UNIVERSITY OF CALIFORNIA, SAN DIEGO

Using the Atlas Metaphor to Assist Cross-Cutting Software Changes

A thesis submitted in partial satisfaction of the
requirements for the degree Master Of Science
in Computer Science

by

Wesley Y. Leong

Committee in charge:

Professor William G. Griswold, Chairperson
Professor Jeanne Ferrante
Professor James Hollan

2002

Copyright
Wesley Y. Leong, 2002
All rights reserved.

The thesis of Wesley Y. Leong is approved:

____

____

____

____

Chair

University of California, San Diego

2002

To my Mom and Dad

TABLE OF CONTENTS

Signature Page . . . iii

Dedication . . . iv

Table of Contents . . . v

List of Figures . . . vii

Acknowledgements . . . viii

Abstract . . . ix

I Introduction . . . 1
A. Evolving Software with Aspects . . . 2
1. Aspect-Oriented Programming . . . 2
2. Using Information Transparency to Find Aspects . . . 2
3. Aspect Browser . . . 3
4. Hypothesis . . . 3
B. Overview of the Thesis . . . 4

II Aspect Browser and the Map Metaphor . . . 5
A. Seesoft . . . 5
B. Map Metaphor . . . 6
C. Aspect Emacs . . . 8
1. Aspect Manager . . . 9
2. Text Editor . . . 10
3. Aspect Emacs Tools . . . 10
D. Nebulous . . . 10
1. File Panels . . . 11
2. Aspect Pattern Legend . . . 13
3. Zooming with the File Panels . . . 13
4. Traversing the Aspect Patterns . . . 14
E. Case Study with Aspect Browser . . . 14
1. Results and Observations . . . 15

III Aspect Browser and the Atlas Metaphor . . . 18
A. Scaling the View with the Atlas Metaphor . . . 18
1. The Limited Views of AB 1.1 . . . 18
2. The Atlas Metaphor . . . 19
3. Scaling the View of Aspect Browser . . . 19
B. Scaling Performance . . . 22
C. Other Modifications to AB . . . 24
D. The New Role of Aspect Emacs . . . 25
E. The New Features in Nebulous . . . 25
1. Magnifying Glass . . . 25
2. Aspect Index . . . 25
3. Tool Bar . . . 27
4. Menu Bar . . . 29
F. Discussion . . . 31

IV Benchmarks and Case Study . . . 32
A. Performance Benchmarks . . . 32
1. Switching Panels . . . 33
2. Traversing and Scanning Large Projects . . . 34
3. Matching Aspect Patterns in the Cache . . . 34
4. Conclusion . . . 35
B. Case Study . . . 35
1. Setup of the Case Study . . . 36
2. State of the Tool . . . 37
3. General Observations . . . 37
C. Detailed Observations and Analysis . . . 40

V Conclusion . . . 49
A. Contributions . . . 50
1. Designing the Atlas Metaphor for Crosscutting Changes . . . 50
2. Tool for Making Crosscutting Changes . . . 51
3. Unifying the Interface Between Multiple Software Tools . . . 51
B. Future Work . . . 51
1. Large Scale Case Study . . . 51
2. Understanding Caching . . . 51
3. Global Scanning and Updating . . . 52
4. Extending the Atlas Metaphor . . . 52
5. Visual Cues . . . 53

Bibliography . . . 54

LIST OF FIGURES

II.1 Aspect Emacs' Aspect Pattern Manager List 9
II.2 Aspect Emacs' Text Editor 9
II.3 Nebulous from AB 1.1 11
II.4 Aspect pattern legend in Nebulous, AB 1.1 12
II.5 The zoom panel in Nebulous, AB 1.1 13
II.6 The traversal panel in Nebulous, AB 1.1 13

III.1 View of Nebulous with the atlas metaphor 19
III.2 Simple conceptual example of a Reflexion model. 20
III.3 Directory Tab Panel 22
III.4 A directory panel with file panels and child directory tabs. 22
III.5 Aspect Index 26
III.6 The tool bar from Nebulous 27

ACKNOWLEDGEMENTS

I want to thank Bill Griswold for being a supportive advisor. I really appreciate the help and advice he had given me.

I want to thank my parents for supporting me through graduate school.

I want to thank Michael Copenhafer for helping me finish my thesis and for being a supportive friend. I would also like to thank him for volunteering for my case study.

I would also like to thank my two friends Derrick Kondo and Eric Wing for just hanging out with me.

I would also like to thank Joseph Kaine for his Linux support.

ABSTRACT OF THE THESIS

Using the Atlas Metaphor to Assist Cross-Cutting Software Changes

by

Wesley Y. Leong

Master of Science in Computer Science

University of California, San Diego, 2002

Professor William G. Griswold, Chair

As a software system evolves, the integrity of its modules falter because the software design cannot anticipate and localize every possible change. The modules develop dependencies and these dependencies are crosscutting concerns, or aspects. These dispersed pieces of code make further modifications difficult. Aspect Browser helps the programmer to manage and to find these aspects by displaying the code like a map. The map-like interface enhances spatial reasoning. It reduces recall with visual recognition because the display becomes a memory aid that reminds the programmer of the existence and location of each aspect.

An earlier version of Aspect Browser focused on the map metaphor, the designing of an interface and display that mimicked a map. A case study was conducted and it demonstrated that the map metaphor did assist the programmer in modifying a large software system with crosscutting concerns. However, the performance and the the user interface in this version of Aspect Browser did not scale well with large software.

The focus of this thesis is the atlas metaphor, a metaphorical design concept that extends the map metaphor and scales the performance and interface of Aspect Browser. An exploratory case study was conducted to test this version of the software tool. The case study did not reveal any strong evidence that the atlas metaphor enhance productivity, but it did generate some interesting insights about the map metaphor and the atlas metaphor. It also identified many opportunities for improvement.

Chapter I

Introduction

The short product cycles of software in a competitive market encourages developers to evolve existing software instead of writing it from scratch. Software evolution has made certain software design methods popular such as modularization and object-oriented programming. Modules create separation of concerns, hiding design decisions and anticipating future changes [Par72]. However, software evolution degrades modularity because not all design decisions can be completely contained within a module and not all changes can be anticipated. Many changes may have to be made to the software that do not fall along the boundaries of a single module. In the long run, modularity problems will appear in any evolving software system.

Non-modular code may appear because of the limitations of the programming language and the design methodology. Object-oriented languages like Java and C++ have language features such as classes, polymorphism, encapsulation, and inheritance that help define specific types of modularity. These typical mechanisms are good for defining types with private member fields and creating functional relationships within a hierarchical structure. However, they may fail in modularizing code that deals with security, performance optimization, and exception handling because the code is dependent on execution paths instead of well-defined types [VV00], [KLM+97].

Code that defies modularization and shares a common concern is called an aspect or a crosscutting concern. An aspect can impede the progress of software evolution because each relevant code fragment may have to be found and examined before the correct modifications can be made to the software. This process can be time-consuming and prone to error. The problem gets worse as the software evolves because the new modifications may introduce new aspects. Developers need an easier way to describe and visualize these code fragments.

I.A Evolving Software with Aspects

I.A.1 Aspect-Oriented Programming

Traditional programming languages like C++ or Java do not have any features to explicitly express aspects. Aspect-oriented programming (AOP) is a programming technique that is capable of handling and managing aspects with an aspect language. An aspect language is a programming language that can explicitly define aspects, usually with traditional programming entities such as functions and classes [KLM+97].

AspectJ and HyperJ are two example languages that can do AOP. Both of these languages are based on Java. AspectJ has features that explicitly define aspects with the join point model. The join point model uses the execution path of the software to mix in the code of an aspect with normal code [KHH+01]. HyperJ is based on the idea that modules can be described with multiple dimensions of concerns. Traditional software design usually concentrates on decomposing the modules in respect to a few dimensions of concerns. HyperJ can decompose software systems in respect to many other dimensions of concerns [IBM00]. Both of these languages may remove some problems associated with modularity, but these languages require special compilers to handle the aspect language features. Older software systems cannot take advantage of AOP without significant changes to the code and design.

I.A.2 Using Information Transparency to Find Aspects

Information transparency offers a compromise for existing software that was not originally programmed and designed with AOP. Information transparency is based on the concept of syntactic similarity. Similar looking code may share a common concern or a common behavior. Information transparency only works if the code was written with a consistent programming style such as variable naming conventions. It is independent of the language and compiler being used because it solely works on the syntax of the code[Gri01].

Information transparency can be used to find aspects by using a string matching program. The UNIX program `grep` matches a regular expression with a list of files. `grep` outputs the matching lines of code and indicates which file each match comes from. The output results of `grep` exhibit several immediate problems. First, the results are displayed in plain text and the screen can only show a limited amount of plain text on the screen. If the programmer is familiar with the code and `grep` returns a few matches, the programmer can find the concerns quickly. If `grep` has many results and the programmer is not familiar with the code or the programmer is looking for the complete set of matches relevant to a concern, the programmer has to examine the results in a linear fashion. The programmer has to remember the location and the specific

issues of each match and the programmer can become disoriented while sorting a large list of matches.

Second, the matching lines are printed to the screen out of context because the surrounding code is not displayed. `grep` has a feature to print a certain number of lines before and after the match. However, printing these lines just exacerbates the first problem because there is now more information to sort through. Not every match requires the same number of lines to print before and after because the relevancy of the code around the match will vary. This problem can be avoided by forcing `grep` to print each line number for each match and the programmer uses an editor to view the code itself. The process of moving the editor's view to each matching line is inconvenient.

Third, the programmer is responsible for his own bookkeeping of the regular expressions that are used in the query. Sometimes the programmer may want to combine multiple regular expressions to find an aspect that has multiple signatures or to find the relationships between multiple aspects. The programmer has to remember the exact queries for each result and figure out how to merge the results together. This task may not be easy, especially if the results are numerous. The programmer has no direct way to interact with his findings and reduce the results into a manageable size.

I.A.3 Aspect Browser

Aspect Browser (AB) is a software tool that manages and finds aspects with information transparency. It has a map-like interface that displays code in multiple dimensions without the visual limitations that arise from tools like `grep`.

The first version of AB was implemented with the map metaphor, a visual interface that displayed the code and the aspects like a map. It made the navigation and the examination of aspects easier. However, the interface and the performance of AB did not scale with large projects. The second version of AB integrated the atlas metaphor into itself. The atlas metaphor is a group of features that scaled the performance of AB with an atlas-like interface.

I.A.4 Hypothesis

We hypothesized that the atlas metaphor will affect the programmer's approach and behavior towards evolving software. The improvements in the performance of AB and the ability to load entire projects with AB will make the tool more convenient to use. AB's capabilities of defining and selecting specific views of the project will improve the programmer's visual cognition and the programmer's understanding of the code.

I.B Overview of the Thesis

Chapter 2 describes an earlier version of Aspect Browser that was developed around the concept of displaying code like a map. It also describes a case study that was performed with that version of Aspect Browser. Chapter 3 describes a more recent version of Aspect Browser based on the concept of the atlas metaphor. This version enables Aspect Bowser to view larger products without significant performance penalties. Chapter 4 discusses has a case study that was conducted with the newer version of Aspect Browser. Performance benchmarks were also conducted to show how long it takes to scan a large project for aspects. Chapter 5 is the conclusion.

Chapter II

Aspect Browser and the Map Metaphor

The Aspect Browser (AB) is a visualization tool that uses the map metaphor to help programmers manage and find crosscutting concerns in a software system. The development of the previous version of AB, designated as AB 1.1, was strongly influenced by Seesoft, a software visualization tool. AB 1.1 also borrowed many ideas from the field of cartography. AB was used in a case study to examine how well these borrowed ideas help a programmer perform crosscutting changes on a software system consisting of approximately 500,000 lines of code. The case study has revealed scalability problems when AB opens many files. This chapter discusses in detail about the role of the map metaphor in AB and the case study.

II.A Seesoft

AB was originally based on Seesoft, another software visualization tool that displays the age of each line of code [ESS92]. In Seesoft, all the source files are shown on the screen as narrow vertical windows. Each line in a window represents a single line of source code in the file and it is highlighted a specific color depending on its age. If the file is too long to fit on the screen vertically, it wraps around to the right starting from the top again. This display of multiple files with each line of code highlighted according to its age can reveal useful information such as possible problematic modules in the program that require frequent maintenance.

AB borrowed the general idea from Seesoft of displaying many files at once in a two dimensional display. AB shows the files of the project from left to right in small vertical strips. Each line in the window represents a single line of code. The lines of code that has a specific

aspect are highlighted with color.

II.B Map Metaphor

Cartography is the study of the creation and application of maps. Maps are a visual representation of information and their goal is to portray that information in an efficient and meaningful manner [Tuf83]. Useful maps are usually abstract representation of the data. The abstraction helps emphasize important data and deemphasize or remove non-essential data, making the maps easier to understand [MG90]. For example, a road map usually outlines the roads with lines and the cities appear as dots.

Cartography has realized that presenting visual data to other people is not the only application of maps, but they can also be used in the exploration and in the discovery of new facts or ideas. Maps do not have to be necessarily geographically related, but can also be applied to other fields such as in scientific visualization. Humans have a natural tendency to find patterns in the data when it is visually displayed [MG90]. The process of realizing the patterns and associating such patterns with a hypothesis is called *"visual thinking* [Mac94]. The hypothesis can be confirmed or tested by manipulating the visualization. This manipulation can be done relatively quickly with computers today, giving back immediate feed back. After the facts and conclusions are found, the spatial data can be filtered and placed into perspective so the target audience can easily understand what the data mean. This presentation of the results is *"visual communication"* [Mac94].

AB uses the map metaphor and gives the programmer a lever in understanding and managing the aspects that are dispersed across many files. The programmer can *"visually think"* about the problems and the solutions at hand with the AB. Combined with the Seesoft-like view, AB incorporates some cartographic features such as map symbols, colors, scaling, map legends, and map indexes[Yua00]. The following sections are a general description of each of the cartographic features included in AB.

Map Symbols

Map symbols are good for showing the location of discrete and dispersed objects over an area. The shape of the symbol is important. Usually the shape of the symbol is an already recognizable shape that readily describes what it is representing. The shape and size of the symbol can be used to describe another dimension of information about that specific object it is representing, but these multiple shaped or sized symbols may lead to confusion [Mac94].

Colors

Colors are used on a map to indicate specific types of information. They can be used to define areas on a map or create contrasts between two separate areas, like on a political map where no neighboring country can have the same color. However, colors can be assigned values, both discrete values and continuous values. Discrete values are best represented with different hues or colors. Continuous values are best displayed with gray scale or varying the brightness of a single color [Mac94].

Scaling

Scaling is important in maps. A small-scaled map, or a zoomed-out view, shows a large area far away. A large-scaled map, or a zoomed-in map, shows a small area upclose. The details should be adjusted accordingly to the scale too. Small-scaled views should filter out details to keep the map readable. Large-scaled views can have more details without clutter.

Map Legends

Map legends describe what the symbols on the maps mean. It can also be used to describe other pieces of information displayed on the map such as coloring and scaling.

Map Indexes

Map indexes provide a convenient way of discovering the location of a specific entity on the map. They allow a person to query the map in a different way, such as finding a position of a city in a alphabetically ordered list of cities.

Animated Maps and Computer Maps

Computer maps let users manipulate and view maps in many ways. Animated maps have the ability to change the view of the map along another dimension such as time. Watching the continents drift around the earth over millions of years is an example of an animated map. Animated maps include other changes of the map such zooming-in and zooming-out [Pet].

However, there are other ways of manipulating and interacting with maps on computers that does not modify the view. For example, plotting a path between two points or leaving a pin on the map to indicate a position. Such actions help solve the problem at hand visually and reduces the amount of required recall by using the map itself as a memory aid or as a scratch pad.

Visual Representation

By definition, geographical maps naturally have a graphical representation of their respective geographical areas. However, other types of information may not have an obvious form of creating a visual representation. Therefore, an effort must be made in designing a display model that can show the information in a convenient and usable manner.

In [Yua00], Aspect Browser must display the information in a consistent and persistent manner before the map metaphor can be used. First, an ordering must be enforced in displaying the files and directories of the source. This ordering creates some form of orientation. Further, the consistent ordering and the relatively static placements of the display allow the landmarks to exist. Recognizable landmarks reduces the cognitive problems associated with recall.

Aspect Browser 1.1

AB is a tool for software visualization that finds and manages crosscutting concerns in the code. Like a map, AB spatially shows the aspects and crosscutting concerns on an interactive display. It has the navigational features and the customizable views that prevent disorientation and reduce distraction from unnecessary visual information.

AB is a syntactic-based tool that makes it compatible with any programming language. It can handle projects that are written in multiple languages. AB does not require the parsing or compilation of the source code in order to operate. It just reads in the text of the source code.

AB is composed of two different software tools. These two tool are Aspect Emacs and Nebulous. Aspect Emacs manages aspects and provides text editing. Nebulous is the visualization program that creates the map-like views of the entire project. In short, Aspect Emacs provides administrative features of handling aspects and viewing code at a small scale. Nebulous provides the large scaled views of many files at once. The following sections describe the version of these two tools in depth that is inside of Aspect Browser version 1.1 (AB 1.1).

II.C Aspect Emacs

The Aspect Emacs tool that is part of AB 1.1, is an Emacs Lisp extension that has three primary functions. First, it manages a list of aspects and the files in the project. Second, it is a text editor that has the capabilities of showing the actual text of the files. Third, Aspect Emacs provide two tools to help search for possible aspects.

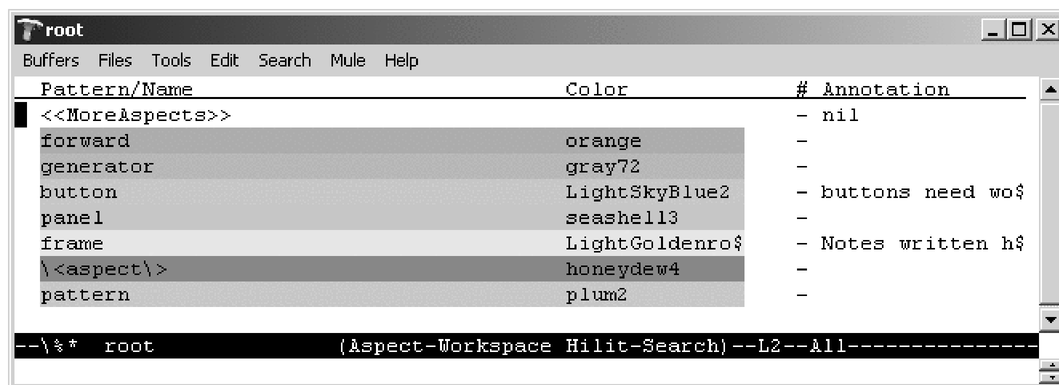

Figure II.1: Aspect Emacs' Aspect Pattern Manager List

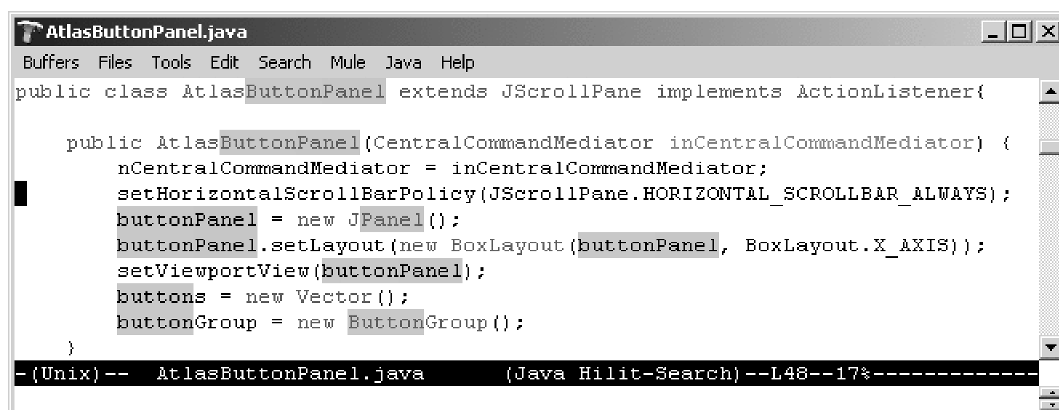

Figure II.2: Aspect Emacs' Text Editor

II.C.1 Aspect Manager

Aspect Emacs displays a list of aspects as shown in Figure II.1. Each aspect on the list is an aspect pattern associated with a color and an annotation. The aspect pattern, which is just a normal regular expression, is the pattern used for matching the text in the code. The matching text are highlighted with the aspect pattern's color. A default color is chosen by Aspect Emacs during the creation of an aspect pattern, but the color can be manually changed later. Aspect Emacs tries to choose the default colors that vary the most in hues because different hues are easier to distinguish. The annotation are notes or comments a programmer can attach to an aspect.

An aspect can be added, deleted, hidden, or shown on the list. Nebulous will change its map-like display of its aspects when these commands are invoked in Aspect Emacs. For example, hiding an aspect in Aspect Emacs will also hide the aspect from display in Nebulous.

Aspect Emacs are managed with *workspaces*. A workspace is a list of aspect patterns.

It can also contain other workspaces too, forming a possible hierarchical organization. Aspect Emacs begins with a default workspace that is ready for the addition of new aspect patterns and new workspaces. Entire workspaces can be hidden and shown; the aspects underneath a hidden workspace are all hidden and the aspects underneath a shown workspace are all shown.

Aspect Emacs reads in a text file, called *modules*, that has the list of files of the project. The files are partitioned into special aggregations called *directory panels*. These directory panels usually fall along the boundaries of the directory, but they can be customized to fit other software designs. These directory panels affect how the files are displayed, which is later explained in more detail.

The workspaces and the aspects can all be saved to file. Later sessions of using AB 1.1 can load these saved files and quickly restore the saved workspaces.

II.C.2 Text Editor

The Emacs editor automatically provides Aspect Emacs with text editing capabilities (Figure II.2). The text editing already gives an up close view of the actual text in the source. Any text in the editor's view that matches any of the aspect patterns' regular expressions will be highlighted with the aspect pattern's assigned color.

II.C.3 Aspect Emacs Tools

Aspect Emacs has two simple tools to help search for possible aspects. One tool searches for redundant lines of code in the program. The redundant lines of code are probably copied-and-pasted code, a strong indicator of an aspect [Gri01]. These lines of codes are kept in a special workspace and they can be copied to another workspace.

The second tool is an inference tool that scans the patterns of each word in the source. Many programming styles usually use some sort of naming conventions for variables. For example, some programmers may use capital letters as word boundaries in their variable names such as isVariableName while others may use underscores, such as is_Variable_name. These inferred aspects can be copied and pasted to another workspace.

II.D Nebulous

Nebulous is the visualization tool that generates the map-like views of the files. It is written in the Java programming language. This tool houses the majority of the map-metaphor features and provides navigational tools for tracking aspects. The following sections that describe the features of Nebulous will be referring to the version that comes in AB 1.1.

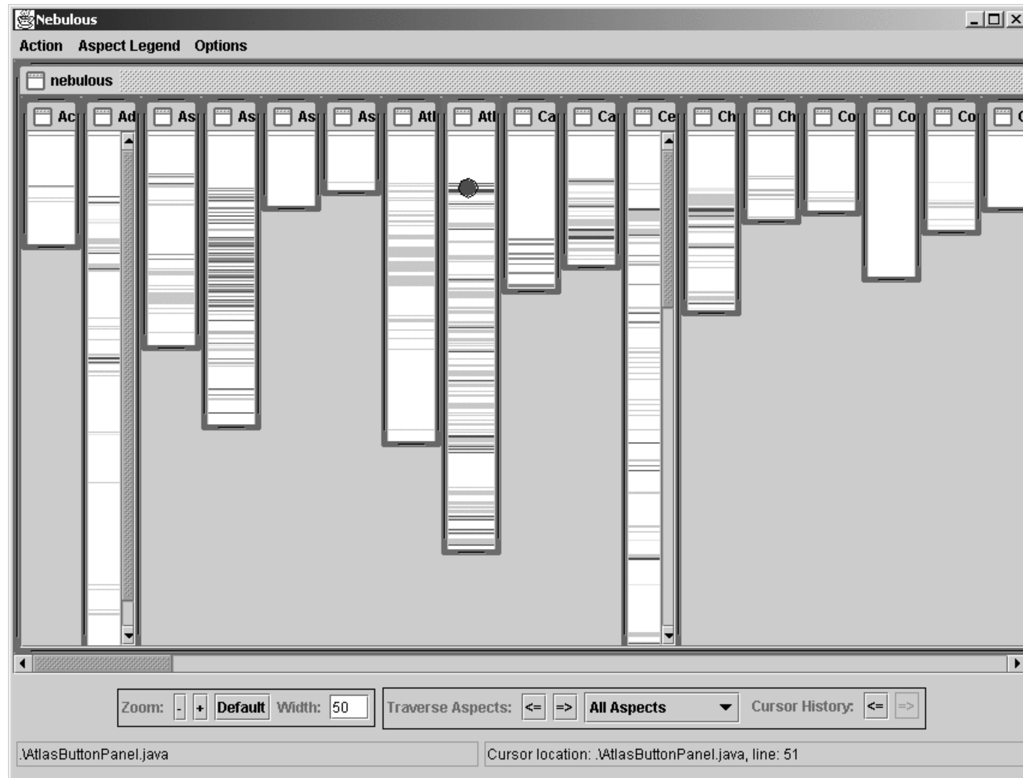

Figure II.3: Nebulous from AB 1.1

II.D.1 File Panels

The file panels are the main visual feature of Nebulous that creates a two dimensional map-like display. In terms of the map-metaphor, the file panels are a large-scaled view of the source while Aspect Emacs is the small-scaled view of the source. The simultaneous viewing of both views of different scales makes examining crosscutting concerns convenient.

The file panels are the little strips of windows that represent the files in the project (see Figure II.3). Each line, one pixel in height, in the file panel represents one line in the code. If the number of lines in the file exceed the space available to show the entire file panel on screen, a scroll bar is added to the window.

If a line of code has a matching aspect pattern, it will appear highlighted on the file panel with the aspect pattern's color. If more than one aspect pattern matches on a single line, the entire line is highlighted in red as an indicator of a collision. Collisions may indicate a possible relationship between multiple aspect patterns. These highlights on the file panels are like map symbols because they have spatial information about aspect patterns, such as how dispersed they are across the modules. The coloring helps distinguish between different aspect patterns. Aspect Emacs and Nebulous communicates to each other, so when an aspect is added, deleted, shown,

Figure II.4: Aspect pattern legend in Nebulous, AB 1.1

or hidden in Aspect Emacs, Nebulous will reflect those changes on the file panels accordingly.

The order in which the file panels appear in is important because proper placement of the file panels might reduce disorientation. The order of the file panels that appear on the screen can be adjusted. The default order is the file system. Other orderings include alphabetical ordering, increasing file size, and decreasing file size. File size can also be measured in the number of lines or in the number of bytes. The ordering of the file panels can be chosen in the **Options Menu**.

Further organizing and ordering of the file panels can be achieved with directory panels. A directory panel is a framed window that encapsulates a cluster of file panels. The title of the directory panel can be defined, but its title is usually the name of the directory that contains the file panels. In AB 1.1, all of the directory panels must be displayed with a side-by-side layout.

Interacting with the File Panels

The file panels are interactive. Double-clicking the mouse pointed at the file panel will create a red cursor at that spot. The red cursor is a marker to indicate *"I am here"*, just like a map with the present location marked on it. Aspect Emacs loads the corresponding file and then moves the text view to the location that was clicked on.

If there are too many file panels appearing on the screen, Nebulous will create a scrolling window that allows the browsing of all the file panels. If the red cursor is not on the screen, the cursor can be found by choosing the **Scroll To Cursor** command in the **Action Menu**. Then the scrolling window will move to the red cursor.

The position of the red cursor, the file and line number, is found in a text field on bottom right-hand corner of Nebulous. The bottom left-hand corner displays which file panel the mouse cursor is on.

Figure II.5: The zoom panel in Nebulous, AB 1.1

Figure II.6: The traversal panel in Nebulous, AB 1.1

II.D.2 Aspect Pattern Legend

A map legend lists the symbols and explain what they represent. Nebulous has its own legend which associates each aspect pattern with its color. The hit count, or the number of matches found, is placed next to the aspect pattern. This legend is accessible as a pull-down menu, marked as **Aspect Legend** on the menu bar. Figure II.4 shows this legend.

II.D.3 Zooming with the File Panels

One of the animated map features Nebulous has is its ability to change the scale of the file panels. The file panels default scale is one pixel per line of text. The scale can be adjusted by clicking on the + or - buttons as shown on Figure II.5. The + button zooms in, adding more pixels per line of text and increasing the width of the file panels. The individual lines of text will be represented by thicker lines on the file panels. The - button zooms out, making the scale smaller and narrowing the width of the file panels. The view is squeezed and certain lines of text will have to be grouped together. These groups will be represented by a single line on the file panel. Therefore, neighboring highlights will be merged into a single red line which represents a collision.

The zoom panel can adjust the width of the file panels. The file panels width is basically independent of the scale because the information is displayed on a line by line basis, which is a one dimensional display of the files. A text box is used as a numerical entry to adjust the file panel's width in pixels. This feature is basically a convenience for viewing the file panels. Shrinking the width allows more file panels to be viewed at once, but the new view can be cluttering. The scrollbars that appear in the file panels may obscure the view of the file panels if the file panels are too thin.

II.D.4 Traversing the Aspect Patterns

Nebulous has some navigational tools that help locate certain aspect patterns quickly. Figure II.6 shows the controls for traversing from one aspect pattern to another aspect pattern. This feature, referred to as the *traversal feature*, is like a map index, showing the locations of a specific aspect. The aspect pattern is chosen with the combo box, a text field combined with a menu. The combo box has an extra choice of traversing through every aspect collision, which can be handy for discovering relationships between multiple aspects. It also has the choice of traversing to the next neighboring aspect that is presently visible.

The left and right arrow buttons moves the red cursor on the file panel to the next appearance of the aspect pattern. The left arrow button visits the previous matching pattern and the right arrow button visits the next matching pattern. The next and previous matches are defined in a complicated way because the layout of the file panels and the lines of the codes are not continuous. The next match is considered to be the closest match that occurs either below the red cursor or in a neighboring file panel to the right of the red cursor, scanned top to bottom. The previous match is the closest match above the red cursor in the same file panel or in a neighboring file panel to the left of the red cursor, scanned bottom to top.

All appearances of the aspect pattern in the code can be visited this way and the linear order of the visits prevent disorientation when examining the aspect patterns. Every time the red cursor is moved to the next aspect, Aspect Emacs updates the editor display to show the code at the location of the red cursor.

The positions of where the red cursor has visited is recorded and can be traversed with the *cursor history* feature. Two arrow buttons are provided on the interface to traverse through the history of the red cursor in backwards or forwards order.

II.E Case Study with Aspect Browser

The case study described in this section was performed by the previous tool author of AB. For a comprehensive coverage about this case study, refer to [Yua00]. The following description of the case study below is a summary of the case study for the reader's convenience. It describes some of the flaws discovered in AB 1.1 and gives perspective about the solutions and changes made to AB after AB 1.1.

The Bioengineering department of UCSD had an advance finite analysis program written in Fortran. The program was going through some system-wide changes in removing an abandoned feature out of the program. This refactor provided a perfect opportunity to perform a case study with this software and AB. The case study was to determine if the programmer's interaction

with AB follows the map metaphor by monitoring the programmer's gestures and behavior. The second goal was the determination if AB increases productivity. The last goal of this case study was to find flaws within AB so future versions can compensate for these flaws.

The advance finite analysis was written with 500,000 lines of Fortran and several thousand lines of C code. The program was organized into 20 different directories with a total of over 2500 files. The total size of the source was approximately 40MB large. The computer that was used to modify the code with was a SGI Octane Irix (UNIX) computer.

The task that was performed during the case study was to remove an abandoned feature from the software called *regions*. This change crosscut across many directories and files.

The observational method of the project is called constructive interaction where paired programmers work together [Myu86]. Programmers can talk to each other about what they are planning to do instead of forcing the unnatural behavior of forcing a single subject to describe his thoughts verbally as he works.

The primary programmer, or subject, was the person who was the expert of the existing software. The partner was to be a tool author who was fluent in Fortran. The tool author shortly explained to the first programmer how to use AB and some of its features. The tool author was not to enforce a specific method of using the AB tool because part of the project was to observe the natural tendencies of using AB.

The computer monitor was videotaped during the experiment to examine how the subject used the AB. What the subject said during the course of the experiment was recorded too along with the videotape.

The actual version of AB that was used in this case study was not AB 1.1. It was an earlier version with the following missing or changed functionalities. First, only exact matching was supported, not the full regular expression. Second, tool tips were used to indicate which file the mouse cursor is pointing at instead of using a text field. The tool tips were removed in AB 1.1 because it took too much time for them to appear. Third, the order of the file panels could not be customized and the default order was the file system's order. Fifth, hierarchical workspaces was not implemented yet. Sixth, the list of the files in the project, which was stored in the file *modules*, had to be generated by hand.

II.E.1 Results and Observations

The *regions* feature was completely removed from the software after 18 hours of programming. The subject had created 128 aspect patterns during the entire experiment. The software was compiled and tested. The software ran a simulation of an electrical wave propagating through the heart. The simulation completed without any problems and the performance of

the program was increased by a rough estimate of 25%.

The case study had demonstrated that the map metaphor did help the subject tackle some of the problems associated with the task. The subject only viewed the files on a directory basis. In the many directories he had visited, he first adjusted the view in Nebulous by narrowing the file panels and changing the zoom level. He then examined the left most file panel for highlights. If there was a highlight, he double-clicked on it to bring up a zoomed-in text view of the code in Aspect Emacs. When he reached the last highlight in the file panel, he continued the same process on the neighboring file panel to the right. This routine behavior indicated that the order of the file panels and the highlights in Nebulous had created a road map for him. This road map usually kept him oriented in the code and reminded him where he was. While the road map gave him a large scaled view of the directory of files, Aspect Emacs had given him a convenient view of the code at the highlighted spots.

The subject used the navigational tools in AB. The subject would usually scroll in the editor view to the next highlight if the highlights in the files are close together. If they are separated far apart, he double-clicked the next highlight on the file panel. If the next highlight is not on view on the file panel and there is a scroll bar on the file panel to indicate its longer than the window, he sometimes used the traversal feature to locate the next highlight. However, his choice of navigational means was not always consistent in each case.

The subject referred the aspect patterns by their associated colors. He also referred the colors as specific types of tasks too because certain aspects represented a type of change to the code. For example, the subject said during the experiment, “Look at all that green. There’s four new aspect right there, baby.” Further, repeating patterns of the same highlights across many files may indicate that these files have a common structure and a common behavior. The subject have encountered a couple of these occurrences and immediately understood what those files did and how to modify those files. The associations between the colors, the tasks, and the patterns of the highlights make it easier for the subject to identify what needs to be done, especially when visually displayed on the file panels.

The colors of the aspect patterns became even more useful because of the nature of this project which was to remove the *regions* feature. The *regions* feature in the code was reduced to a bunch of highlights in AB. The subject had to change the code at those specific spots in the code and make the highlights disappear. He once said, “I want all this to be white.”

Scalability Issues

The subject viewed the code at one directory at a time for two reasons. First, he discovered that working on the code at the directory level had created manageable partitions.

He compiled the code after modifying the files in the directory to check for new errors. Second, AB suffered from both performance and memory problems when the entire project was loaded into memory. The performance loss of AB made the interaction with AB difficult and the benefits of having all the files loaded into view did not overcome the performance loss. The slow responses from AB inhibited the subject's performance because his immediate goals had to be remembered for longer periods of time with the risk of disorientation, or forgetting what he was planning to do [DRO98].

During the case study, the subject overcame these scalability issues by loading one directory at a time. As he finished modifying one directory and moved to the next one, he had to copy both the *modules* file and the index of the aspect patterns over to the next directory. These files saved him the time from reentering project files and aspects again, but the process was slow. The process probably disrupted his train of thought if he needed to examine another directory before making his final modifications. He only revisited a few directories with AB in the case study which may indicate that each visitation was costly.

The next chapter describes the atlas metaphor, a new addition to AB that addresses the scalability issues of viewing many files in different directories. Other modifications besides the map metaphor were added to AB to help enhance the work flow and the ease of use.

Chapter III

Aspect Browser and the Atlas Metaphor

This chapter covers the features that have been added since Aspect Browser version 1.1 (AB 1.1). One of the changes include the incorporation of the atlas metaphor which scales the interface and display of AB. The second group of changes scales the performance of AB and reduces the performance penalties from loading many files. The third group of changes improves the work flow of AB by consolidating the map-like interfaces into one common interface.

III.A Scaling the View with the Atlas Metaphor

III.A.1 The Limited Views of AB 1.1

The case study in Chapter II had demonstrated that the map metaphor had an impact on both the subject's approach to the problem and the subject's behavior. The subject started to visualize and discuss the problems in map-like terms. The spatial displays helped the subject modify the code across many files.

However, AB 1.1 had performance problems and memory problems when too many files are loaded. It can only display one directory at a time and it had no convenient means of switching the display to a different directory. The subject had to start AB every time he wanted to examine the next directory. The subject had to copy the index of aspects and the list of the project files to that directory before starting AB. This process was cumbersome.

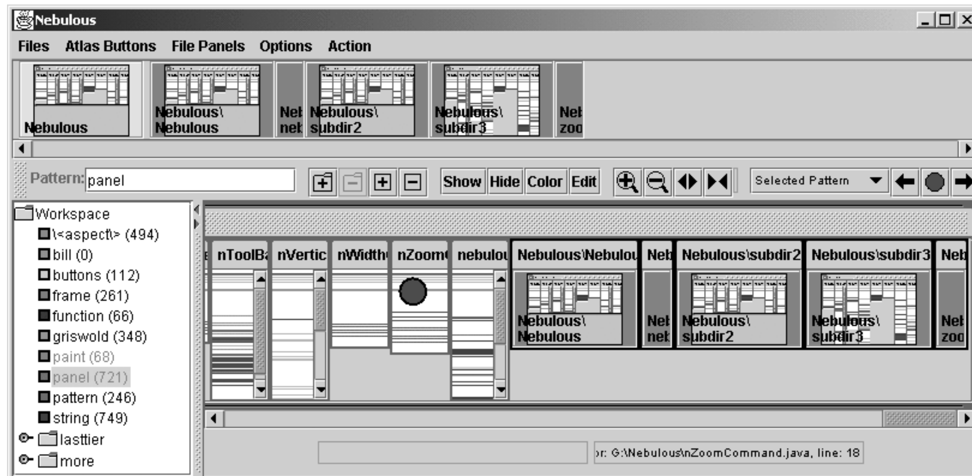

Figure III.1: View of Nebulous with the atlas metaphor

III.A.2 The Atlas Metaphor

A typical atlas is a collection of maps displayed at varying scales. An atlas usually has an overview map that shows the entire area. This expansive map has the role of showing where everything is without details. The other maps in the atlas are a large-scaled representation of specific areas on the overview map. These zoomed-in maps show more details and they concentrate the focus of a specific area by constraining the view. An atlas can also contain maps that show different types of information such as political boundaries or geographical terrains.

An atlas needs an organized scheme in indexing the maps because a poorly organized atlas frustrates the viewer and prevents the viewer from focusing on the intended problem. An example of an organizational scheme for a world atlas is to group the maps of each country into the continents they belong to. The continents can be organized in alphabetical order. The atlas may have an index that lists each country in alphabetical order with its page number. Physical tabs can be placed on specific pages of the atlas to indicate the major divisions inside the atlas. All of these organizational schemes make the atlas more convenient to use.

III.A.3 Scaling the View of Aspect Browser

AB uses the atlas metaphor to scale the view of the files. AB divides up the view of the files into manageable partitions called *directory panels*. As mentioned in Chapter II, directory panels are aggregates of files. Like each map in an atlas, we conceptualized that each directory panel represents a directory. We have chosen the directory as the unit of display because the subject in Chapter II worked on one directory at a time [Yua00] [GYK01].

The directory panel can contain files and subdirectories. The subdirectories in the view

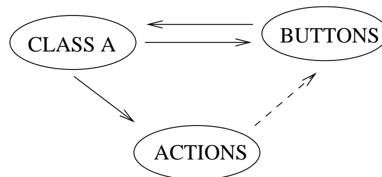

Figure III.2: Simple conceptual example of a Reflexion model.

are represented as other directory panels. Although the directory panel can represent the view of a single directory, the contents of the directory panel are defined by the programmer. These customized directory panels can reflect different software architectures that do not fall along the boundaries of the file system.

As in AB 1.1, Nebulous draws the directory panels as framed windows. The framed windows encapsulate their contents which includes file panels and other directory panels. The subdirectories inside a parent directory panel are displayed in one of two ways. The subdirectories can be recursively displayed like their parent directory panel, with their file panels and subdirectories. The second way of representing the subdirectories on a directory panel is with some symbolic buttons called directory tabs (see Figure III.4). Directory tabs are explained later in this section.

AB needs an interface that selects and displays the directory panels in the project. This interface must organize the display of the directory panels and show the relationships between directories and subdirectories. Second, each selectable item in the interface must be readily identifiable. Third, each item must give cues about its contents. We originally considered displaying a diagram that mimicked a Reflexion model [MN97]. These diagrams are composed of nodes, each representing a directory panel. Lines between the directory panels represent some sort of relationship or dependency, such as function calling and subclassing (see Figure III.2). The directory panel is displayed by selecting the corresponding node.

The construction of a Reflexion-like display is not a good solution. If function calls or subclassing relationships are to be shown on the diagram, AB would have to process the code before hand. AB would stop being a pure lexical tool and it would have to be targeted for a specific programming language. The manipulation and organization of this new display would require a new interface which would add more complexity to AB. The display of nodes and lines would not consistent with the file panels. The correlation between the two views would be disjointed because one view does not directly compliment the other view.

We decided to implement a different interface that unifies the expansive view with the file panels. For space efficiency, the depiction of the contents of the file panels was emphasized and the relationships between the directory panel was deemphasized. We created buttons called

directory tabs that represent each directory panel. Clicking on a directory tab selects and changes the view to the corresponding directory panel. We placed these directory panels in a single row on top of Nebulous. This row of buttons is referred to as the **Directory Tab Panel** (see Figure III.3). The relationship between directory tabs are indicated by inserting large gaps between certain directory tabs. Directory tabs that are siblings directory panels are grouped tightly together. Large gaps are placed between each of these groups. The order of the display of each of these groups is the depth first search order. If all of the directory tabs inside the **Directory Tab Panel** do not fit on the screen, a scroll bar is added.

Each button in the **Directory Tab Panel** is labeled with the name of the directory panel on it. If the directory panel's name is a directory path, the button label is split into three tiers. The directory's name is placed at the bottom of the button. The parent directory's name is placed at the middle of the button and the remnants of the path is placed at the top of the button. The vertical locations of the names make it easier to compare ancestry of the directory panels. For example, the directory panels that have the same parent will have the same label on the second line. The number of lines on the label gives a quick idea of the depth of the directory panel, especially if the depth is less than three directories. The names on the buttons are truncated if they do not fit the width of the button.

A directory tab must have an icon that shows some useful information about its directory panel such as the highlights and sizes of the files. We had considered several abstract indicators such as colored dots and bar charts that represent the most frequent aspect patterns. We chose the icon to be a condensed picture of the directory panel and the file panels.

The reduction of the size of a raster image is lossy. One way of reducing the image with little distortion is to use an anti-aliasing technique found in Information Mural [JS98]. The Information Mural technique converted the color values of the picture into intensity values as it scales the size of the picture. However, the colors of the highlights would be lost during the transformation into intensity values. Because the colors of the highlights are important, we decided to use the a normal scaling algorithm found in the Java Swing library. Nebulous constructs the entire image of the directory panel in memory. It then shrinks the image into a thumbnail. If the directory panel is too large, the directory panel is truncated because the scaling should be the same across all of the directory tabs. This simple thumbnail is useful because it creates an alternative visual of the button. It also gives cues about the directory panel such as the size and highlights of the first several files. The thumbnail may remind the programmer of the role of that directory panel and the tasks associated with that directory panel [KS98].

Although these thumbnails create landmarks in the index and relieve recall, the generation of the these thumbnails is computationally expensive. The thumbnails are only generated

Figure III.3: Directory Tab Panel

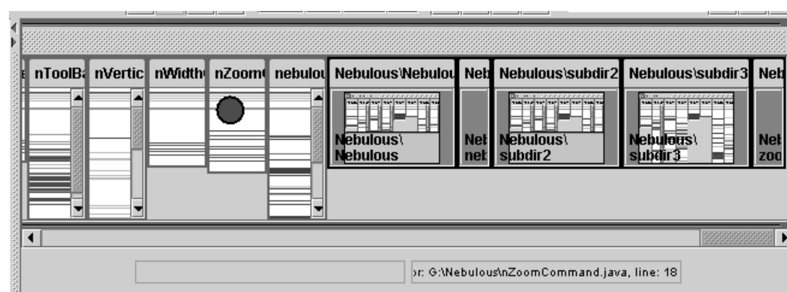

Figure III.4: A directory panel with file panels and child directory tabs.

or updated when the directory panel is visited or left. Therefore, the directory tabs without a thumbnail are the directory panels that have not been visited yet. Only their names are printed on the directory tab. This beneficial side effect makes it easy to spot previously visited directory panels. If only a few directory panels are of interest in the project, the appearances of a few thumbnails catches the focus of attention.

The directory tabs without file panels are smaller in width (see Figure III.3). Narrowing the widths of these directory tabs enhances the focus on other directory tabs with actual content. Keeping these empty directory tabs is important because they show the relationships with other directory panels. The smaller directory tabs reduce the amount of scrolling in the **Directory Tab Panel** and more directory tabs can fit on the screen.

As mentioned before, the subdirectories of a parent directory can be optionally drawn on the screen as directory tabs inside the parent directory panel. These directory tabs are placed to the right of the file panels as show on Figure III.4. The placements of these directory tabs gives the programmer a feeling that he is looking at the actual directory. These directory tabs can be selected to change the view.

III.B Scaling Performance

Both performance and memory problems arise in Nebulous when too many files are loaded. The file panels consume memory because they need graphical resources and a copy of the files in memory. The more code there is, the more time Nebulous needs to scan for matches in the code. All of these factors slow down performance and increase the usage of memory. If the

amount of memory required is too high, Java's garbage collector may exacerbate the performance problems.

These scalability problems can be avoided by breaking down the monolithic view of AB 1.1 into multiple views. These smaller views are the directory panels because the directory panels provide a natural mechanism that restricts the number of files in memory. The matching of aspect patterns is only constrained to those files in memory. However, the assumption is made that a single directory is small enough to fit comfortably into the computer's memory. If the directory has too many files, the files can be partitioned further into user-defined directory panels.

Although loading only one directory panel into the display avoids the scalability problem, changing from one directory panel to another can be time consuming. Nebulous has to read in the new files and then match all of the aspect patterns. The required time for these operations is dependent on both the amount of contents inside the files and the number of aspect patterns. Matching the aspect patterns is the biggest potential bottleneck, especially if they are regular expressions.

The time required to switch between directory panels can be reduced with a cache. Nebulous has a cache and each cache entry can contain a single directory panel. This cache has a limited number of cache entries and the number of entries can be set by the programmer. This cache can only have compulsory misses and capacity misses because the entries are placed in a flat index. Compulsory misses occur when Nebulous has to fetch a directory panel for the first time. Capacity misses occur when a directory panel is accessed again by Nebulous, but that directory panel was previously removed from the cache to make space for another directory panel.

The cache checks if the directory panel is already in the cache when the display is changed. If the directory panel is in the cache, Nebulous does not have to read the files from the disk. It just draws the directory panel's window. If the request results in a cache miss, the directory must be read from the disk.

The cache entries also remember the highlights found in the directory panels and Nebulous does not have to rematch all the aspect patterns when a cache hit occurs. However, if a large number of new aspect patterns were added since the last time the directory panel was visited, the amount of time to match the new aspect patterns would be noticeable, and perhaps intolerable, to the user. This problem was alleviated by scanning all the entries in the cache when a new aspect pattern is added. The amount of time required to match a single aspect pattern with all the cache entries will take longer than scanning just a single directory. However, the extra time required for scanning the other cache entries should be small unless the cache has too many large

directory panels. All other forms of content manipulation such as deleting aspect patterns and hiding aspect patterns are done with the contents in the cache.

The cache uses a generational least-recently-used policy. Each cache entry has two age fields, the total age and the last-used age. If the cache has no room left, the entry with the oldest last-used age is removed. If two cache entries tie, the entry with the oldest total age is removed from the cache. This policy is not yet known to be the best policy because it has not been thoroughly tested yet. We assumed that the programmer will visit certain files repeatedly.

Nebulous has some viewing options that can show multiple directory panels. These viewing options cause problems with memory because if too many directory panels appear on the display, the contents in the cache combined with the contents on the display may degrade performance quickly. This problem was solved by using the cache to contain the contents of the display too. The directory panels on display are immediately added to the cache. All manipulations to the displayed contents are performed as if they are cached entries, which simplifies the design of Nebulous. A problem occurs when the number of directory panels on display is larger than the cache index. In this case, the cache temporarily grows to accommodate the directory panels on display.

The performance of the cache depends on the user's habits and the viewing options that are used. The size of each directory panel makes a difference because large directories may slow the system down and cause memory problems. The programmer has to break up the directory panels into smaller directory panels. However, a more compromising way of tweaking performance is the adjustment of the cache size. The cache size in Nebulous is manually adjustable. We need to further study the effects of the cache size on performance and memory usage, but the intention of having a manually adjustable cache size is to give the programmer the option to balance the performance of switching between directories and the usage of memory.

III.C Other Modifications to AB

Aspect Emacs in AB 1.1 had several problems. First, the text-based interface was not consistent with Nebulous's interface. Aspect Emacs disrupts the train of thought with Nebulous and the programmer loses focus. Second, the Emacs editor had to be used with Nebulous. Any programmer that does not know how to use the Emacs editor will not receive the full benefits of AB 1.1. Although these problems were not an issue in the case study mentioned in Chapter II, these problems were discovered during the development of AB. The following sections will discuss the changes that were made to AB in solving these problems. Figure III.1 shows the new GUI of Nebulous.

III.D The New Role of Aspect Emacs

Nebulous has replaced most of the functionalities of Aspect Emacs. Nebulous can now be launched or executed without the support of Aspect Emacs. Aspect Emacs is now only used as an optional vehicle to display text and highlight matches. However, Aspect Emacs can still add, delete, show, and hide aspect patterns. These commands are passed to Nebulous. Nebulous then manages the aspect patterns and workspaces accordingly. The removal of the many dependencies between Aspect Emacs and Nebulous simplifies the future additions of having other editors work in conjunction with Nebulous.

Aspect Emacs has a new command to bookmark a specific place in the text view and place a red cursor at that spot on the file panel. This bookmark is recorded by Nebulous and it can be visited later by using Nebulous's traversal feature.

III.E The New Features in Nebulous

Figure III.1 is a picture of the new interface of Nebulous. The new interface has the magnifying glass, **Aspect Index**, tool bar, and the menu bar. The magnifying glass gives a text view of the file panels. The **Aspect Index** displays the workspaces and aspect patterns. The tool bar has many commands for managing the aspects and the views of the file panels. The menu bar has the commands for saving files, changing the views, and performing other commands. Each of these items are described in the following sections.

III.E.1 Magnifying Glass

If Aspect Emacs is not used, the magnifying glass appears when a programmer double clicks on a file panel. The magnifying glass is a window with a zoomed-in-text view of the file. The window is automatically scrolled to the location of the double-click. Any lines with matching aspect patterns are highlighted. The rest of the file can be viewed with scrollbars on the sides of the window.

If another spot is double clicked again, the view in the same window jumps to the new spot. If the right button is double-clicked instead, a new window pops up. These features makes it easier for a programmer to view multiple files at once.

III.E.2 Aspect Index

The **Aspect Index** is a large panel on the left side of Nebulous that displays the aspect patterns and the workspaces. A close up view of this index is shown in Figure III.5. The **Aspect**

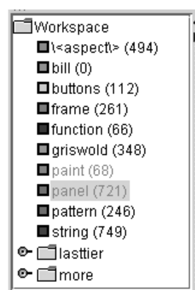

Figure III.5: Aspect Index

Index replaces two features from AB 1.1., the **Aspect Legend** from Nebulous and the **Aspect Manager** from Aspect Emacs. It is an interface for selecting and manipulating both workspaces and aspect patterns. The actions that can be performed on the entries include adding, deleting, hiding, showing, and changing colors. The tool bar, described in detail below, houses the buttons that perform these functions.

Each aspect pattern in the **Aspect Index** is displayed with its regular expression, color, and hit count. The color of the aspect pattern is represented by a square icon filled with the aspect pattern's color. The hit count is the number of hits found on the directory panels that are presently on the display. If the aspect pattern is hidden, the regular expression is grayed out.

A workspace in the **Aspect Index** is a folder that is displayed with its name and icon. The default icon for a workspace with no color is a folder. If a workspace has an assigned color, the icon is a large square that is filled with the color of that workspace. A hidden workspace has its name grayed out. When a workspace is unfolded, it reveals its aspect patterns and workspaces. The aspect patterns are first displayed in alphabetical order before the workspaces are displayed in alphabetical order.

The annotations of the aspect patterns can be added, viewed, or modified through a special dialog box. This dialog box only appears when an aspect pattern is right clicked.

The same aspect patterns can be inserted multiple times into the **Aspect Index**, but only in different workspaces. The deletion of such an aspect pattern only removes the specifically chosen copy from the index. All other copies in the index are hidden so the highlights on the file panels disappear as feedback. Hiding or showing an aspect pattern affects the other copies simultaneously.

The **Aspect Index** has a feature that automatically condenses three or more neighboring hidden aspect patterns into special folders, or hidden aspect folders (HAF's). We hypothesized that most programmers would only have a few aspect patterns activated at any given time. By placing the hidden aspect patterns into the HAF's, the view of the **Aspect Index** would be less

Figure III.6: The tool bar from Nebulous

cluttered. The activate nodes or the nodes currently shown would appear more pronounce than the hidden nodes. The HAF's are labeled with the names of the first and last entries of the folder with ellipses inserted in between them. If any of the hidden aspect patterns is activated, the folder is destroyed and the active patterns are returned to their original workspace. Any clusters that contain less than three hidden aspect patterns are also returned to their original workspace. Any clusters that contain three or more hidden aspect patterns are inserted into new HAF's.

III.E.3 Tool Bar

The tool bar has buttons that perform a variety of actions. The tool bar can manipulate and manage aspect patterns and workspaces. It can modify the views of the files panes and it has the traversal feature. This tool bar is shown in detail in Figure III.6,

Any command that directly manipulates an aspect pattern or a workspace is applied to the currently selected entry in the **Aspect Index**. These commands, which are described in detail below, include hiding, showing, adding, deleting, changing colors, and editing.

Adding Aspects Patterns and Workspaces

A workspace must be selected before an aspect pattern or a workspace can be added to it. The name of the workspace or the regular expression of the aspect pattern must be entered into the **Pattern Text Field** before clicking on one of the two add buttons. The two different add buttons exist because Nebulous needs to know if the content in the **Pattern Text Field** is intended to be an aspect pattern or a workspace.

Deleting Aspect Patterns and Workspaces

The deletion of an aspect pattern removes the aspect pattern from the **Aspect Index**. Any remaining copies of the aspect pattern are hidden. The deletion of a workspace recursively removes all its contents before deleting the workspace itself.

Hiding or Showing Aspect Patterns and Workspaces

Showing or hiding an aspect pattern makes the highlights on the file panels to appear or disappear. Showing or hiding a workspace recursively shows or hides all of its contents. Hiding grays out the entry in the **Aspect Index**.

Changing the Color of an Aspect Pattern or Workspace

Changing the color of an aspect pattern brings up a dialog box with a selection of colors. All the highlights will be updated with the new color.

Changing the color of a workspace applies the new color to all of the immediate aspect patterns residing in that workspace. Again, the color is chosen from a dialog box and the dialog box has a special option of removing any existing color associated with that workspace.

Editing an Aspect Button or Workspace

The edit button performs different actions depending what type of item is selected in the **Aspect Index**. If an aspect pattern is selected, the edit button gives the programmer the option to edit the annotation of that aspect pattern. If a workspace is selected, the edit button gives the programmer the option to change the name of the workspace. These two commands are fused together into this button in order to save space on the tool bar.

Changing the View

The tool bar has a zoom-in button and a zoom-out button that modifies the viewing scale of the file panels. The scale of the file panels are expressed in the number of pixels per line of text. Decreasing the scale or zooming out decreases the number of pixels that represent one line of code. Decreasing the scale far enough can force multiple lines of code to be represented by one pixel. Increasing the scale or zooming in increases the number of pixels that represent one line of code.

The tool bar has two more buttons that widens and narrows the width of the file panels. The display can fit more narrow file panels on the screen. However, very narrow file panels may clutter the view and may make the manipulation of scroll bars inside the file panels difficult.

The Traversal Feature

The controls for the traversal feature are placed on the right-hand side of the tool bar. The traversal feature is a navigational aid that makes the visitation of the aspect patterns easy. As each aspect pattern is visited with this navigational tool, a red cursor is drawn on the file panel to show the current location. The text view in the magnifying glass or in Aspect Emacs is moved to the new location too.

The controls of the traversal feature on the tool bar include three buttons and a pull-down menu. The first button is the **Forward Button** and it moves the red cursor to the next pattern. The second button is the **Backward Button** and it moves the red cursor to the

previous pattern. The last button is the **Bookmark Button** and it moves Nebulous's display to the current location of the red cursor if the red cursor is not already on display. The pull-down menu selects the mode at which the aspect patterns are traversed in. The different traversal modes includes traverse of all aspect patterns, traverse collisions only, and traverse only the selected aspect pattern from the **Aspect Index**. The fourth mode, traverse through cursor history, is not really part of the pattern traversing, but it offers the feature of traversing through the history of where the red cursor has been.

The traversal feature has been modified to work with the atlas metaphor. If the last match on the display has already been found, the next match is searched in the other directory panels. If no match is found in the other directory panels, the cursor stays in its present position. If a match is found in another directory panel, the directory panel that has the match is loaded into view. The red cursor is automatically placed on the new directory panel. The order of the search is the same order of the directory tabs displayed on the **Directory Tab Panel**. The direction of the search is left to right if the **Forward Button** is pushed or right to left if the **Backward Button** is pushed.

III.E.4 Menu Bar

The menu bar has the less commonly used commands and options. The following sections discuss the possible actions and options that exist in the menu bar.

Adding Project Files Into AB

In AB 1.1, the *modules* file had a list of files for Nebulous to display. Aspect Emacs created a default *modules* file by asking the user for a single file extension. However, any future additions of files had to be done by hand by editing the *modules* file.

The newer version of Nebulous made the addition of project files convenient. The **Quick Load** feature has a quick way of loading many files in at once. The programmer enters the file patterns using wild-card matching, a limited form of regular expression matching used by most UNIX consoles. The file patterns are stored on a list. Nebulous then matches the files in the current working directory with the list of file patterns. The matching recursively continues down the subdirectories. A full path or a relative path can be added as a prefix to each file pattern if the search is to take place in another directory beside the current working directory. Nebulous automatically generates the directory panels for each directory it encounters and places the files into their respective directories panels.

The **Add and Remove Files** option brings up a dialog box that gives the programmer the ability to define the directory panels. A programmer may prefer to use the directory panels

as a way to define software modules if the directories do not coincide with the boundaries of the modules. This feature is especially helpful in grouping certain files together that are already known to have specific concerns, but they are scattered between different directories. The programmer first generates a list of directory panels. Then the programmer assigns the files and file patterns to each directory panel. If the programmer wants Nebulous to recursively scan a directory, the directory panel's name has to be the directory path with an asterisks placed at the end.

Saving and Loading Sessions

The files in the project and the aspect patterns in the **Aspect Index** can be saved to a file. Nebulous can also load these saved sessions. The **Save Session** and **Load Session** actions are accessible in the **File** menu on the menu bar.

Nebulous Display Options

The display order of the file panels can be modified. Alphabetical order is the default display order. The other possible display orders are increasing file size and decreasing file size.

The directory panels have three different display modes. The default mode shows only one directory panel. The second mode shows only the directory panel and its immediate child panels. The child panels are recursively displayed inside the parent panel with their file panels, but the recursion stops at the immediate child directory panels. The third viewing mode draws all of the descendants and the descendant's file panels recursively. Each descendant is fully displayed with their file panels.

Empty directory tabs, or directory tabs that have no files, have the option to be narrowed to one-quarter of their normal size. The labels and the thumbnails on these narrowed directory tabs are truncated, making them virtually unidentifiable. These combined visual effects reduces clutter. The directory tabs with file panels become the focus of attention.

Redundant Lines

Redundant lines of code may indicate the possibility of an aspect because code that looks similar may have similar behavior [Gri01]. Nebulous can create a list of redundant lines of code, ranking them from most to the least frequent appearing lines of code. All spaces in the front and rear of the string are removed before making the comparison. After the list is compiled by Nebulous, the list is shown in a dialog box. A line of code can be selected and added to the **Aspect Index**.

III.F Discussion

AB 1.1 had some scalability problems when displaying a large number of files. It had no means of dividing the files up into manageable partitions, therefore it had to display the entire list of files. If poor performance or memory problems occurred from displaying too many files, the programmer had to reduce the view to a manageable size. The other files could not be viewed unless AB 1.1 was restarted. The process of switching between the different views was tedious and lengthy.

AB with the atlas metaphor has the ability to break down large projects down into manageable views. These smaller views can be defined by the programmer, but by default, AB defines these views along the boundaries of the directories. A view is visited by selecting one of the **Aspect Tabs**. Each **Aspect Tab** is labeled with the name of the directory panel and a thumbnail for easy identification. The caching reduces the switching time between frequently visited directory panels. The atlas metaphor makes the exploration and management of aspects in large projects easier and quicker.

The work flow of AB is now more consistent and homogeneous after moving the functionalities of Aspect Emacs into Nebulous. The **Aspect Index**, the directory tabs, and the tool bar concentrate all of the interactions with Nebulous. Programmers can now use AB without knowing Emacs because Aspect Emacs is now optional.

The next chapter is about a case study done with AB and the map metaphor. The case study is to determine how the programmer interacts with AB and how the approach towards a programming task is affected. The next chapter also has some performance benchmarks to give some idea how well it works with very large projects.

Chapter IV

Benchmarks and Case Study

The purpose of the cache and the atlas metaphor was to enhance the performance and the interface of Aspect Browser (AB). This chapter evaluates the effectiveness of these features. The first part of this chapter presents benchmarks that demonstrate how well the atlas metaphor and the cache improved performance. The second part of this chapter describes a case study that evaluates how AB is used by a programmer.

IV.A Performance Benchmarks

As described in Chapter II, Aspect Browser version 1.1 (AB 1.1) was slow because its performance did not scale with large projects. It behaved like a batch program and only displayed a monolithic view. Chapter III introduced the cache and the atlas metaphor as the solutions to these problems. The effectiveness of these two solutions in respect to performance are measured by three benchmarks. Each benchmark is based on a possible interactive scenario that can expose strengths and weaknesses of our cache design. At the same time, these benchmarks can convey how AB might be expected to perform.

There are two things that these benchmarks do not assess. Whether the performance is satisfactory to the user depends upon the user's needs and expectations, so we do not attempt to address that issue here. Nor do we compare performance to AB 1.1. AB 1.1's major performance weakness was that it loaded and displayed the entire project in a single view. This resulted in very long start up times for the size of projects for which AB is intended. Given that AB loads panels on demand, start up is quite fast, and the only question remaining is whether the caching we designed provides adequate interactive performance once the tool is started.

The configuration of the computer used in these benchmarks is an 800 MHz Pentium III processor with 512 MB of RAM. AB was operated in the Windows 2000 environment with JDK

1.3. The time was measured with measurement code introduced into certain execution points of Nebulous.

IV.A.1 Switching Panels

Because AB loads panels on demand, switching between panels can be a more costly operation than in AB 1.1. The cache was designed to reduce the amount of time for switching views. We designed a benchmark that measured and compared the times of switching between views, with and without use of the cache. Although we did not turn off the cache during the benchmark, we assumed that displaying an uncached view was equivalent to displaying a view with no cache at all, because any overhead incurred by the cache is insignificant compared to the amount of time it takes to generate a view. We also wanted the benchmark to reflect a possible user scenario. Three directory panels were created, each containing a copy of the Nebulous source code. The Nebulous source code has 21,000 lines of code. Thirty aspect patterns were then created. Our cache was designed to remember the highlights and we expected that the matches would become the bottleneck while displaying an uncached directory panel.

Each time a different directory panel was selected, the time to display it was recorded. We first viewed all three directory panels one at a time. The first pass resulted in all compulsory cache misses because the directory panel has to be loaded into the cache for the first time. Then we selected all the views again for the second time, resulting in all cache hits. The times from each pass was collected and averaged. Both sets of averaged times are shown in Table IV.A.1.

Benchmark Test	Time
Changing the View Benchmark, Uncached	5.7
Changing the View Benchmark, Cached	0.9

Table IV.1: Cached versus uncached times of switching between views.

The cache itself does not incur any significant overhead because the cache look-up is fast compared to the overall time of generating the view of a directory panel. Therefore, AB only has to gain from the cache with respect to changing views. However, the overall performance of view switching really depends on the frequency of cache hits and cache misses.

As shown in this benchmark, a cache entry can be displayed six times faster than an uncached entry. However, the speed-up is dependent on many factors. We assumed the programmer would repeatedly view a small set of directory panels. The performance differences between cached and uncached views will grow as the list of aspect patterns grows because loading an uncached view must match the entire aspect list to create the display, while the cached view already has performed those matches.

IV.A.2 Traversing and Scanning Large Projects

Searching for the next match in other directory panels with the traversal feature may incur many cache misses in large projects because the cache can only contain a small number of directory panels at a time. In this case, AB will begin to behave as if it did not have a cache. We designed a benchmark that had tested AB's operational performance to an extreme level, forcing it to scan the entire project without the benefits of the cache. Each directory panel is loaded once and then discarded. We also compared its performance with `grep`. The source code in this benchmark was the Linux kernal version 2.4.17. The kernal code has over 4 million lines of code. 9000 files are stored in over 500 directories. The source is about 125 MB. A special string pattern was inserted into the kernal code at two spots, the first and last files as displayed in AB. We then had AB traverse the entire project in search of the two strings, searching for the pattern in every file. Then we tested `grep` under the Cygwin environment to search for that string pattern. Table IV.A.2 shows the amount of time to complete the task.

Benchmark Test	Time
Traversal Benchmark	90.5
`grep` Benchmark	14

Table IV.2: Total times to scan the entire Linux kernal 2.4.17.

The `grep` program scanned the entire Linux kernal six times faster than AB. We were expecting AB to perform much worse, something on the order of several minutes and at least two orders worse than `grep`. AB was not optimized to perform operations like this and the Java virtual machine and libraries incurred a performance overhead. We have demonstrated that AB can navigate within large projects.

The scenario represented in this benchmark is unrealistic because the chance of having the only two string matches at the extreme ends of the project is improbable. However, this unrealistic scenario is based on a more probable scenario where the traversal feature has to find the next match in the rest of the directory panels. Such a scenario may include examination and modification of every appearance of an aspect pattern. The net cost of examining over four million lines of code was about 90 seconds.

IV.A.3 Matching Aspect Patterns in the Cache

As explained in the Chapter III, the cache speeds up the switching time between directory panels, but the entire cache content, has to be updated every time the programmer manipulates the aspect patterns. We created a benchmark that compared the length of time of

adding a new aspect pattern to a single cache entry versus three cache entries. We first created one directory panel of the Nebulous source code. We recorded the length of time required to add a simple string pattern. We then restarted AB again and generated a total of three directory panels, each of them having a copy of the Nebulous source code. All of three panels were loaded into the cache before adding the same aspect pattern. Table IV.A.3 shows the two different lengths of time of adding a simple aspect pattern.

Benchmark Test	Time
Matching Aspect Pattern Benchmark, 1 cache entry	0.2
Matching Aspect Pattern Benchmark, 3 cache entries	0.6

Table IV.3: Total times to match and add new aspect patterns with one and three cache entries.

As expected, AB required at least three times the amount of time to match an aspect pattern with three cache entries compared to one cache entry. If Nebulous had no limit to its cache size and it had to load the entire project into memory, like AB 1.1 did, then AB would match the pattern with the entire project. With a limited number of cache entries, the amount of time required to match a large project with the pattern would be reduced because only a confined portion of the project would be matched by AB. If the directory panels are very large, the programmer can reduce the size of the cache to reduce the matching times.

IV.A.4 Conclusion

The cache and the atlas metaphor scales the performance of AB and empowers AB to work with large projects. The benchmarks had demonstrated that AB can handle very large software projects by loading only a portion of it into memory. AB can match aspect patterns across an entire project. The cache speeds up the time for switching between repeatedly viewed directory panels. Because of the cache's limited size, the cache also reduces the matching time. AB now has the potential to assist programmers to perform crosscutting changes with large software projects.

IV.B Case Study

This section describes a case study that was performed with AB and the atlas metaphor. The focus of this case study is almost the same as in the case study described in Chapter II, but this case study is exploratory in nature. We wanted to determine if the atlas metaphor and the new consolidated interface can assist the subject in evolving the code. As in the previous case study, the behavior and actions of the programmer were observed. Through our observations, we

can assess the weaknesses and strengths of AB and determine what type of improvements can be made to AB.

IV.B.1 Setup of the Case Study

This exploratory study was open ended and gave the voluntary participant, or subject, some choices. The subject chose the source code and the programming task, but we wanted the source to be at least 100,000 lines of code. The subject had the right to end the experiment at any time.

The main subject for this case study was a graduate student from the UCSD Software Evolution Lab. This subject decided to use AB on his software project, StarTool. StarTool is a software visualization tool that displays diagrams of the hierarchical classification of the uses of data structures and variables[GCBJLC98]. These diagrams assist the programmer in planning and making code modifications. The subject wanted to retarget StarTool for the analysis of C++ code, but StarTool needs an intermediate language (IL) representation of C++ code. The Edison Design Group (EDG) C++ front end was chosen to generate the required IL, which happens to be an Abstract Syntax Tree (AST). The subject had to modify and reverse engineer this C++ front end to make it work with StarTool. The total size of the project, including StarTool and the EDG C++ front end, is over 250,000 lines of code.

Before the case study, the subject had written some code to create his own IL because the original AST from the compiler was large and had extraneous information. The subject was interested in retrieving the source mapping from the IL, information that correlates each IL node to a specific line number and column number in the source code. The source mapping in the IL nodes were not stored in the same format and certain IL nodes did not have a source mapping. His existing implementation extracted most of the source mapping from the AST, but he knew there were a few obscure cases where it failed to retrieve the source mapping. He wanted to use AB to find those specific cases and fix it.

The subject had another programming task. He wanted to figure out how the EDG front end generates the unique names for the symbol table and how the front end controls the scope of those names. At the time of the experiment, StarTool for C++ could not make a distinction between different declarations with the same identifier. For example, if `foo` was a variable that was declared in three different files in three different scopes, StarTool would treat the variable `foo` as the exact same variable in all three different scopes. This problem arose because the EDG compiler created a separate AST for each file. No mechanism existed that could tell StarTool which identifier corresponds to which exact declaration. With AB, the subject wanted to figure out how the symbol table worked and how the EDG C++ front end keeps the identifier names

persistent. Then he wanted to incorporate a mechanism in StarTool to retain this persistent information between different AST's.

As in the previous case study, the observational method of this case study was constructive interaction [Myu86]. Two people collaborated on the task. In this case, one person is the subject and the other person is the tool author. As in Chapter II, the tool author's role is to provide support in using AB. The support includes a tutorial. However, the tool author should not enforce any standards in using the tool because that would violate the goals of our exploratory case study.

The case study was conducted in the subject's lab. A video camera with microphones was used to record the sessions of the subject using the tool. Each participant had a microphone. The subject had the right to end the recording at any time. The video camera recorded the contents on the computer monitor and any hand gestures made in front of the computer monitor.

We used two different computers in this case study because of privacy issues in the lab. A Solaris Ultra 10 computer with 512 MB of RAM was used in the first part of the case study. During the latter half of the case study, the subject's computer was used. The subject's computer was a Pentium II 300 MHz computer and it had 512 MB of RAM. The computer's operating system was RedHat Linux 6.2.

After the bulk of the case case study was over, the subject was interviewed. The subject could discuss anything about the case study. The tool author asked him questions about certain actions or evens that occurred during the case study.

IV.B.2 State of the Tool

At the time of the study, wild-card matching was not implemented for convenient file selection. Only a simple file extension can be used to quickly grab many files.

IV.B.3 General Observations

The experiment spanned across three days. Each day, the subject spent two hours using Aspect Browser. Because of the subject's time constraints, the project ended before he was able to implement all of the changes. He made about twenty aspect patterns during the entire experiment. He got as far as exploring the code and uncovering certain things he needed to know about the C++ front end. We videotaped only the hours he was actively using AB, less than six hours. The subject did spend some time alone, outside from our observation, studying the reference manual of the front end.

During the beginning of the case study, the tool author was supposed to give a tutorial about how to use the tool. However, the subject was familiar with AB 1.1. The tool author

briefly went through the new features and the subject quickly figured out how to use most of the other features with little help. The subject knew how to use AB to explore the EDG C++ front end code. The tool author only intervened when the subject requested help in using the more obscure and complex features of AB.

The subject loaded the project files into AB with the **Quick Load** feature. Nebulous created three directory panels. Because the EDG C++ front end was inside one directory, one large directory panel contained the entire EDG C++ front end. The other two directory panels represented extraneous subdirectories. These subdirectories contained patch files and backup files. The subject pointed out that he was only interested in the directory that contained the EDG C++ front end.

Layout of the Windows on the Desktop

The subject had a preference in placing his windows on the desktop. In X Windows, he took advantage of the multiple desktops and placed the `Vi` editor on one desktop. The subject used `Vi` to edit the code. He then started Aspect Browser in a different desktop. At first, he did not use Aspect Emacs. However, he became frustrated in using the magnifying glass window because the arrow keys did not scroll the text view, so he began using Aspect Emacs as his primary text viewer. He placed Aspect Emacs on top of Nebulous, covering everything but the traversal feature. He wanted to examine the code at each highlight as he traversed through the highlights.

Exploring the Code

The subject wanted to use AB to help him explore the code and track down specific concerns that spanned many files. Before starting any of the tasks, he consulted the reference manual. The reference manual had some information about the data structures and the functions related to his task. When he first started working on either task, he would enter the names of variables or functions mentioned in the reference manual into AB and examine those highlights. The highlights were usually relevant to his concern because the naming convention reduced the number of irrelevant matches. The naming convention in the EDG C++ front end separated each word component of a name with underscores. The names usually described the action of a function or the properties associated with a variable. Abbreviations were sometimes used to indicate variable types and pointers. An example of a variable in the EDG C++ front end is `namespace_ptr`, a pointer to a name space.

The subject decided to work on the source mapping problem first. He made several aspect patterns in the course of this task. Most of the aspect patterns represented the declara-

tions of the different types of IL nodes inside the AST. The first aspect pattern he entered was `a_correspondence_source`, which is a struct type that has the column and line number of where an IL node appears in the source. He wanted to check if his existing code properly identified the type of the IL node before extracting the source-mapping, because not every type of IL node had the source mapping. While checking his code, he discovered that he had made a mistake because two similar types of IL nodes in the AST had similar names. One IL node type had the `a_correspondence_source` and the other didn't. He confused the names between the two IL node types and extracted the source mapping from the wrong one. He added a comment in the code to remind himself to make the changes later.

He discovered later that a whole class of IL nodes had no `a_correspondence_source` and they all had the prefix `a_scope` added to their names. These nodes represented the scopes that exist in the C++ language such as function scopes and class scopes. These scope nodes were composed of other IL node, sometimes including other scope nodes. Eventually, all scope nodes broke down into atomic IL nodes that contained a source mapping. After he made this discovery, he modified one of his functions that retrieved the source mapping information, `get_source_position()`. He modified this function to recursively call itself if the node is a scope node. The function continued to recurse each IL node that is part of that scope node until all of the source mapping had been retrieved.

After he made the function recursively call itself to handle scope nodes, he decided to stop further investigation of the source mapping problem because he said it would be repetitive to make all the appropriate changes during the case study. At the end of this task, he had several aspect patterns representing the various types of IL nodes. He had created several workspaces and used the workspaces as bins to categorize the aspect patterns. For example, all of the scope IL node patterns went into a scope-node workspace.

The second task of exploring the symbol table in the EDG C++ compiler was conducted intermittently when the subject was bored with the source mapping task. Starting this task was not as straightforward as the other task because he had to figure out how the symbol table worked. He discovered that the symbol table was a hashtable and the hashtable was indexed by the identifier of the symbol. Each hash entry contained a list of different symbols that have the same identifier. Each symbol in this list was the head of its own linked list that consisted of every appearance of that symbol in the code. The subject wanted to create a function that would determine if two appearances of the same identifier will map back to the same symbol. He was hoping that a pointer in the corresponding IL node would point back to the corresponding node in the linked list of appearances. Then he would traverse up this linked list of appearances until he reached the head that contained the symbol. If the identifiers had the same matching

head, it was the same symbol. However, he could not find a simple pointer in the IL nodes that led back to a node in the linked list of appearances.

After an hour of further exploration of the code, the subject found the function that matched an identifier to a symbol. At this point in time, the subject did not have the time to confirm his discoveries and to finish the code modifications. He also left two major issues unresolved. First, he did not resolve the issue when the identifier was specified with `extern`. Second, he never built the mechanism that creates a persistent symbol table for StarTool.

IV.C Detailed Observations and Analysis

Difficulties of Analyzing the Case Study

The subject had previous experience building and using software tools because he worked with StarTool. He also read some papers about AB 1.1. His prior knowledge could have affected our data. For example, he could have had preconceptions about using AB with his code exploration. He could have intuitively figured out how to use certain features in AB which an inexperienced tool user may have problems with. As a consequence of these possibilities, we potentially missed several interface problems.

The amount of time we had to conduct the case study was limited. The subject did not have time to finish code exploration and make the changes. He performed few code modifications to StarTool or to the EDG C++ front end. We only had the opportunity to observe him explore the code and define his own directory panels. We did not get the opportunity to observe him finish his tasks.

Aspect Index

Observations. The subject had some trouble learning how to use the **Aspect Index**. In the beginning, he did not know what the buttons did and he had a hard time remembering what the buttons did. He sometimes made the mistake of adding an aspect pattern to the **Aspect Index** instead of adding a workspace because he pushed the wrong button.

After examining some code and creating several aspect patterns, the subject categorized his aspect patterns with some workspaces. However, moving or copying the aspect patterns to the workspaces was cumbersome because these actions required many mouse operations. First, he retyped the entire regular expression into the **Pattern Text Field**. Then he selected the workspace before adding the copy to that workspace. If he was moving the aspect patterns to a new workspace, he then deleted the old copies. After the subject had copied several aspect

patterns by retyping them, the tool author had told him that he could save time by selecting the aspect pattern and the pattern would then automatically appear in the **Pattern Text Field**.

The subject sometimes wanted to edit the regular expression of the aspect pattern. Nebulous did not have a feature that directly edits an existing regular expression. The subject eventually figured out that he can just create new variations of the aspect pattern until he got the correct pattern. He then deleted all the incorrect aspect patterns. The subject typed each variation by hand until the tool author told him can copy the regular expression into the **Pattern Text Filed** by selecting the aspect pattern before editing the regular expression.

The hidden aspect folders (HAF's) confused the subject many times. As mentioned in Chapter III, HAF's are created when three or more neighboring aspect patterns are hidden. The hidden aspect patterns are then moved into these HAF's. When the subject hid three or more neighboring aspect patterns, the tree changed shape because the hidden aspect patterns are moved into a HAF. The changing of the tree's shape disoriented the subject. These HAF's also made it difficult to find an aspect pattern because these folders were automatically closed after their creation. Sometimes the subject had to open every HAF in the **Aspect Index** to find a specific aspect pattern, even when the aspect patterns were categorized and placed into the proper workspace.

The subject complained about the default colors Nebulous chose for the aspect patterns. Sometimes Nebulous would pick colors that are alike or too dark. He said that he preferred the lighter colors over the darker colors such as light green and light blue because those lighter colors are easier to tell apart. Darker colors makes the highlighted text in Aspect Emacs and in the magnifying glass harder to read. Very dark colors are hard to tell apart because they would all look like the color black.

Analysis. The buttons on the tool bar that manipulated the **Aspect Index** had several user interface problems. Some of the buttons were not self-explanatory because there is no universally understood symbol to represent their actions. For example, we could not find an intuitive symbol to represent the addition of an aspect pattern. We also tried to associate these buttons with the **Aspect Index** by placing them right above the **Aspect Index**, but the subject did not realize that all of those buttons manipulated the **Aspect Index**. Later, the subject made the comment while waving the mouse from the tool bar to the top part of the **Aspect Index**, "You know, you should just move this closer to here". He wanted some of the buttons to be in the same panel of the **Aspect Index**.

Many of the problems the subject had with the **Aspect Index** may be solvable with a drag-and-drop interface. The subject wanted to reorganize his folders by copying and moving

aspect patterns between workspaces. As mentioned in the observations, the process of moving or copying an aspect pattern required several steps and is prone to error. The subject also had a natural tendency to interact with the **Aspect Index** with drag-and-drop motions. For example, he made the mistake of placing the aspect pattern in the wrong workspace. He tried to correct the error by clicking and dragging the aspect pattern towards the correct workspace, but Nebulous did not support drag and drop. He then said, "I didn't mean to create the aspect in that workspace here. It would be nice to drag it." He also associated the drag-and-drop behavior with multiple selection. Throughout the case study, he repeatedly tried to select multiple aspect patterns and perform an operation on them. However, Nebulous did not support multiple selections.

The subject also wanted a feature to edit the actual aspect pattern instead of creating new variations of the aspect pattern. He said that feature would be more intuitive than creating new aspect patterns and then deleting the unwanted aspect patterns.

The hidden aspect folder (HAF) had confused the subject. The subject was frustrated with the automatic creation of the HAF's. He complained, "...the thing I didn't like again was the grouping of aspects in triplets, like I click hide and it collapsed the aspects into a node I did not create." Later, he mentioned, "You should not waste your time trying to second guess the user with grouping aspects by three. Or maybe if I had hundreds of aspects, like again, I feel like I am the type of person who want control over the grouping." Nebulous should perhaps make the HAF optional.

Because the subject found himself activating only one aspect pattern at a time, he requested a feature that can automatically activate the selected aspect pattern and hide all others.

The subject's overall response about the **Aspect Index** was positive because he was using it to organize his discoveries and his concerns. After creating a half a dozen aspect patterns and a few workspaces, he said, "The thing I am finding the tool is helpful for is categorizing different kinds of identifiers related to some kind of aspect. Like, when I use the term aspect, I like to think of these folders as the real aspects because being an aspect is like a coarser grain concept." Further, he said that categorizing the patterns and activating certain highlights helped him search for specific concerns. This process helped him iteratively refine his understanding of the software.

Traversing and Examining Aspect Patterns

Observations. The subject usually followed a process of examining each highlight. As mentioned before, he moved Aspect Emacs on top of the Nebulous window, but he left the traversal feature exposed. He then activated one aspect pattern and hid the rest of them. He visited the

highlights and he examined the code at each highlight.

The subject had a few problems with the traversal feature. After he selected a different aspect pattern to traverse, he expected the first place he would visit was the first highlight in the code instead of the nearest highlight from the present position of the red cursor. He got around this problem by scrolling to the furthest left file panel and immediately clicked on the first visible highlight. If no highlights were visible, he sometimes scrolled down the panel in search of a highlight. Other times, he would click on top of the first file panel and then let the traversal feature find the first occurrence.

The subject did not like the traversal feature when it started to scan the other directory panels after visiting the last highlight. He was not interested in the other two subdirectories that were accidentally created with **Quick Load**. He would accidentally visit the next match because he did not see the visual cue of reaching the last highlight on the display, a blinking red and green cursor. Nebulous would pause for about a minute because the program had to swap the view of the EDG C++ front end with one of the two other directory panels. After he realized his mistake, he would switch back to the EDG C++ front end and he would have to wait about another minute. The switching of the views took a long time because the computers he was using for this case study could barely contain the entire view of the EDG C++ front end in memory.

The subject briefly used the file panels for very specific purposes. After adding a new aspect pattern, he examined the files panels to see how many matches he had and how dispersed these matches were. He would use the file panels to navigate repeatedly and quickly between two places and then examine the code at both spots. Other times, he would just double-click on a highlight and examine the code at that spot.

Analysis. In the first task of extracting the source mapping, the aspect patterns became a list of tasks. The subject created many aspect patterns that represented the different types of IL nodes. He then traversed each of those apsect patterns one at a time because he was searching for the declarations of each of those IL nodes. After finding the declaration of the IL node, he would create a specific aspect pattern that would only highlight that declaration. When the subject started to modify StarTool, this collection of the declarations of the IL nodes became a list of tasks because each IL node represented a specific modification to StarTool.

When the subject was trying to solve the symbol table problem, the subject retained many of the same habits from the previous task such as traversing with one aspect pattern and creating aspect patterns that highlighted only the declaration of a variable. However, the symbol table problem was more complex because the subject needed to examine the relationships between the aspect patterns before he could understand how the symbol table worked. The activation of

multiple aspect patterns was more common, but we do not know when he had forgotten to turn the other patterns off or when he intentionally wanted to see other aspect patterns.

The user expected the traversal feature to behave differently. He expected the traversal to visit the first highlight after selecting a different aspect pattern. Instead, the traversal feature visited the next highlight near the present location of the red cursor. Later, he requested a button that would send him to the first highlight on display. He also expected the traversal feature to wrap back to the first highlight after reaching the last highlight on the display. Instead, the traversal feature scanned other directory panels for the next match.

The subject had a few problems with the controls of the traversal feature. The subject said that the pull-down menu was not intuitive. He thought it was a menu that selected the type of highlights to display instead of a menu that selected the traversal mode. He did not mention if those other traversal modes would have been useful to him, but he never complained about not having those other traversal modes. The second user interface problem was the bookmark feature. He asked the tool author what the purpose of that button was towards the end of the case study.

The subject preferred to use Aspect Emacs to examine the code because it had the highlights and it had shortcuts for navigation. The magnifying lens was not as useful because it did not respond to any key commands for navigating around the text. The subject still used `Vi` to edit the code.

Atlas Feature

Observations. The subject did not use the atlas feature for most of the case study. Two of the three directory panels that were created with **Quick Load** at the beginning of the case study were created by mistake. The subject had no interest in their contents and the accidental switching of the directory panels with the traversal feature was a hindrance.

Towards the end of the project, the subject had enough understanding of the code to group the files by concerns. He created three directory panels. One directory panel was for files related to the symbol table. The second directory panel was for files that had concerns about the source mapping. The last directory panel was suppose to contain the rest of the files that did not belong to the other two concerns, but AB had no quick and convenient method of grabbing the files that excluded the other two concerns. The subject decided to include the entire EDG C++ front end, including the files from the other two directory panels, in the third directory panel.

The dialog box that defined the directory panels, the **Add and Remove Files** dialog box, was not intuitive and the subject got lost. The tool author had to guide the subject every step of the way in customizing the groups of code. Wild-card matching for file selection was

not implemented yet. Fortunately, the files he wanted to create the directory panels with were contiguous in alphabetical order and the dialog box could easily choose a contiguous block of files in alphabetical order.

After the three legitimate directory panels were created, the subject turned on all of the aspect patterns. He examined all the directory panels and was surprised that some of the highlights in the symbol table directory panel overlapped in the source mapping directory panel and vice versa. He never mentioned if this overlap was important.

The subject decided to end the case study at this point. We never had a chance to observe how he would have used the atlas feature while exploring and modifying the code.

Analysis. At the beginning of the case study, the **Quick Load** feature created some directory panels for the subject, but the directory panels did not fit the needs of the subject. **Quick Load** was designed on the premise that most programmers would like to see their projects automatically partitioned by the directory structure of the source. Because the EDG C++ front end code only had one directory, a single monolithic view was created by **Quick Load**. This single view did not help the subject because it rendered the the caching and atlas metaphor ineffective. In addition to the single monolithic view, two extraneous subdirectories were added to the project.

Later, the subject used the atlas feature as a tool to organize the EDG C++ front end. The code was decomposed according to the subject's primary concerns. These partitions could have assisted the subject in modifying the code if he had continued with the case study. Separating the files would have improved the performance of AB and the absence of extraneous files had the potential of enhancing his visual reasoning because he could then focus on the files that had direct relevance to his concerns.

The subject used the **Add and Remove Files** dialog box to create his three custom views. The **Add and Remove Files** dialog box was not intuitive. The subject struggled with the dialog box because the text entry fields, the list of files, and the list of directory panels cluttered the interface. The subject did not know which text entry to fill out in which order. He also did not know what the buttons on the dialog box do. The tool author had to give the subject a step-by-step tour of the dialog box.

During the case study, the subject had mentioned an idea about how aspect patterns can manipulate the view of the directory panels. If an aspect pattern is selected, Nebulous should produce a display that shows only the file panels and directory panels with matches. However, this view can be achieved with the version of AB used in this case study. The user has to select the viewing mode that recursively shows all of the directory panels. Then the file panels without matches should be folded. This view would be almost the same view the subject

wanted, except the directory panels without matches would still appear. Achieving this view is convoluted because it requires many steps to achieve. A menu option or a button command can make this view more easily achievable.

Analysis of the Atlas and Map Metaphor

Assessing the effectiveness of the map metaphor from the subject's behavior and actions was not straightforward. In the previous case study, the effectiveness of the application of the map metaphor was measured by the subject's behavior and actions. If the subject explained things with map-like terms or symbolically associated colors with code, the map metaphor was considered to be applied by the subject. In this case study, those occurrences were few if non-existent. We focused our analysis on how the subject used AB for code exploration.

Although the subject did not approach his problems like a person using a map, he did use many of the fundamental features to assist him in his exploration of the code. The subject kept his interactions with the tool simple enough to just get his task done. First, the subject examined one or few aspect patterns at a time. Second, he only traversed the code with one aspect pattern at a time. He was not interested in traversing to collisions. Third, for the most of the time, the subject did not examine the file panels for any significant amounts of time. We do not know if this is because he did not find any useful information on the file panels or because he only needed to glance at the files panels before understanding the information. If he found the file panels useful, the subject might only need to glance at the file panels with a few active aspect patterns. He did use the file panels several times to quickly navigate to a highlight. These observations demonstrate that the atlas metaphor had helped him, whether or not the subject was actively participating in the map metaphor.

The subject's background biased his approach. He was the author of StarTool, another visualization tool. His expertise with software tools and his advanced knowledge in software design was at least as strong an influence as the map metaphor. He was only using AB when it was appropriate and convenient.

However, during the interview, the subject did mention that his tasks could have been completed with just **grep**, but AB made it easier and more convenient for him to explore the code. He did not point out which features of AB made it easier for him, but we assumed its combination of the file panels, the highlights, the **Aspect Index**, and the traversal feature.

The atlas metaphor and the map metaphor are interwoven into each other in the interface and this could make the assessment of the atlas metaphor hard. Before the case study, the original observations of the atlas metaphor were confined to certain features of AB such as traversing across directory panels in search of the next match, defining directory panels, and

choosing which directory panel to display.

The subject said that the definable views are very important. It helped him refine his concerns and put the concerns into perspective. This process of refining his concerns is equivalent along the lines how Rigi or Reflexion models help software engineers decompose software into specific modules [MTO+92] [MN97].

The subject knew how to use the **Directory Tab Panel** and the directory tabs without any help. The labels on the buttons helped him identify which button correspond to which directory panel. However, the tool author did not know what the shrunken pictures represented. He thought they were static icons until he changed the views several more times and turned some of the aspect patterns on and off. He eventually realized that the thumbnails changed as he visited the different directory panels. However, he mentioned that he wanted a feature that can globally match the aspect patterns and indicate which directory panels have those aspect patterns without visiting them because AB only updates the thumbnails during visitation.

We did not get comprehensive data about the effectiveness of combining the traversal feature with the atlas metaphor. The subject only used it by accident when his three directory panels included a monolithic view of the EDG C++ front end and the two extraneous directories. He complained about the amount of time he had to wait to switch between views because the computer could barely fit the entire view of the EDG C++ front end into memory. However, we expected most software to be decomposed into much smaller parts and the switching time should be less. Because he never really used this traversal feature with the actual intention of looking for other matches in other directory panels, we do not have any evidence that can help us determine the effectiveness of this feature.

Overall Analysis

The subject's reaction towards AB was positive. He had some insights about the program that surprised us. AB was originally designed to help programmers find concerns and then make the correct modifications without disorientation. AB did assist the subject in finding concerns, but it also had an unanticipated side effect as explained by the subject during the interview:

> "I really mean this, the tool is fun to use. It is interesting to organize some thoughts. With command line grep, its difficult to go back to a particular pattern you have chosen previously as being useful. This tool makes it obviously really easy to go back to a previous one. So right, I think this tool added value in the sense, iteratively refined my understanding of certain things about EDG and saved that process in the project file that I can actually see which aspects were related to the symbol table and see the ones I created later."

As mentioned before, AB is a tool that helped iteratively refine the programmer's understanding of the code. The aspect patterns in the **Aspect Index** and the directory panels gave the subject the power to group concerns together in multiple dimensions. First, the **Aspect Index** allowed him to keep which concerns he is interested in and generate hierarchical relationships with groups of aspect patterns. Second, the subject defined his own directory panels based on the partition of concerns between the symbol table and source mapping. The subject claimed that being able to break down the views by their concerns was helpful. As mentioned in the observations, the subject discovered that some aspect patterns existed in both the symbol table directory panel and the source mapping directory panel.

The subject had also said that AB is a software tool that can record a programmer's thought process and remember the programmer's discoveries. He said that if he did not have AB, he would have approached the problem quite differently. As a habit, he would have made a copy of the code and inserted his comments with his initials into the code. Later, he would `grep` the code for his comments. He mentioned that there are two problems with this process. First, he would have to go through the repetitive process of commenting the same concern that is dispersed across many files. Second, if the program changed enough, the comments would become out of date and the subject would have to redo his comments again. This process would be tedious. AB is a software tool that provides a mechanism that associates an annotation with a dispersed concern, independent of the state of the source. The programmer can now review updated code with his old comments. As the software evolves and the contexts of the concerns change, the programmer can quickly update his annotations. If the numbers or locations of the concerns change, AB can automatically match the aspect patterns again and locate the new appearances of those concerns.

Chapter V

Conclusion

Aspect Browser (AB) helps programmers make crosscutting changes. AB 1.1 was designed around the map metaphor, which displays code through map-like features. The first case study with AB had demonstrated that the map metaphor had assisted the programmer in making crosscutting changes across a large software system. However, the case study had also revealed that the performance of AB did not scale with large programs. The user interface had some limitations in navigating and viewing different files. The atlas metaphor was then incorporated into AB in solving the performance problems and user interface problems.

We added three main features to AB. The interface of AB was scaled with the directory tabs. These buttons choose the directory panel to display rather than attempting to display them all. Each button is labeled with the name of its directory name and has a thumbnail of its file panels.

The second feature is the cache. The cache stores a limited number of previously visited directory panels and reduces the amount of time to display a cached entry when the programmer demands to view it. Switching to a cached view is quicker because AB does not have to match any aspect patterns before displaying that cached view; the cached directory panels retain their highlights and updates. In contrast, AB must perform the lengthy task of matching all of the aspect patterns before displaying an uncached directory panel. Each time the programmer adds or manipulates an aspect pattern, each directory panel in the cache is updated. The larger the cache, the more time AB needs to update the entire cache. The programmer can adjust this performance tradeoff by adjusting the size of the cache.

The third feature is the unification of the user interface. The browser functionalities in Aspect Emacs were moved to Nebulous. Nebulous can now manage aspect patterns and workspaces with the **Aspect Index**. It has a tool bar that contains the traversal features and

the zoom features.

Although the data from the case study in Chapter IV is limited, the subject had demonstrated that it can be an effective tool for making crosscutting changes. The **Aspect Index** was used for organizing the subject's aspect patterns into groups. The source files that resided in a single directory were grouped by their concerns and then assigned to a specific directory panel. These actions show that this tool can be used to define other forms of decompositions. These new decompositions can be generated without modifying the actual code.

We conducted three performance benchmarks with AB and all three benchmarks showed promising performance results. The first benchmark showed how quickly AB can switch directory panels with or without caching. The cache did reduced the amount of time for changing the view. The second benchmark tested how quickly AB can read and match every file in the Linux kernal source. AB took about 90 seconds to scan the entire Linux source kernal. We expected the performance to be much worse because the matching code was unoptimized and the Java virtual machine added a performance overhead. The third benchmark demonstrated how capping the size of the cache improves the performance of matching aspect patterns. If a large project was entirely loaded into memory, AB would have to match the entire project for each aspect pattern. Because the amount of time required to match a pattern is linearly proportionate to the total size of the cache content, restraining the size of the cache has improved the performance of AB.

V.A Contributions

V.A.1 Designing the Atlas Metaphor for Crosscutting Changes

The atlas metaphor was designed for finding and manipulating crosscutting concerns. AB visually integrated two different views of the crosscutting concerns in the software with the **Directory Tab Panel** and the file panels. This visual integration provided a global view, saved space on the screen, and kept the overall interface consistent. The **Directory Tab Panel** was designed to show information at an expansive level because the position of the directory tabs and the labeling scheme on the directory tabs show the relationship between directory panels. The thumbnail of a directory tab keeps the expansive view consistent because it is picture of a directory panel. The traversal feature visits the directory panels in the same left-to-right order of the directory panels inside the **Directory Tab Panel**. The programmer can easily associate the expansive view with the actual directory panels on display.

V.A.2 Tool for Making Crosscutting Changes

The case study has shown that AB can assist in crosscutting changes. With AB, the programmer can record new concerns and visualize how those concerns crosscut the software. The traversal feature can give the programmer a tour of the project because it can display the code at each highlight. As the programmer understands more of the code, AB can be customized to reflect his understanding with definable views.

V.A.3 Unifying the Interface Between Multiple Software Tools

Unifying the interface from Aspect Emacs to Nebulous has made the tool easier to use. The programmer can find all the commands in one spot and does not have to switch focus between Emacs and Nebulous before he has access to a specific action. The removal of the dependencies between Nebulous and Aspect Emacs made it easier to retarget Nebulous with another editor.

The case study also showed that consolidating the interface to the tool bar and the menu made the commands more accessible. The subject habitually moved his mouse directly to the tool bar when he wanted to perform an action. When the subject wanted to explore the different actions he can perform, he can find the new commands easily on the tool bar and in the menus.

V.B Future Work

V.B.1 Large Scale Case Study

The case study that was presented in this paper did not fully evaluate AB. First, we wanted the case study to be conducted in industry because we wanted to observe how AB helps programmers in a complex and realistic environment that has external factors such as deadlines and budget constraints. The software project should have over one million lines of code and it should be separated into several modules. We wanted to observe how a programmer would maintain and modify an extremely large software system with AB.

V.B.2 Understanding Caching

The atlas metaphor attempted to address the performance issue with caching. The cache scaled AB's performance and restricted the amount of content that could be. The performance of the cache could be improved by observing how a programmer interacts and uses AB. From these observations, we can create better caching policies and maybe implement new caching features such as predicting and prefetching directory panels.

V.B.3 Global Scanning and Updating

AB would benefit from new features that globally scan and update the entire project. For example, the directory tabs should be updated in real time as the programmer manipulates the aspect patterns instead of only updating the directory tabs when the programmer visits them. The redundant line feature already performs this global scan, but only on demand. It scans each file one at a time to conserve memory. However, Nebulous is unresponsive during this lengthy scan. This pause is acceptable because we expect programmers to only occasionally invoke this feature. However, this pause would be inconvenient if it was associated with a frequent event. Any operation that requires global scanning or global updating can be processed in the background and the programmer can continue interacting with Nebulous. However, these types of features may compromise stability and correct operation because they may introduce unwanted race conditions unless carefully designed.

V.B.4 Extending the Atlas Metaphor

The atlas metaphor gives the programmer an interface to separate his views of the files while preserving a vestige of a global view through the directory tabs. Whether or not the programmer has to create his own separations or the separations are already offered by the directory structure or by documentation, the atlas metaphor only offers an interface that limits the view of the project. The ability to display multiple directory panels is limited in AB because it can only show the entire project recursively or just the parent and child directory panels. Nebulous can be enhanced by two more display features. First, Nebulous should have the option of displaying any combination of directory panels. This was omitted from the original atlas metaphor because separating and limiting the view of the files was a priority. Second, the programmer should be able to generate displays that only have the directory panels that contain only certain concerns.

The directory panels do not depict every property about a crosscutting concern. The highlights do not display direct relationships between files and directory panels. For example, a programmer does not know if the highlight is a function call or function declaration until the programmer has examined the code at those highlights. Further, a programmer has no direct way of annotating and recording the relationships between the highlights. AB needs a display that shows the actual relationships between the modules. Rigi and Relexion modeling are two software tools that let the programmer iteratively define boundaries between modules and the relationships between the modules [MTO+92],[MN97]. These relationships may include function invocation and class hierarchies. However, these tools are language specific and require

an intermediate representation of the code.

V.B.5 Visual Cues

Since AB 1.1, visual cues have been a problem. AB needs better visual cues to indicate when a lengthy task is done or is about to be performed. For example, the red cursor blinks green when it has found the last match in the directory panel. This indicator is often overlooked. If the programmer does not notice this blinking cursor and unintentionally scans for the next match, AB would start scanning the other directory panels. Instead, a dialog box should pop up and ask permission before AB continues the scan for the next match.

Progress meters are important for time-consuming tasks because they tell the programmer how long a task could take. Many of the lengthy tasks that AB can perform were originally designed to have a progress meter, but multi-threading issues have caused lock-ups. For now, the programmer must wait for the task to finish without a progress meter. For some of these time-consuming tasks, a dialog box does appear to tell the programmer that the task has been completed.

Bibliography

[DRO98] Jr. Dan R. Olsen. *Developing User Interfaces.* Morgan Kaufmann Publishers, Inc, San Francisco, CA, 1998.

[ESS92] Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner. Seesoft - a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, November 1992.

[GCBJLC98] William G. Griswold, Morison I. Chen, Robert W. BowDidge, and J. David Morgenthaler Jenny L. Cabaniss, Van B. Nguyen. Tool support for planning the restructuring of data abstractions in large systems. *IEEE Transactions on Software Engineering*, July 1998.

[Gri01] William G. Griswold. Coping with software changes using information transparency. In *Reflection 2001: The Third International Conference on Metalevel Architectures and Separation of Crosscutting Conterns*, Kyoto, Japan, September 2001.

[GYK01] William G. Griswold, Jimmy J. Yuan, and Yoshikiyo Kator. Exploiting the map metaphor in a tool for software evolution. In *Proceedings of the 2001 International Conference on Software Engineering*, Toronto, Canada, March 2001. IEEE.

[IBM00] Multi-dimensional seperation of concerns: Software engineering using hyperspaces, 2000. `http://www.research.ibm.com/hypserspace/`.

[JS98] Dean F. Jerding and John T. Stasko. The information mural: A technique for displaying and navigating large information spaces. *IEEE Transactions on Visualization And Computer Graphics*, 4(3):257–271, July–September 1998.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Budapest, Hungary, June 2001.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242, Finland, June 1997. Springer-Verlag.

[KS98] Eser Kandogan and Ben Schneiderman. Elastic windows: Design, implementation, and evaluation of multi-window operations. *Software-Practice and Experience*, 28(3):225–248, March 1998.

[Mac94] A.M. MacEachren. *Some Truth With Maps: A Primer On Symbolization & Design*. American Geographer, Washington D.C., 1994.

[MG90] A.M. MacEachren and J.H. Ganter. A pattern identification approact to cartographic visualization. *Cartographica*, 27(2):64–81, 1990.

[MN97] Gail C. Murphy and David Notkin. Reengineering with reflexion models: A cast study. *Computer*, 30(8):29–36, 1997.

[MTO+92] H.A. Muller, S.R. Tilley, M.A. Orgun, B.D. Corrie, and N.H. Madhavji. A reverse engineering environemnt based on spatial and visual interconnetion models. *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments (SIGSOFT 1992), ACM Software Engineering Notes*, 17(5):88–99, December 1992.

[Myu86] N. Myuake. Constructive interaction and the iterative processs of understanding. *Cognitive Science*, 10(2):151–177, 1986.

[OT99] Harold Ossher and Peri Tarr. Multi-dimension seperation of concerns in hyperspace. Technical Report 21452, IBM Research Division, April 1999.

[Par72] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[Pet] Michael P. Peterson. Between reality and abstraction: Non-temporal applications of cartographic animation. `http://maps.unomaha.edu/AnimArt/article.html`.

[Tuf83] Edward R. Tufte. *The Visual Display Of Quantitative Information*. Graphics Press, Ceshire, Connecticut, 1983.

[VV00] John Viega and Jeffrey Voas. Can aspect-oriented progrmaming lead to more reliabale software? *IEEE Software*, pages 19–21, November/December 2000.

[Yua00] Jimmy J. Yuan. Using the map metaphor to assist cross-cutting softare changes. Master's thesis, Department of Computer Science and Engineering, University of California, San Diego, 2000.