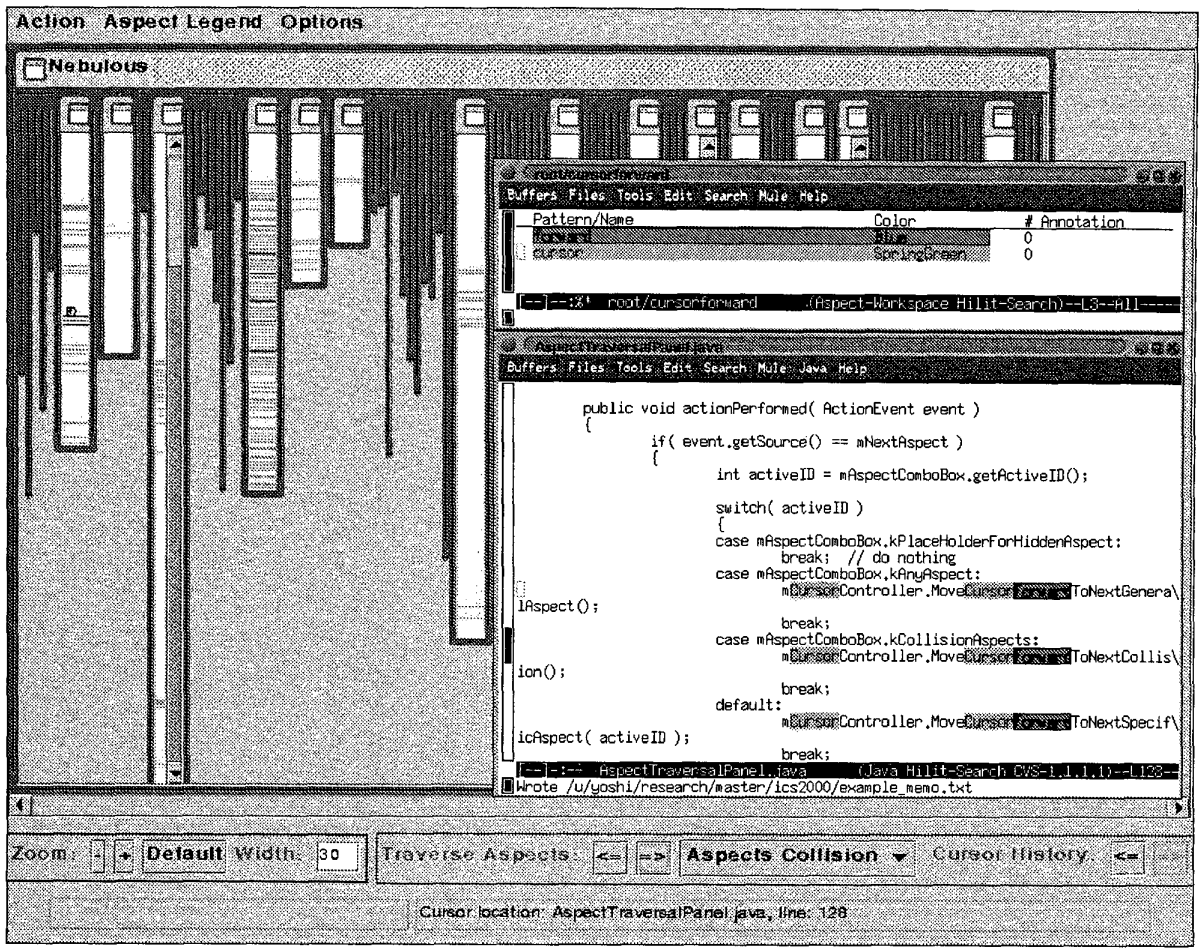


**Figure 1. A screen shot of Aspect Browser being used on the Nebulous source. The top of the foreground panel is the aspect index, containing a pattern `forward` in blue and `cursor` in green, chosen to help find where code has to be added for a new traversal feature, based on conflict highlighting. Below the index is an Emacs buffer containing these aspects. In the background is Nebulous with aspect-less files folded under. The Nebulous cursor is about 2/3 down the first unfolded file from the left. The zooming and traversal features appear near the bottom, with the status bar below them.**

**Nebulous.** Nebulous provides the main map of the program (or task), using a view based on the Seesoft concept [4]. Each file is represented as a region—a vertical strip—in which the first row of pixels in the strip represents the first line of code, and so forth (Figure 1, background). Each file's background is set to the editor's background color to provide a cue that the file strips are low-resolution maps of editable source code. Files are placed by default in alphabetical order from left to right, although other orders (e.g., by age) may be chosen. Subdirectories of the project are also shown as (large) regions, each containing its out set of file strips. The hierarchical and alphabetical listing provides a spatially accurate map of how a project is represented in the file system, minimizing reorientation when the programmer changes focus from the tool to the command prompt and *vice versa*.[2] The files in a project are specified by a list of regular expressions and an indication of whether inclusion should be hierarchical; the specification is stored in a project file for use across sessions.

In Nebulous, each displayed aspect is highlighted in its color at matching locations on a line-by-line basis, permitting the nature of the crosscutting to be readily perceived. If more than one aspect resides in the same line (e.g., two aspect patterns match different parts of the same line), Nebulous highlights the line in red to indicate that there is an "aspect collision". This is necessitated by the low resolution of the view, but it also proves useful during software evolution tasks by providing a cue as to where aspects are likely

[2] Alternate hierarchical presentations of the project can be specified.

interacting. Proximity of aspect highlighting provides a similar cue, when there is not a collision. A separate map key lists the aspects in the view (including the "conflict aspect"), listing each aspect's pattern, color, and number of matches found throughout the system.

All state-changing operations such as enabling an aspect pattern or saving a file are reflected in the Nebulous map. Double-clicking on a strip in the Nebulous window causes Emacs to open the file and display the portion of the file where the user clicked, in essence generating an editable map "inset". This action leaves a red cursor in the Nebulous view, helping the programmer to stay oriented when returning to Nebulous.

In addition to the "you are here" cursor, Nebulous provides several mechanisms for managing large maps. First, it provides the standard, un-map-like *scrolling* operations: horizontal scrollbars for showing hidden files and vertical scrollbars for showing hidden portions of a file.[3] A "Where was I?" operation is also available for quick-scrolling to the current cursor, should it be moved off the screen during perusal of the view.[4]

Since invisibility of information is generally undesirable, *zooming* permits changing the resolution of the map to better fit the display or show more detail. Zooming out one level, for example, causes two program source lines in a file to be mapped to one row of pixels in a file region; merging two highlighted lines of different aspects results in red conflict highlighting. File width can be separately adjusted if desired. Since a narrow width can clip a file's name, a *status window* shows the name of the file currently under the mouse.

*Dimming* grays out files that do not contain any highlighted aspects, helping the programmer to focus attention on the brighter files. *Folding* narrows the interior of each file that does not contain highlighted aspects, leaving its border as a "crease" that indicates a file has been folded under. Folding provides compaction of the view, but it also alters spatial relationships such as positioning. This can bring portions of an aspect closer together, but can also disorient the programmer by putting files in unfamiliar locations in the map (i.e., farther to the left). The amount of folding that occurs can also provide a quick assessment of how extensively an aspect crosscuts the system.

To both manage scale and to ease non-modular operations such as aspect perusal (i.e., make aspects first-class entities in the tool), Nebulous provides two kinds of traversal operations. *Aspect traversal* enables walking forward and backward through the lines comprising an aspect. A single chosen aspect can be traversed, or all enabled aspects may be traversed together. Forward traversal proceeds from top-to-bottom, left-to-right in the map, with the cursor indicating the location on the map and the editor window showing the code at the location. If the traversal starts in the upper left-hand corner, then the cursor divides the map into a "visited" region to the left and a "to be visited" region to the right, thus helping the programmer keep track of progress in the traversal or task with a single concise cue. *History traversal* allows backing up and revisiting a series of previously visited locations, whether visited by aspect traversals or double-clicking with the mouse. It is useful for reviewing or revising prior changes.

**Aspect Emacs.** Aspect Emacs (AE), an Emacs-Lisp extension to GNU Emacs, provides the map indexing, map inset, and editing capabilities of AB. When an aspect is enabled, AE highlights the matching text in any displayed buffers with the aspect's corresponding color (Figure 1, bottom foreground). Because of the high resolution of the editing view, there is no need for conflict highlighting—each aspect match is independently highlighted in its color.

AE manages aspects through an aspect index browser (Figure 1, top foreground). The browser shows the pattern, color, match count, and programmer annotation of each aspect. The user can perform a number of operations on aspects: create, delete, hide (disable), show (enable), list (show all lines of an aspect), change color, and edit annotation. Because a large number of aspects may be created and the tool may be used for a variety of tasks, hierarchical indexes are provided to organize aspects. An index is essentially a compound disjunctive aspect, so all aspect operations can be applied to indexes, providing concise manipulation of a group of aspects. There is a simple tool for walking a user through initializing a project (specifying the project files and naming conventions), and the full browser state can be saved across tool sessions.

As requirements evolve, modules may be explicitly added by the programmer, but aspects can emerge, unnoticed. Consequently, it is helpful if aspects can be discovered for the programmer. AE uses two simple proof-of-concept tools for aspect discovery—automatic generation of aspect indexes—called `redundancy-finder` and `tag-finder`. The `redundancy-finder` tool searches the project to find redundancies in the code, reporting any line that appears more than once (ignoring leading and trailing white space). This approach is effective in identifying code introduced by copy-paste programming, a common programming tactic [15]. The `tag-finder` 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. Both tools report their results as an index of aspect patterns and associated occurrence counts, although their initial state is unhighlighted (disabled), unlike manually created aspects.

[3] One could argue that scrolling is akin to flipping through the pages of an atlas. An atlas is in essence an ordered list of maps with a shared index, packaged in a small form factor (small for maps, anyway). They are most effective when the coupling between maps is minimal, which is not necessarily the case with the evolution of crosscutting aspects.

[4] In which case the cursor is functioning like a bookmark in an atlas.

## 4 Case Study

There were three questions we wished to explore with respect to AB's use in making large-scale, crosscutting changes. First, we wanted to see whether a programmer would use AB as a map; that is, adopt behaviors consistent with using maps, including language, gestures, and actions. Such behaviors would suggest a naturalness of the metaphor to software evolution and indicate that we had been successful in employing it. Second, we wanted to see whether the map metaphor actually assisted the programmer. Third, we wished to identify potential improvements to AB.

### 4.1 Experimental Method

We chose to perform a case study—that is, introduce our tools into a real work environment, with minimal manipulation by the experimenters [16].[5] Such a choice would achieve high realism in all aspects of the task. We chose a project in the Bioengineering department at UCSD that had been developing and using an advanced finite element analysis tool for a number of years. This large project had been encountering problems in on-going development due to legacy software issues, making it suitable for study.

The programmer who volunteered for the study had used an earlier version of the tool, and understood the concept of crosscutting aspects. He usually uses the `vi` editor rather than Emacs, but since he felt our tools would help him with a specific task, he volunteered for this case study.

We used an observational method called constructive interaction, in which a pair of programmers work in conjunction to make the change [8]. Paired problem solving results in natural, productive conversation, providing insight into the programmers' thought processes without the unnaturalness of consciously talking aloud for the experimenters. The programmer (hereafter called the subject) had used paired programming extensively as an undergraduate, so this was a natural work mode for him.

The subject, however, used paired programming only occasionally in this work context. Since he was not very familiar with AB or Emacs, using one of the tool authors to help with unfamiliar features seemed natural, as paired programming often occurs in learning situations. Since the program was written in Fortran, we chose a tool author who also had Fortran expertise. Together, these would reduce the "second wheel" status of the second subject, resulting in more constructive interaction. (Observation bore this out; the second, participant–observer subject successfully questioned the primary's programming several times and instructed him on some obscure aspects of Fortran.) Below, when we refer to "the subject", we are referring to the primary subject.

For data collection, we videotaped the monitor of the SGI Octane Irix (Unix) workstation where the subjects worked. Each subject had a small microphone clipped to him in order to record the dialog throughout the experiment. These two sources allowed us to study the discussions, mouse movements, and process through which the work got done.

The second subject provided a brief tutorial of new features added to Aspect Browser since the subject had last used the tool. After this, the subject was allowed to use any tools as he saw fit, and the second subject was instructed to not influence such choices (and stuck to this instruction). Interaction between the subjects and experimenters took place only if there were technical problems with the tools or the like.[6]

**The Task.** Prior to the study, the subject had selected a maintenance task that a tool like Aspect Browser should be able to help him perform. We did not set any time limit for the task, since it was part of his job (it took 18 hours). Likewise, the subject was free to take breaks as he liked and decide when the experiment was over.

The program being modified was a finite element analysis program consisting of about 500,000 lines of Fortran code and several thousand lines of C, totalling about 40MB of data. The program is divided into about 2500 files over 20 directories.

A feature called *regions* had been abandoned many years ago, but not removed. Regions had been implemented by adding a dimension to existing arrays and an integer value denoting the number of regions being used in the current problem. This integer, usually communicated to functions as a parameter called `NR`, was hardwired to be 1 in every place it existed in the code, and in other places it was never put in at all. The task was to remove all code pertaining to regions without breaking the program. The feature's code spanned across all major directories of the program source.

### 4.2 State of Aspect Browser

A few features described in Section 3 were not complete and hence unavailable during the study:

- tool for project file creation (an editor was used instead),
- hierarchical aspect indexes,
- Nebulous's status panel (tooltips were shown instead),
- file strip ordering options in Nebulous (defaulted left to right in the order read in from the file system),
- complete regular expression support.

Although useful, none were deemed essential for the task, and issues that arose that pointed to their usefulness would be valuable outcomes. One sub-feature of regular expression matching *was* supported, exact-word matching, which permitted matching on short variable names without extraneous matches to variables that contained the pattern as a substring.

[5] A common concern with a single-case study is that it represents just one data point. Case studies, however, are analogous to experiments, whose results are tested via replication, not sampling [16, pp. 45–50].

[6] The presence of one or more observers, either external or participant–observer, is a necessary component of case study research for the purpose of data collection; the potential for bias is minimized by issuing careful instructions for interaction and following them [16, pp. 86–89].

### 4.3 Observations

**High-level observations.** The subject knew a key variable for the regions feature was NR. He also had some documentation (on a website) that listed functions and arrays, some of which took NR as a parameter. This was a starting point for the task. He wrote the function, array, and variable names on paper, and said they were his initial aspects.

The subject ran AB on a per-directory basis, choosing the directories in the order reported by the Unix `ls` directory listing command. He used `grep` on the first couple of directories, and concluded that they contained no code of interest. In directory `fe02`, his search produced many matches, so he started AE. He created an exact-word pattern for NR to avoid matching the letters "NR" contained in any word.[7] He also added the other names he had written down, and changed their colors to something less objectionable.

He then created a file for Nebulous that specified what files are part of the project, and started Nebulous. `fe02` contained 90 files. He zoomed in once to make the aspect lines thicker. (In this project the convention is to have one function per file, so this zooming only obscured lines in a file for the largest functions.) He changed the file strip width from 50 to 30 pixels to get more files on the screen.

Starting with the leftmost file in the Nebulous view, he double-clicked on the first highlighted line, which brought up an Emacs view with highlighting of the selected aspect line (and those nearby) in the view. He began removing occurrences of NR and related entities while scrolling down using editor commands.

Several subtasks emerged. One, he removed references of NR appearing in function calls. Two, if he had not encountered this function before, he wrote it down to be added as an aspect, along with an annotation about the parameter position at which NR appeared. All new functions would be added when he finished modifying the file. Three, such function definitions had to have the parameter declaration deleted as well. Four, he removed uses of NR as an array index. Five, he removed the region dimension from the declarations of said arrays. Six, he removed loops iterating over the regions (with NR representing the upper bound of the loop). Finally, because certain variable declarations (formerly) related to regions were imported (i.e., `include`'d) but unused, he checked for this and removed the import if not used. The include file's name was an aspect, serving as a reminder to do this subtask.

When the task was complete, he had created 128 aspects. He then ran a small set of standard multi-part test cases to expose any introduced bugs, working from easiest to hardest, running them interactively so he could isolate any misbehaviors. Only one malfunctioned, but the subject could not determine if the problem was associated with the changes he made or not, in part because he had not run the test cases immediately prior to the change as a baseline. Later, it was determined that it was a problem unrelated to the change. The most difficult test case was the most demanding problem ever run on the system—a simulation of an electrical wave propagating through a heart—and it ran to completion without any glitches. The subject was surprised at the performance improvement, claiming it was 25% faster.

**Routine Behaviors.** For each directory, the subject entered the directory, copied the project file specification from the previous directory, and started Emacs with aspect browser mode. He then loaded the list of aspects he had built up and saved from previous directories. Launching Nebulous, he folded under files without aspect matches, changed the file strip width to be thinner than the default, and zoomed in once. Starting at the leftmost file strip, he always double-clicked on the first match, which brought up Emacs with the match in its view. When done with a file, he would save it and then raise the aspect index window to add any new aspects, immediately saving the list. He would then slide the mouse over the Nebulous view where the cursor had been left behind by the double-click navigation to Emacs, slide the mouse to the next file strip to the right, and then double-click on its first match to bring up the code at the selected match in Emacs.

For functions that took NR as an argument, the subject wanted to define an aspect using not only the name as a pattern (e.g., `ZPZE`), but he also wanted to use regular expressions to capture the full argument list of the call (`ZPZE(.*,.*,.*,.*,.*,.*)`). In this way, only unedited calls would be matched (in the example, those containing 7 rather than 6 parameters). Thus when the file was saved after editing, Nebulous would be updated to show that the aspect no longer appeared in the file, signalling that the change was complete (*"I want all this to be white,"* he said, true to the map metaphor, although in fact the map would have been refolded to remove irrelevant "white" file strips).

Lacking full regular expression support, the subject depended upon a less reliable combination of cues and process. The subject would double-click from Nebulous to the first match in a file, and inspect the matches top-down in the edit buffer, normally using line-by-line scrolling (e.g., hold down control-N for continuous scrolling), quickly looking at each match, making a change if necessary, continuing to the bottom of the buffer. He would then (normally) rescroll to the top of the buffer to make sure everything was OK. This second backwards pass also served to return the subject to the top of the buffer where sometimes the possibly unused import resided, permitting it to be deleted if no uses of the import were encountered on the way up. This process ensured that every match—some spurious—was inspected up to two times. Relevant lines were usually highlighted in two

[7] Although NR is not global, apparently the naming conventions on the project virtually ensured that any function parameter representing the number of region would be named NR. This is not just fortunate, but a wise tactic, as Fortran's simple data representations and type system cause programmers to encode many entities as integers, rendering program type information useless in distinguishing entities. Even with modern languages, of course, programmers often use integers directly for all kinds of quantities.

places in two colors: a use of the parameter NR and a function known to take a region number as a parameter. The bright green highlighting of NR was a sure hit, but sometimes just the function was highlighted because the constant literal 1 was passed instead of NR to the function. At a glance, such calls have the appearance of an edited call or other kinds of spurious matches. The subject in fact missed one of these, but it was caught by his partner.

Ultimately, the subject used three different methods to visit each match in a file. He volunteered that he tended to scroll over the matches in the edit buffer rather than use the traversal features because he did not have to take his hands off the keyboard. (He suggested that we add a hotkey shortcut for aspect navigation.) He said he used double-clicking on each match in a file strip (always top-to-bottom) when the matches were so far apart that it was faster to move his hand off the keyboard and use the mouse. Finally, he noted that he used the aspect traversal buttons for large files whose file strip was not fully visible (and hence not fully clickable without scrolling). This method took him to the next file automatically, which he complained about because he wanted to save a file before going on to the next. His choice of method was not entirely consistent with these conditions, however, suggesting that factors like habit played a role in "choosing" a traversal method.[8]

After completing changes to the directory, the subject would typically recompile the directory to expose syntax errors. (Recompiling and running the whole program was impossible until all declarations and uses were made consistent across all directories). The most common error uncovered was deletion of the wrong ENDDO. Sometimes an NR reference in a function body was overlooked, which was undefined because its declaration had been deleted from the function definition. Also, after modifying the first directory, several array uses referenced fewer dimensions than their declarations defined; the subject immediately went to the include directory and deleted the region dimension from the declarations. Mismatches between procedure calls and declarations were not checked by the compiler or linker.

**Other Observed Behaviors.** The subject deviated from these routine behaviors in several instances, three of which we discuss here. One, after directory `fe07`, which introduced many new aspects, the subject turned off all non-function aspects and reinspected the matching function calls in the directory to make sure no mistakes had been made. Two, he realized that a variable he had been pulling out, NRMX was not related to regions and those changes needed to be undone. Rather than start over, he checked out a clean version of the system, created an aspect for NRMX for it, and used these matches to show what code should be reinserted into the modified code. Three, in a directory with a small number of matching files, he postponed the biggest file with the most matches until last. This departure from the normal left-to-right walk over the directories later led him to question if he'd already done a particular file, despite having a simple metric to remind him which file had been skipped.

The first two departures from routine behavior are somewhat analogous to his routine behavior, using aspect symbols on the map as places to revisit in a geographically contiguous order. The last also used a symbol to adjust the visitation, but the symbol—a large region—was an affordance symbolically unrelated to the change and hence more prone to confusion. Together, these exceptional behaviors point to the power of the metaphor and importance of using symbols related to the change to ensure completeness in the task.

### 4.4 Discussion

**Appropriateness of the Map Metaphor.** The appropriateness of the map metaphor can be gauged by map-like and successful use of features, and more generally by behaviors and language use that are consistent with the use of maps.

Not surprisingly, the subject had no trouble with the basic symbols adopted by AB. There was no confusion that the enclosed regions in the view (what we've called file strips) denoted the files in the current directory, and the subject used the colored lines of pixels within a file strip to denote cross-cutting aspects. He used, without confusion, double-clicking of aspect symbols to generate an inset for detailed inspection.

More interesting is the use of the aspect symbols as an enumeration of places that required *change*—relating symbols to *tasks* as well as to *things*—and to organize these changes in a spatially continuous route (left-to-right, top-to-bottom) through the view. His repetitive zooming and folding behaviors are consistent with idiomatic map use, in that map users constantly manipulate maps to make all the information of interest accessible without becoming ungainly. Indeed, he deviated from this folding behavior for some smaller directories, where visibility was less of an issue.

Although the subject did not—and could not—memorize the colors of all 128 aspects, the symbol content of color nonetheless assisted the change. About two hours into the study he said, *"Green is* NR*"* while looking at a file in the Nebulous view, and double-clicked on the green line to examine it more closely. A little later, he gestured at a bunch of highlights in Nebulous and said, *"These are all* RANGE*."* He also made a point of coloring similar objects the same color, although inconsistently, perhaps because it required an additional command. A color is connected not only to an aspect, but to a particular task. *"Look at all that green. There's four new aspects right there, baby. That's* NR*."* Thus, the recognition of symbols triggered the recall of subtasks, reducing the chance that they would be forgotten.

Patterns of color across file strips also served as task triggers. Swirling the mouse arrow around two adjacent strips that looked similar in length and coloring, the subject said, *"I knew I could delete all that because this file matches pretty*

[8] Methods of locating information on a paper map are also redundant. For example, in looking for a park, a person could either go to the index and look-up the grid coordinates for the park, or look at the map itself, methodically searching for green areas representing parks until the desired park is found.

*well with the one I just edited. All the colors are the exact same here."* This recognition was dependent on numerous spatial and color cues. It also depended on the proximal placement of the files to ease recognition; the chance of such placement is increased by folding irrelevant files under.

Because he was using colored line symbols to denote places requiring change, the presence and lack of these symbols were used as a progress indicator (including being done) and in estimation. *"I want all this to be white"* he said early in the task. In the first interview he said, *"The way that I think of this tool, if there's color, then I still have more to do, from looking at Nebulous."* The number of files and the particular highlight colors were also important. When starting Nebulous in a new directory he said *"Oh, there's a lot of files. Somebody shoot me."* Then after folding *"Hey it all fit. These* all *take* NR," and then ironically, *"This is great,"* expressing that he had a lot of work to do.[9]

Despite these behaviors, the subject said that having more than two colors (representing match, no match) was not essential. *"I didn't really see what colors things were, except for things that showed up a lot, I would accidentally memorize what colors they were. So, I would...see a file and I'd say, that guy's got* NET *in it. That guy's got* NR *in it. 'Cause those guys showed up a lot.... It wasn't super helpful because I was gonna have to pull those things out no matter what color they were anyway."* We judge this in part to be an artifact of the particular task and how the subject chose to use symbols to guide it; if AB had made it easier to map the aspects onto a smaller number of colors (say, one for each of the seven subtasks) a different choice might have been made.

The subject frequently spoke in spatial terms. We judge that most of these would have been used regardless of the tools used (*"Where does* NELEM *get defined?"*) to the extent that file system and program organizations already manifest the region concept of maps (which is a good thing, in our view, for software evolution and for relating what is stored on disk to what is displayed in Nebulous). Other language was more clearly related to the map metaphor of AB. While working, he pointed to the upper left corner of Nebulous, saying, *"Start here,"* and another time, *"We're making it over to the right side of the screen."* He also said *"...I've got the marker dealy on where I clicked in so I know what file was the last one I did anyway."* This language indicates some notion of a left-to-right tour across the map, with the cursor keeping track of his place. In an interview the subject said, *"Nebulous gives me a good look from far away"*, and *"It gives me an idea of how far along I am, too."* The use of these phrases suggests that the map metaphor is appropriate because it was natural for the subject to speak in terms that people may use when using geographic maps.

The failures of two un-map-like features of AB also point to the value of the map metaphor. One, occasionally a portion of a file strip was not shown in Nebulous because the file it contained was especially long. This caused the subject to overlook some hidden highlights once, despite the scroll bar's cue (the scroll bar is omitted altogether if the file fits). We also observed this problem in a separate study [17]. A zoom-to-fit button might make file-strip fitting easy enough that this problem could usually be side-stepped. Two, the non-alphabetical ordering of file strips led the subject to indulge in a time-consuming search to find a particular file in the view. Although two files might have no *a priori* adjacency to dictate their proximal placement, the consistent location cues provided by maps are nonetheless a common and powerful way for humans to find things. Directory listing commands like `ls` that produce their results alphabetically by default reinforce this cue.

**Scalability.** The program used in this experiment was simply too big to view in AB all at once. Even directories of 90 files (125 files was the largest) produced a full-width view after adjusting the width and folding under unmatched files. Loading all directories would have incurred a significant performance cost, spreading out the activity in time and stressing recall, with no added visibility to offset the loss.

The directory, however, appeared to be a natural unit of work for the task. The files of a directory were (sometimes loosely) grouped into a subsystem. Thus, the definition of a function and all of its uses might appear in one directory. The coherent functionality of a subsystem made it likelier that the purpose files served were consistent throughout a directory, resulting in repetitive changes in a narrow time frame and hence becoming easier to recall and perform as work progressed through the directory. Also, the directory was a natural unit for validating changes. After finishing a directory, the subject often performed some visual checks on the directory using AB (e.g., revisiting remaining highlights), and—taking advantage of the `Makefile` that appeared in each directory—typically recompiled after finishing a directory. Because the changes had been performed relatively recently, any problems were generally easy to diagnose and repair. In this respect, the subject's chunked use of AB is not unlike using an atlas, which copes with scale by providing a collection of related maps with a shared index. The use of a common aspect index was crucial to the chunked application of AB; retyping the 128 aspects discovered in the course of the change was not an option.

These behaviors would not have been precluded if the whole program had been loaded into AB. The hierarchical display of subdirectories provides cues than can remind the programmer to perform subsystem validation, and demand-

[9] Estimation and assessments of progress are useful in planning breaks in a long-running task so that they do not cause a disproportionate loss of the context (memory) that the programmer has developed to draw upon during the change. For example, at Noon, the subject asserted that he would take a lunch break as soon as the current directory was done. In fact, he did an additional directory because the change was quite small (*"Oh look! There's only one file here. We'll do* `fe04`*, too, then."*), enabling him to reuse that context and preserve a sense of momentum. After finishing `fe04`, a quick check of the next directory, `fe05`, suggested that it was quite large, so he decided to break for lunch.

driven computation could improve performance. The point is that (a) invisibility is countered not just by the global display, but also by the global index, and (b) scalability could be better managed by providing more explicit atlas support—perhaps filing directories under "tabs", currently a popular way to pack more function into tools.

**Other Tools.** The compiler played a key role in directory validation, finding syntax errors and inconsistent declarations. As these capabilities are outside of AB's scope, the compiler was a critical, complementary tool for this task.

The subject made limited use of other tools providing matching functionality, and most of these were early in the study. In the closing interview, the subject said *"...looking in some of the directories with grep, I'd see all this output from* `grep` *for like an aspect, and that was just one of the 128 aspects, and I'd...look at that and say, this is impossible, and I'd just get discouraged, but I'd say, well, I've gotta do it. So, I brought up...Aspect Emacs and Nebulous, and when I brought up Nebulous, I see the whole thing, from...a little further back, [and think] I can do it, it's not so scary."*

Yet he said that a tool like `grep` is superior when *"All I want to know is, is it there, right? Just [type]* `grep`*, then a name, then a star. And if anything comes up, then I have the information that I wanted. And that will always be the fastest way to find out that information."* He also noted that auxiliary or preliminary aspect searches were best done with `grep`, even when AB was already up and running, *"because I didn't have to turn off the aspects,"* to clearly see the search's results. He was unsure that the features provided by hierarchical indexes would have overcome this advantage.

Another question is whether tools unavailable to the subject might have proved useful. A tool like Lackwit (implemented for C programs) could in principle infer the *de facto* type of the region number and provide a graphical display of the relationships across functions [12]. However, Lackwit, like AB, must be bootstrapped with an initial set of variables that are of the appropriate type. Likewise, this identification is complicated by the passing of the literal `1` instead of `NR` in some places, interrupting the transitivity of the inference. Of course, once such an analysis is complete, the viewing and editing of the inferred types and their uses throughout the program could benefit from a view based on the map metaphor.

### 4.5 Completeness and Consistency

The introduction of bugs—incompleteness or inconsistency of the change—were rare; such mistakes were caught by the compiler, the subject himself, or by the subject's partner. The subject's change process enhanced completeness and consistency, and was molded by AB's expression of the map metaphor. The subject almost always worked from left-to-right, top-to-bottom in both Nebulous and Emacs, using the cursor as a cue of his location and progress in the task. However, the subject had to remember to add new aspects—there were some close calls—which the subject eased later in the study by placing the browser so that its bottom was not covered by any window. Explicit completeness and consistency actions, like backtracking over a subset of matches upon completion of a directory, were enabled by AB's traversal and aspect-disabling features.

Errors were more frequent early in the study. Compiling the first modified directory produced a number of syntax errors because matching `ENDDO`'s had not been deleted properly and because uses had been edited but not their declarations (which were in a separate include file). Updating the declarations was a one-time activity, but to avoid future problems in deleting loops, the subject decided, *"I think the first thing I'm going to do now is to delete the ENDDO at the end,"* that is, to delete matching `ENDDO`'s immediately rather than continuing to make other changes in the file while scrolling down to the (supposedly) appropriate `ENDDO`. Such problems and the subsequent decision show the tension between removing the feature (an aspect) and preserving the integrity of the current procedure (a module). In short, the completeness and consistency of the aspect change were in tension with the consistency of the module. AB addresses this tension in part by using location to denote module properties and using colored symbols to denote aspect properties. A slight change in the subject's editing process reduced the module consistency problem, but at the expense of extra scrolling in the file and potentially losing his place in the change of the aspects. For the subject, this no doubt increased the importance of being able to make the view "all white" (or all files folded under) to show that aspect change was complete.

## 5 Related Work

RIGI is a reverse engineering tool for capturing the module structure or architecture of a system [9]. It extracts an initial structural view of a program based on its hierarchical structure, function calls, and variable references. Boxes represent modules of any kind and edges show relationships like reference; nesting shows membership. A programmer can then customize the view to manage detail and capture conceptual relationships; graph layouts may be customized to achieve a more informative view.

Reflexion modelling (RM) provides a mechanism for recording, displaying, exploring, and iteratively refining a task-specific box-and-arrow conceptual view of a system [11]. It provides a regular expression mechanism for specifying component membership and a lightweight, robust source model extraction tool [10] for inferring a wide array of relationships among the components.

In both, spatial relationships present and organize information, but the tools do not complete the metaphor with map features like cursors, indexes, and folding. Neither is suited to capturing fine-grained crosscutting, although either in principle might. If given the capability to dynamically reflect changes to the software they are modelling, either could be extended with the map metaphor to more directly plan

and carry out evolutionary changes. Conversely, AB does not show relationships via edge connections.

## 6 Conclusion

The crosscutting changes that arise in the evolution of large software systems are costly, because the dispersal of related code creates a tension between managing the code in a single view and ensuring the integrity of the modules in which the crosscutting code is embedded.

Tools like `grep` and Seesoft provide basic technology to counter these problems, but are an incomplete solution to the difficulties of large-scale software evolution. The technology of maps, a highly evolved mechanism for managing scale when coping with dispersed but spatially related entities, suggests a way to augment and integrate such software technologies for their application to software evolution.

We implemented Aspect Browser to demonstrate the applicability of the map metaphor to issues in software evolution, and performed a case study to determine how the map metaphor influenced and aided software evolution. Using AB, the programmer was successful in making a complex change—removing a crosscutting feature—from a 500,000 line system. The programmer's behaviors were analogous to a map user's—such as the use of symbols to mark places requiring change and visiting the places via a tour of the symbols on the map. Moreover, the processes and strategies that he developed around AB—such as left-to-right, edge-to-edge tours of the map to ensure completeness and combining highlighting and hiding to focus attention—were successful in minimizing the introduction of bugs and produced a running system with a minimum of debugging.

The problems the programmer encountered point to additional ways the map metaphor could be applied, such as the inclusion of atlas features to improve scalability. Future studies can provide additional insight into the appropriateness of the map metaphor and how it might better assist software evolution.

**Acknowledgments.** An early version of Nebulous was implemented by Eric Lundberg. Gregor Kiczales suggested the idea of highlighting in program text to show crosscutting. We are grateful to our anonymous subject as well as Andrew McCulloch and his group for welcoming us into their lab. We also thank the anonymous reviewers, Michael Copenhafer, and Wesley Leong for their detailed, constructive comments on the original submission. Aspect Browser is available on the internet from `http://www.cs.ucsd.edu/users/wgg/Software`.

## References

[1] A. V. Aho. Pattern matching in strings. In R. V. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 325–347. Academic Press, New York, 1980.

[2] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976. Reprinted in M. M. Lehman, L. A. Belady, editors, *Program Evolution: Processes of Software Change*, Ch. 8, APIC Studies in Data Processing No. 27. Academic Press, London, 1985.

[3] R. W. Bowdidge and W. G. Griswold. How software tools organize programmer behavior during the task of data encapsulation. *Empirical Software Engineering*, 2(3):221–267, April 1997.

[4] 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.

[5] Y. Kato W. G. Griswold and J. J. Yuan. Experimental study on scalability of tools utilizing information transparency. In *International Conference on Software, 2000 IFIP World Computer Congress*, pages 877–882, August 2000.

[6] 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.

[7] A. M. MacEachren. *How Maps Work: Representation, Visualization, and Design.* Guilford Press, New York, 1995.

[8] N. Miyake. Constructive interaction and the iterative process of understanding. *Cognitive Science*, 10(2):151–177, 1986.

[9] H. A. Muller, S. R. Tilley, M. A. Orgun, B. D. Corrie, and N. H. Madhavji. A reverse engineering environment based on spatial and visual software interconnection models. In *Proceedings of the SIGSOFT '92 Fifth Symposium on Software Development Environments*, pages 88–98, December 1992.

[10] G. C. Murphy and D. Notkin. Lightweight source model extraction. In *ACM SIGSOFT '95 Symposium on the Foundations of Software Engineering*, pages 116–127, October 1995.

[11] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: bridging the gap between source and high-level models. In *ACM SIGSOFT '95 Symposium on the Foundations of Software Engineering*, pages 18–28, October 1995.

[12] R. O'Callahan and D. Jackson. Lackwit: A program understanding tool based on type inference. In *Proceedings of the 19th International Conference on Software Engineering*, pages 338–348, May 1997.

[13] J. Preece. *Human Computer Interaction.* Addison-Wesley Publishing Company, Menlo Park, California, 1994.

[14] A. H. Robinson, J. L. Morrison, P. C. Muehrcke, A. J. Kimerling, and S. C. Guptill. *Elements of Cartography.* Wiley, New York, 6th edition, 1995.

[15] 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.

[16] R. K. Yin. *Case Study Research: Design and Methods.* Sage Publications, Newbury Park, CA, 1989.

[17] J. J. Yuan. Using the map metaphor to assist cross-cutting software changes. Masters Thesis, University of California, San Diego, Department of Computer Science and Engineering, April 2000.