UNIVERSITY OF CALIFORNIA,
IRVINE

Unifying Artifacts and Activities in a Visual Tool for Distributed Software Teams

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Information and Computer Science

by

Jon Edward Froehlich

Thesis Committee:
Professor J. Paul Dourish, Chair
Professor Andre van der Hoek
Professor David Redmiles

2004

The thesis of Jon Edward Froehlich is approved:

_____

_____

_____
Committee Chair

University of California, Irvine
2004

DEDICATION

To

my grandfather, Andrew Froehlich,

in recognition of his unwavering support and wisdom

And now I cannot remember how I would have had it.
It is not a conduit (confluence?) but a place.
The place, of movement and an order.
The place of old order.
But the tail end of the movement is new.
Driving us to say what we are thinking.
It is so much like a beach after all, where you stand
and think of going no further.
And it is good when you get to no further.
It is like a reason that picks you up and
places you where you always wanted to be.
This far, it is fair to be crossing, to have crossed.
Then there is no promise in the other.
Here it is. Steel and air, a mottled presence,
small panacea
and lucky for us.
And then it got very cool.

-John Ashbery, 1988
Inscription on the Siah Armajani's Irene
Hixon Whitney Bridge in Minneapolis, Minnesota

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I am forever grateful to my committee chair and advisor, Professor Paul Dourish. He has presented me with opportunities not normally afforded a Master's student and for that, I am thankful. He has served his role as advisor in every sense of the word, both personally and academically. His students are well aware of how fortunate we are. Paul, thank you. I will miss our interactions.

I would like to extend my appreciation to the rest of my committee, David Redmiles and Andre van der Hoek. Thank you for your kind words and confidence.

I would also like to thank my fellow graduate students for their support, specifically those in the dourish.com research group (who are probably just glad that they won't have to sit through anymore Augur practice presentations): Danyel Fisher for his feedback, advice, and creativity; Jack Muramatsu for his constructive comments, interface confusions, and conversation; and May Tan for her work on the JFreeChart visualizations. Thanks also to Scott Hendrickson for reviewing the table, figure, and section references.

I am indebted to my girlfriend, Cassandra Hearn, if for no other reason than understanding how much a windbreaker and a song might mean to me. She has demonstrated incredible patience and understanding in the final weeks of writing this thesis (and the final two years of graduate school for that matter). She may also be nearly as stubborn as me, which is probably more befitting a lawyer than a computer scientist (I concede).

As I continue onward through this crazy academic life, one constant is sure to remain, that of the love, support, and encouragement I receive from my family. I am truly blessed with wonderful and caring parents (Happy 31st Anniversary—June 9th, 2004) who I can always look to for comfort and guidance. I would also like to recognize my sister and brother-in-law for their Faith and love and my grandparents, for providing a strong foundation for it all.

In closing, I would like to thank a group of writers who inspire me: Steven Paul Smith, Jack London, Ken Kesey, Rivers Cuomo, Herman Wouk, Peter Yorn, Steven Patrick Morrissey, Jon Favreau, Thom Yorke, Justin Frankel, Gary Gulman, Edward Louis Seversen III, Chris Martin, Myra Ellen Amos, Martin Eden, Robert Smith, Alanis Morissette, Dane Cook, Ben Folds, Stephan Jenkins (circa 1997), Alexandre Dumas, Robert Frost, Charles Stewart Kaufman (Spotless Mind), Robert Festinger (In the Bedroom), Ronald Harwood (The Pianist), Frank Darabont (Shawshank Redemption), Mike Figgis (Leaving Las Vegas), John Mayer (Kid A Acoustic), Bruce Springsteen (I'm On Fire), Dada (Dim), Fenix TX (Threesome), New Order (Love Vigilantes), Blink-182 (Man Overboard), Jeff Buckley (Last Goodbye), Jackson Browne (The Load Out), Duncan Sheik (That Says It All), Bleu (I Won't Go Hollywood).

Portions of this work have been previously published in:

Froehlich, J. and Dourish, P. 2004. Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams. Proceedings of the International Conference on Software Engineering ICSE 2004 (Edinburgh, UK), 387-396.

This work was supported in part by the National Science Foundation under award IIS-0205724.

ABSTRACT OF THE THESIS

Unifying Artifacts and Activities in a Visual Tool for Distributed Software Teams

By

Jon Edward Froehlich

Master of Science in Information and Computer Science

University of California, Irvine, 2004

Professor J. Paul Dourish, Chair

In large projects, software developers struggle with two sources of complexity – the complexity of the code itself, and the complexity of the process of producing it. Both of these concerns have been subjected to considerable research investigation, and tools and techniques have been developed to help manage them. However, these solutions have generally been developed independently, making it difficult to deal with problems that inherently span both dimensions.

We describe Augur, a visualization tool that supports distributed software development processes. Augur creates visual representations of both software artifacts and software development activities, and, crucially, allows developers to explore the relationship between them. Augur is designed not for managers, but for the developers participating in the software development process.

We discuss some of the early results of informal evaluation with open source software developers. Our experiences to date suggest that combining views of artifacts and activities is both meaningful and valuable to software developers.

Chapter 1 INTRODUCTION

Virtually all software systems of reasonable size are developed by teams rather than by individual software developers. In large-scale efforts, these teams may be distributed over wide geographical areas, and often may also be distributed in time (as team members come and go and the system evolves). Consequently, large-scale software development must deal with two sources of complexity: the complexity of the artifact being produced (the code itself), and the complexity of the activities around that artifact (the distributed process of software development.) Software process models (e.g. [Boehm and Bose 1994, Finkelstein 1994, Sutton et al. 1997]) attempt to help teams with the complexity of activities, while techniques and analysis and testing (e.g. [Egyed 2003, Naumovic et al. 1999, Richardson 1992]) focus on the artifacts.

Although any development effort will inherently involve both of these sources of complexity, most tools and techniques offered to software developers concentrate primarily on one or the other. It is, of course, possible to use tools of each sort in the course of development, and most well-managed software projects will endeavor to do so. However, since each tool deals only with one source of complexity, developers must switch back and forth to solve problems that involve combinations of the two. For example, activity-based tools can alert developers to their colleagues' activity and summarize recent updates, while artifact-based tools can analyze source code and highlight dependencies between modules; however, this separation makes it difficult to find, for example, which modules depend on those recently updated or currently being worked on by others.

To address this separation, we have developed a novel visualization system called Augur. Visually, Augur is based on the line-oriented approach pioneered by Eick and his colleagues with SeeSoft [Ball and Eick 1996, Eick et al. 1992]. Augur extends SeeSoft by combining information about the structure of both artifacts and activities within the same visual frame. This allows users to quickly establish where the development activity in the codebase is concentrated, what sort of code has been modified, and which developers are involved.

Augur supports two main uses:

1. **Monitoring activity in a distributed software project.** This provides developers with an enhanced understanding of the ongoing activities of their colleagues. Sample uses might be on a peripheral display or on a shared view in a project war room;
2. **Exploring the distribution of activities in time and space.** This allows developers to "drill down" to explore the history and context of particular development activities in the code base.

Four considerations have driven Augur's design. First, to better adapt to different development settings, Augur is designed to support end-user visualization rather than automatic inference. Second, Augur favors online rather than offline analysis for dynamic integration into the development process. Third, Augur is designed to be used by developers concurrently with development rather than retrospectively for management analysis. Finally, Augur's design emphasizes interoperability and extensibility so that it may be extended by other development efforts without significant overhead.

In this thesis, we first review the background of research into technologies for program comprehension and collaborative software development before exploring design criteria in more depth; we then introduce Augur and its underlying architecture. We close by presenting the findings of informal evaluations, and discussing opportunities for further work.

1.1 Background

A variety of tools and techniques have been developed to help programmers comprehend software systems. Such facilities are particularly important for effective software maintenance. For instance, software reflexion models can help programmers understand large systems by highlighting how the actual system relates to a high-level description of expectations [Murphy et al. 1995, Murphy and Notkin 1997]. Reflexion models can be valuable as developers analyze the structure of large software systems. The Rigi system also uses visual techniques to provide developers with a graphical overview of the structure of a software system [Storey and Mueller 1995]; Rigi is designed primarily to support reverse engineering tasks where a developer must achieve a working understanding of an unfamiliar software product. Similarly, software development environments have long included facilities that allow a programmer to inspect the internal structure of the software system being developed, at least as far back as Interlisp's Masterscope facility [Teitelman 1974].

These tools can give the software developer valuable insights into the structure of the system under examination—but our goal here is rather different. Augur is designed both to help developers understand a software system and to support them in coordinating collaborative development work.

Our particular interest is to do this by bringing together views of artifact and activities. The source code of the system is already the central focus of developers' activity. Is it possible, then, to enrich this artifact in such a way as to provide developers with information about activities?

One strategy is to allow the artifacts themselves to carry information about previous activities. Hill and Hollan (1994 and 1992) propose "history-enriched digital objects," information artifacts that carry with them records of the accumulated actions that they have sustained, in just the same way that dust, dog-ears, and thumb marks reveal which books on a shelf are read often, and which are not. This mechanism allows the artifact itself to convey information about the activities that have taken place around it.

Researchers in Computer-Supported Cooperative Work (CSCW) refer to this as "awareness" – the informal understandings people maintain of ongoing activity [Dourish 2001]. In a shared physical space, people can monitor each other's activities and use this information to coordinate their collaboration; for example, it helps them to deliver information when it is needed, predict upcoming tasks, know who to talk to about particular topics, avoid contention over shared resources, etc. In distributed collaborative work, where a shared physical space is not available, technology may provide channels that allow people to maintain an awareness of each other's actions.

Studies show that these informal means of information sharing exist alongside all formalized models, no matter how detailed. They are the mechanisms by which people put formal processes to work—understanding how and when to initiate actions, meshing independent activities, understanding upcoming actions, avoiding problematic situations, etc. A number of studies have noted the role that informal awareness plays in formalized

software engineering processes. For instance, Grinter's investigations uncovered how, in addition to their primary function, configuration management technologies also provided developers with a view of each other's activities [Grinter 1995]. In a more recent study, de Souza et al., reporting on empirical studies of a software development team, note that even with a complex configuration management system available to them, developers still conduct a good deal of "out-of-band" communication and monitoring in order to maintain a broad collective understanding of team activity [de Souza 2003]. More generally, formal processes can serve an awareness purpose; Dourish (1992) has suggested that that process models can be used not only to regulate but also to account for activity in collaborative settings, using the process description as a lens through which to see collective action as it emerges.

These observations have prompted researchers to develop technologies specifically designed to promote awareness in collaboration. For example, RearViewMirror [Herbsleb et al. 2002] uses Instant Messaging technologies to support inter-developer communication that are integrated with their development activities; alternatively, Palantir [Sarma et al. 2003] provides an awareness framework that operates in concert with configuration management systems.

Our research stems from the observation that developers struggle with the complexity of both the artifact and the development activities around that artifact – and that, historically, there has been a lack of serious research effort to utilize configuration management data and source code analysis techniques to investigate the relationships between the two.

Chapter 2 RELATED RESEARCH

There are three primary areas of related work worth noting here. First, we examine the general area of software visualization research and its contribution to program understanding. Second, we look more specifically at a set of software visualization systems descending from the original SeeSoft system. Finally, we look at other approaches for understanding activities and artifacts in development, particularly those that rely on configuration management systems for activity data.

2.1 SOFTWARE VISUALIZATION

The essence of the visualization approach is to shift load from the cognitive system to the perceptual system, capitalizing on the human visual system's ability to recognize patterns and structures in visual information [Robertson et al. 1993]. Recent work in the scientific visualization community has demonstrated how visual representations support the rapid assimilation of information [Tufte 2001, Harris 2000]. They show how significantly large data sets can be reduced to graphic form in such a way that human perception is able to detect patterns and latent structures that are otherwise unavailable or inaccessible in the original format [Robertson 1993]. Research has also shown that the eye's optical system has extraordinarily good visual acuity. Tufte observes that "our eyes can make a remarkable number of distinctions within a small area. With the use of very light grid lines, it is easy to locate 625 points in one square inch, or equivalently, 100 points in one square centimeter" (2001). Most visualization systems endeavor to leverage these innate properties of the human visual system.

Software visualizations, in particular, have a long, varied history in computer science. One of the earliest systems, developed by Haibt, could draw flowcharts automatically from Fortran and assembly language source code [Price 1993]. The system was meant to aid developer understanding of the data flow in their programs. In 1966, Ken Knowlton, a researcher and pioneering computer artist at Bell Labs, was the first to use dynamic rather than static techniques to visualize software and also the first to address the visualization of data structures [Price 1993]. Early research in the 1970's extended software visualizations into the pedagogical domain and was used to teach students about the inter-workings of software programs (e.g. algorithmic execution). The late 1970's and early 1980's were marked by research to improve the readability of code. Termed "pretty-printing" by Ledgard in 1975, the use of spacing, indentation, and layout was studied and tuned to make source code easier to read [Clifton 1978]. This was some of the first research in attempting to understand the implications of the visual structure of source code. By the mid-1980's, the introduction of cheap, bit-mapped displays along with the proliferation of windows-based computers helped usher in a new era of software visualizations. Since then, the field has continued to expand through both technical advancements (e.g. high-resolution displays, increases in memory size and computational power) and advancements in psychology and cognitive science (as researchers continue to learn how the human vision system interprets features of the world).

Currently, the term software visualization is used to describe a myriad of topics from program and data structure visualizations to visual programming techniques and algorithm animations [Maletic et al. 2002]. These topics are meant to support a wide range of tasks, from those rooted in software engineering and program comprehension

(e.g. supporting programming, debugging, testing, and code maintenance) to those in education (e.g. teaching algorithm design and time complexity analysis). Our visualizations are meant to support the process of software development both to increase understanding of the software system itself *and* the activities that produced it.

2.2 SeeSoft Visualization Applications

The SeeSoft visualization method is a common technique used to display large amounts of source code on a single screen.

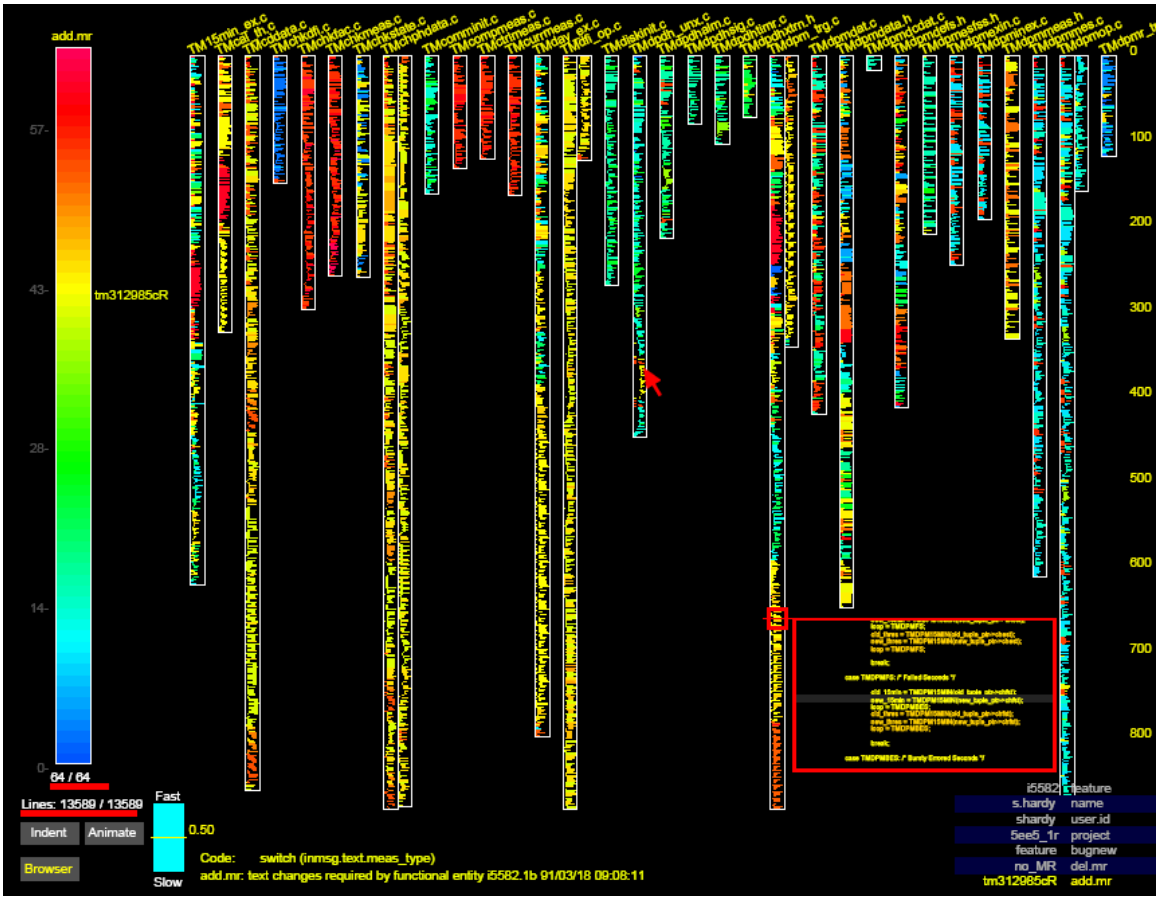

Figure 2.1 SeeSoft screenshot displaying 13,589 lines of code [Eick, 1994] Files are represented as columns underneath their respective names. Each colored line in a column is a proportionally reduced line of text. The blue lines were written in 1990, the yellow lines in 1991, and the the red lines in 1992.

A range of systems have been developed that draw on Eick's pioneering investigations of line-oriented visualization of software statistics. His research group has generated a range of extensions to the original line-oriented view; in their more recent work, they have explored web-based visualizations, as well as a similar "linked visualizations" approach for large-scale software visualization that Augur uses as well [Ball and Eick 1996, Eick et al. 1992]. By incorporating many perspectives, their tools can provide an extremely rich picture of organizational software activity. However, their tools focus on analyzing change records, and not the source code itself, making it difficult to present rich descriptions of the relationships between them. Similarly, their analyses are oriented more towards managers trying to understand organizational action rather than developers attempting to understand their own work and the work of their colleagues as it unfolds.

Griswold's Aspect Browser [Griswold 2001] applies the SeeSoft visualization method to support software maintenance and evolution (see Figure 2.2). Aspect Browser is designed to help a software developer understand how particular features of a system are distributed through the code. Drawing on work in Aspect-Oriented Programming [Kiczales et al. 1997], Aspect Browser is targeted particularly towards cross-cutting concerns—features that touch many parts of the system, cutting across the modular organization of the system. These are particularly difficult to track down, especially in large programs, and so Aspect Browser's visual overview is especially helpful. Aspect Browser clearly focuses more on the artifact than on activities. Given its concern with cross-cutting aspects, though, it defines aspects textually, using regular expressions,

rather than structurally (as Augur does), in terms of the semantic organization of the code.

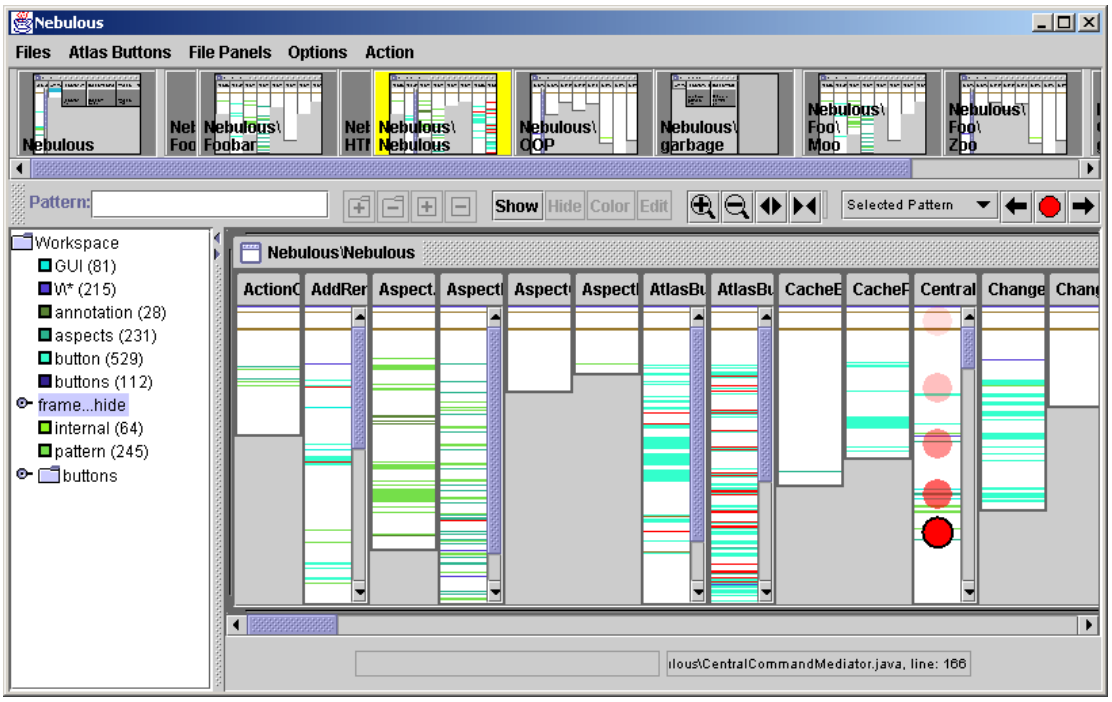

Figure 2.2 A screenshot of Aspect Browser[1]
Aspect Browser uses the SeeSoft view to highlight specific words or phrases that may occur across files. The filetree (left panel) serves as a legend denoting assigned colors for crosscuts and their number of occurrences; each occurrence of a crosscut is highlighted in the SeeSoft view with a specific color. This allows the user to quickly see how a crosscut is dispersed in a project.

A third example of the extended use of line-oriented visualizations is provided by Tarantula [Jones et al. 2002]. Tarantula uses a line-oriented visual display to assist with test analysis and fault localization by visually indicating the degree to which each line of code participates in successful or unsuccessful outcomes from a test suite (see Figure 2.3). This approach to fault localization draws on three properties in which the line-

[1] http://www.cs.ucsd.edu/users/wgg/Software/AB/

oriented visualization method is particularly suited: high-level overview, spatial organization, and cumulative statistics. Like the Aspect Browser, Tarantula uses a SeeSoft-like display to focus on features of the software itself, rather than features of the development process.

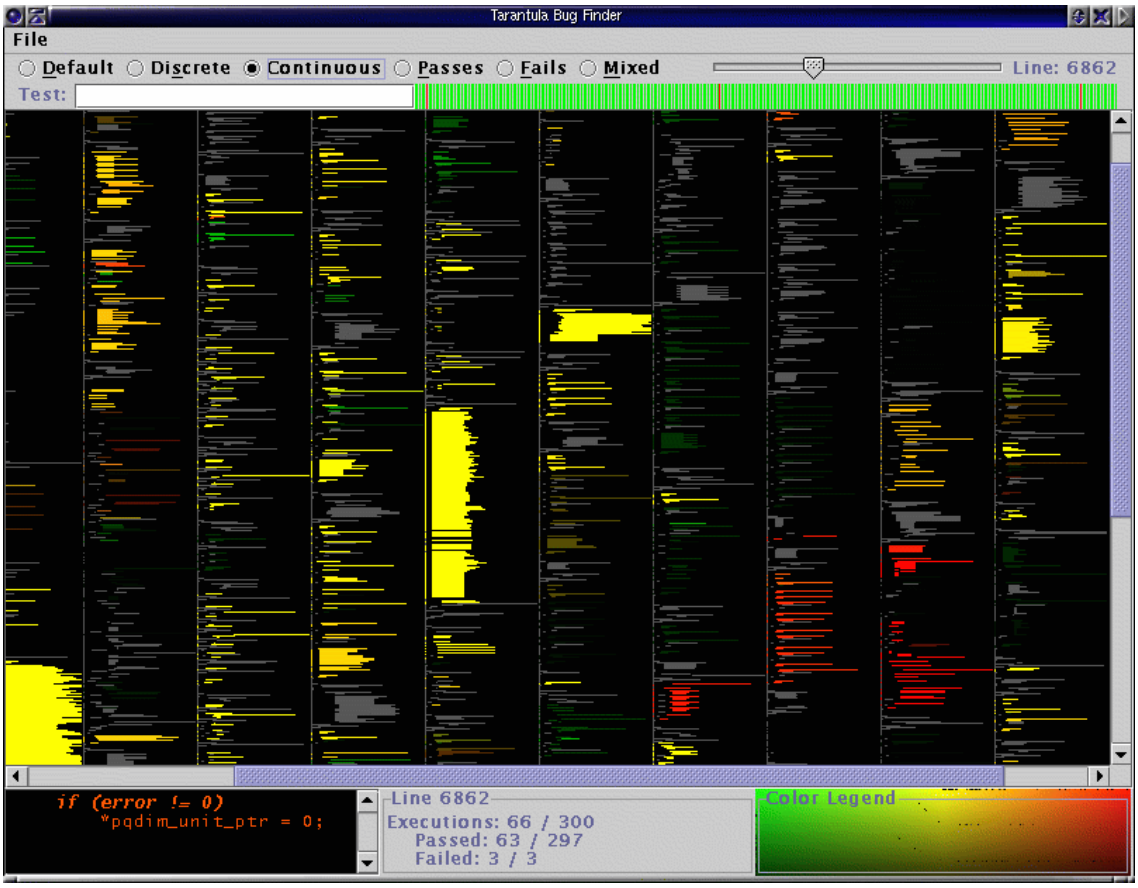

Figure 2.3 Tarantula uses the SeeSoft view for test analysis [Jones, 2002] Tarantula displays lines colored according to their relative success rate as executed in a test suite. Statements that pass a test case progressively become more green while statements that fail test cases progressively become more red. Statements shown in yellow are more ambiguous in that they have both passed and failed test cases.

There are two points to note here. The first is that both artifact-based and activity-based software technologies are broadly useful in software development, as demonstrated by these systems. The second is that the same visual approach has been successfully

applied to problems of each sort. Augur's unique contribution is in the relationship that it draws between the two.

2.3 Understanding Activities and Artifacts

A second set of related investigations concern the relationships between system components or artifacts on the basis of development activities. This body of research typically relies on data archives (e.g. software repositories, bug report databases) to retrieve activity information.

An emerging field termed "Mining Software Repositories" looks at how data available in version control systems (like CVS or Subversion) and other online archives (mailing lists, discussion boards, bug tracking systems, etc.) can be leveraged to better understand various aspects of software development. Early research in this area was limited to studies of large commercial systems as they were the only source of rich historical version control data and large, complex, development projects. Eick et. al, for example, developed SeeSoft in conjunction with the 5ESS Telecommunications Switch Project at AT&T Bell Laboratories, which included millions of lines of code developed over a 10 year period [Eick et al. 1992]. This dependence on proprietary systems made it difficult for researchers to get access to software repositories without industry affiliations. The popularization of open source software on the internet, however, has provided relief from this constraint. Researchers can now access software repositories with projects comparable in size, scope, and complexity to their industry counterparts (e.g. Open Office[2], Linux[3], Apache Web Server[4], etc.). The success of open source has helped

[2] http://www.openoffice.org

advance research focused on using data archives to study software development. A selection of this research is highlighted below.

Experimental systems developed by Bieman et al. (2003) and by Zimmermann et al. (2003) aim to uncover structural relationships between artifact and activities in software development. The ROSE system developed by Zimmerman and colleagues models the relationships between different system components on the basis of "evolutionary dependencies." Dependencies are indicated when two modifications to one component are always accompanied by modifications to another. Where Zimmerman et al. break files down into functions and variables, Biemen et al. develop similar models at the level of class relationships, and use pattern-based relationships to inform class clustering.

Hipikat [Cubranic and Murphy 2003] solves a related problem. It is primarily designed to help newcomers in a project become familiar with its structure quickly. Hipikat treats project archives (including source, bug tracking information, and discussion lists) as a group memory. It helps users navigate them, based on a recommendation approach; as the user examines the system archives, Hipikat recommends other related artifacts that the user might be interested in based on similarity measures.

Expertise Browser [Mockus and Herbsleb 2002], like Hipikat, relies on more than just version control data for its analysis. Expertise Browser uses both version control records and modification change request data to quantify "experience." These quantitative measures are then used to locate people with a desired expertise. "The main idea behind

---

[3] http://www.linux.org

[4] http://www.apache.org

Expertise Browser is visually to query and present relationships between product (code, documentation, design, functionality) and the people or organizations that have a desired type of experience with respect to these artifacts" [Mockus and Herbsleb 2002]. The tool enables developers, for example, to distinguish between someone who has worked only briefly in a area of source code from someone who has more extensive experience with that same piece of code.

The relationships between artifacts derived by all four of these systems are possible approaches to incorporate into Augur; they are new forms of analytic interpretation that can reveal structure. Augur is designed to make this expansion possible and effortless; new source code analysis techniques can be easily added to the structural analysis subsystem (see Section 4.2.2).

There are four primary differences between the approach we have taken and these alternatives. First, we take a visualization approach that is focused on revealing relationships between artifact and activity within the context of the source code itself. That is, the source code itself is inhabited with activity information through the use of annotation columns, not some artifact abstraction (see Section 3.2.1). Second, we combine multiple linked-visualizations within a single tool, allowing developers to move easily back and forth between different aspects of the system being examined. Third, assuming that the source code itself is an artifact that all developers understand, we use it to provide a common spatial model for all views. Our line-oriented display serves both as an exploratory space to inspire interaction and provide an expansive view of the source code and as a device to tie all of our visualizations together. Interesting patterns discovered in one visualization can be related back to the source code itself through the

line-oriented display. These associations are important as the code is, ultimately, the focus of the development activities and not some high-level visual abstraction (e.g. a call graph view). Finally, Augur is not simply a development tool but also a generic framework for visualizing source code. Augur could be easily expanded, for example, to incorporate the visual features of the Tarantula system (e.g. coloring lines by test suite data).

Chapter 3 OUR APPROACH

Our current research proceeds from the observation that software teams struggle with *both* the artifact (e.g. source code) and the activities of development as sources of complexity. Accordingly, we have been developing tools that provide a unified approach and help developers to understand the relationship between them.

3.1 DESIGN CONSIDERATIONS

Our goal is not simply to help developers analyze the codebase, but to help them analyze the activities that occur around it. This change in focus leads to a number of design considerations.

Concurrent vs retrospective. One particularly important issue, which holds implications for the rest of the design, is whether this system is intended primarily for retrospective analysis of development activity, or whether its primary use is for analyzing activity that is currently in progress. Clearly, a case can be made for either; for example, retrospective analysis could support software process improvement and process reengineering. However, empirical studies such as that by de Souza et al. (2003) point to the ongoing problems of coordination within software teams, indicating a need for awareness tools that can be integrated into current practice. A purely retrospective tool would be inadequate for these requirements. It is important, then, that Augur be able to operate alongside existing technologies and provide concurrent views of development activity.

Online vs offline. A related issue concerns the balance between online (during runtime) and offline (before runtime) analysis of source code. Offline analysis can

provide more information, but at the cost of both delays and infrastructure complexity. In the interests of supporting concurrent exploration, we have chosen as far as possible to emphasize analysis that can be performed during runtime. Augur does not need to, for example, preprocess source files in order to perform source code analysis, but rather, can perform these operations during runtime (through the inclusion of source code parsers and structure analyzers, see 4.2.2).

Augur does not, however, currently support real-time version control updates. Augur connects to the version control system once during program startup and disconnects after the data loading stage completes. Therefore, Augur would need to be restarted to check for updates or changes to the loaded project. This design decision is a reflection of the convential passive, non-event driven, version control server/client data model (i.e. CVS and SubVersion client's pull data from the server). To support an event-driven version control system, Augur's underlying database infrastructure would need to be modified slightly (for more information on Augur's architecture, see Chapter 4).

Interoperability. We would like Augur to be broadly usable in real engineering practice. This means that it must be interoperable with a range of existing tools and infrastructures; it must not present significant infrastructure demands, or require that developers and development organizations abandon their own tools and methodologies. We have designed around an open architecture that can support different source code repositories (such as CVS, Subversion, Visual SourceSafe, etc.) as well as providing a consistent framework for analytic extensions.

Visualization vs interpretation. The most formidable balance to be resolved by the design is between visualizing information for the end user and interpreting it for the

system. Visualization approaches create visual depictions of information that allow users to perceive patterns and correlations; the alternative is to have the system interpret the information directly and automatically derive conclusions about system activity.

We believe that each development setting is different, and that the correct interpretation of activity information depends critically on local factors. Accordingly, our overall approach is visualization-based. Rather than encoding specific workflows, we provide a visual tool that allows developers to explore views of their system and its activity. Research into distributed cognition and external cognition has demonstrated the important role of representations in information processing tasks [Hutchins 1995, Scaife and Rogers 1996]. For example, long multiplication and division is much easier to carry out using Arabic numerals than Roman numerals; essentially, the effort is shared between the individual and the external representation. Similarly, information visualization helps users "offload" information processing to the visual representation.

There are two reasons to take a visualization approach to this particular problem. The first is that software development is a particularly complex task, and the needs of individual projects are uniquely based in their specific domain and development history. In the face of this variability, we find it more effective to provide users with flexibility rather than to make assumptions about their needs. The second is that this approach allows us to proceed without committing to particular development processes or organizational contexts.

Note that Augur is not a development technology itself; rather, it is a visualization system that accompanies existing development tools by providing a view of the software development process as it unfolds. Augur embodies the philosophy that there is value in

relating information about an artifact within the context of the artifact itself; specifically Augur attempts to unify information about development activity and source code within the same visual frame.

3.2 AUGUR: INTERFACE AND INTERACTION

Augur provides a set of linked visualizations displaying different characteristics of the software system under examination. The primary visualization is a spatially-organized view of the software artifact, inspired by SeeSoft [Eick et al. 1992]. SeeSoft and related systems (e.g. Tarantula [Jones et al. 2002] and Aspect Browser [Griswold et al. 2001]) present an overview of a software artifact in which each line of code is represented by a line of pixels colored to indicate an attribute of the line such as author or modification history.

3.2.1 Augmented SeeSoft View

An example of the SeeSoft visualization technique is displayed in Figure 3.1. Note that the SeeSoft view is both spatially and proportionally consistent with the original text file (e.g. it retains spatial properties like line length, spacing and indentation); however, the text itself is not readable. By obfuscating the text, SeeSoft is able to reduce its file representations to the pixel line level (1 line of text = 1 pixel line row). This results in extremely high information density (approximately 50,000 lines of code can be simultaneously shown on a standard screen [Eick et al. 1992]).

HelloWorld.java **HelloWorld.java**

```
/**
 * This is a test class to demo Augur!
 * @authors Jon, Sandy, Edward
 */
public class HelloWorld {

 /**
  * Default constructor
  */
 public HelloWorld(){
   Log.println("Constructor called");
 }

 /**
  * Prints HelloWorld msg & logs call
  */
 public void printMessage(){
   Log.println("printMessage called");
   System.out.println("HelloWorld");
 }
}//end HelloWorld class
```

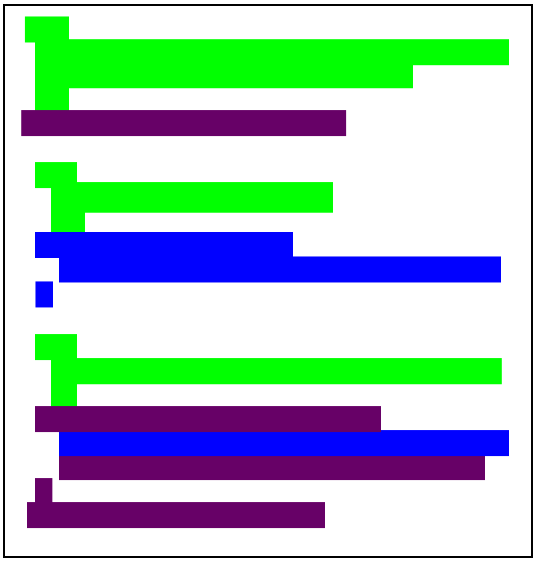

Figure 3.1 Example of the SeeSoft view
(left) HelloWorld.java source code file; (right) HelloWorld.java source code file as rendered by SeeSoft. In this example, the lines are colored by modification date (see Figure 3.2).

The SeeSoft view in our example text file, HelloWorld.java, is colored by modification date (though the lines could be colored by virtually any property—e.g. author, modification count, fault testing data). In this case, coloring by modification date allows us to easily determine the ages of the lines; it's clear that there have been three modifications in this file (represented by the colors green, blue, and purple—see Figure 3.2 below for color descriptions). However, it is difficult to elaborate on this observation, i.e. what type of modifications were made, and by whom?

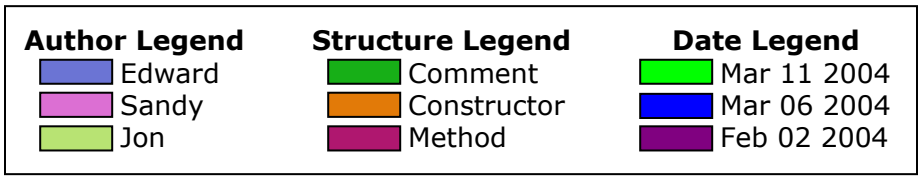

Figure 3.2 Legends describing colors used in Figure 3.1 and Figure 3.3

Coloring by only one property at a time makes it difficult to answer these questions. Augur extends the SeeSoft visualization method to address this issue by displaying multiple properties in a single visual frame. We do this by augmenting SeeSoft with two annotation columns. As with the text lines, these annotation columns can be mapped to a variety of line-oriented attributes or statistics. In Figure 3.3, the left annotation column is colored by author while the right annotation column is colored by structure; the text representation continues to be colored by modification date (see Figure 3.2 for color mappings).

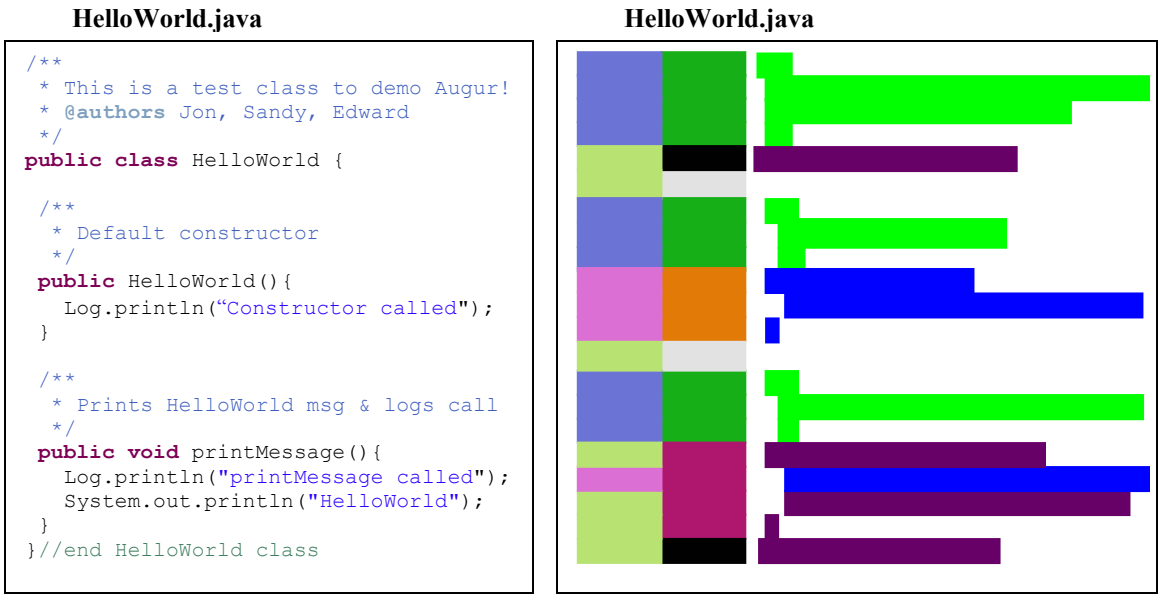

Figure 3.3 Example of the Augur view
(left) HelloWorld.java source code file; (right) HelloWorld.java file as rendered by Augur. Here, the left annotation column is colored by author; the right annotation column is colored by structure; the text representation is colored by modification date (this last part is the same as Figure 3.1). By displaying this information together, the user is able to make richer characterizations about activity.

By augmenting the SeeSoft view with annotation columns, interesting relationships between elements (like author, structure, and time) begin to emerge. For example, in Figure 3.3, it is not just that there were three separate modifications in the

HelloWorld.java file (as can be inferred from the unaugmented SeeSoft view in Figure 3.1), but that these modifications were done at three different times by three different authors on three different types of code structures. The annotation columns make it possible to observe, for example, that:

- Jon was the first author of HelloWorld.java. He also appears to be the original author of the printMessage method.
- Later, Sandy added a constructor and modified one line in the printMessage method.
- Finally, the most recent modification in the file was Edward's addition of three comment blocks. Edward has not contributed any source code lines.

These types of characterizations are only possible through the juxtaposition of views.

By default, line age is the primary display mode in Augur. This mode colors line text by its most recent modified date (where younger lines are green and older lines are purple). A slider-bar control allows the user to select a "time window" of interest (e.g. the past 24 hours, the past 2 weeks, the entire time of the project, etc.), which distributes the color gradient accordingly. Another similar display mode, called history, allows the user to scroll through each check-in from the beginning of the project until the end.

In the line age display mode, the subsidiary elements are author and code structure, indicated in the adjoining columns (like in Figure 3.3). However, users can switch back and forth between different configurations of primary and subsidiary attributes, e.g. making author or structure primary, in order to more easily examine the relationship between system structure and development activity.

The next section highlights a number of these structure/development activity relationships which are perceptible only through the juxtaposition of annotation views.

3.2.2 Activity/Artifact Patterns

Augur's augmented SeeSoft view is meant to expose the relationship between latent features situated in or around source code and the source code itself. We do this by presenting both the latent features and the source code together in a shared visual frame. The following three examples are intended to demonstrate the value in this approach: Method Addition Patterns (Section 3.2.2.1), Java Interface Patterns (Section 3.2.2.2), and Structure Modification Patterns (Section 3.2.2.3). Each example illustrates different types of relationships between artifact and activity that persist in the development process. Note that the figures in this section maintain Augur's default color mappings: the left annotation column is author, the right annotation column is structure and the line display is colored by age (the same color mapping was used in Figure 3.3 in the previous section).

Line age is colored on a continuous scale from light green to blue to dark purple (light green is most recent; dark purple is least recent). Structure is divided into seven distinct sections:

Table 3.1 Java Structure Colors

	Structure Type	Color
1.	Class Variable Definitions	Yellow
2.	Constructor Blocks	Orange
3.	Import Statements	Violet
4.	Method Blocks	Maroon
5.	Multi-line Comment Blocks	Dark Green
6.	Single-line Comment Blocks	Green
7.	Whitespace	Light Gray

These color assignments are also presented in image form in Section 3.2.5 (Figure 3.9). Note that whitespace is only colored light gray outside of method or constructor blocks. Whitespace that occurs inside method blocks is colored the method block color: maroon. Similarly, comments that fall within method or constructor blocks are not highlighted either.

3.2.2.1 Example 1: Method Addition Patterns

Figure 3.4 shows two different types of development activities embedded in the source code. In the left file, the two most recent modifications consist of, first, the addition of four methods by the Brown author and then, later, the addition of four comment block headers around those methods by the Pink author. Comparatively, the file on the right also consists of two recent modifications conducted by two different authors.

In both files, these modifications involve adding comments and functional aspects of source code. However, the activity in the left file is centered on the addition of four methods, while the activity in the right file is centered around the addition of one large constructor block.

Note how, in both files, one author appears concerned with adding and maintaining comments, while another author is concerned with modifying the source code itself. This is somewhat surprising given that it seems contradictory to good software engineering practice (e.g. “comment as you as go”); however, in Java open source projects (where these two files were obtained), one author is often assigned the role of adding and maintaining comments so that, for example, each file is Javadoc friendly and meets the comment guidelines of the project.

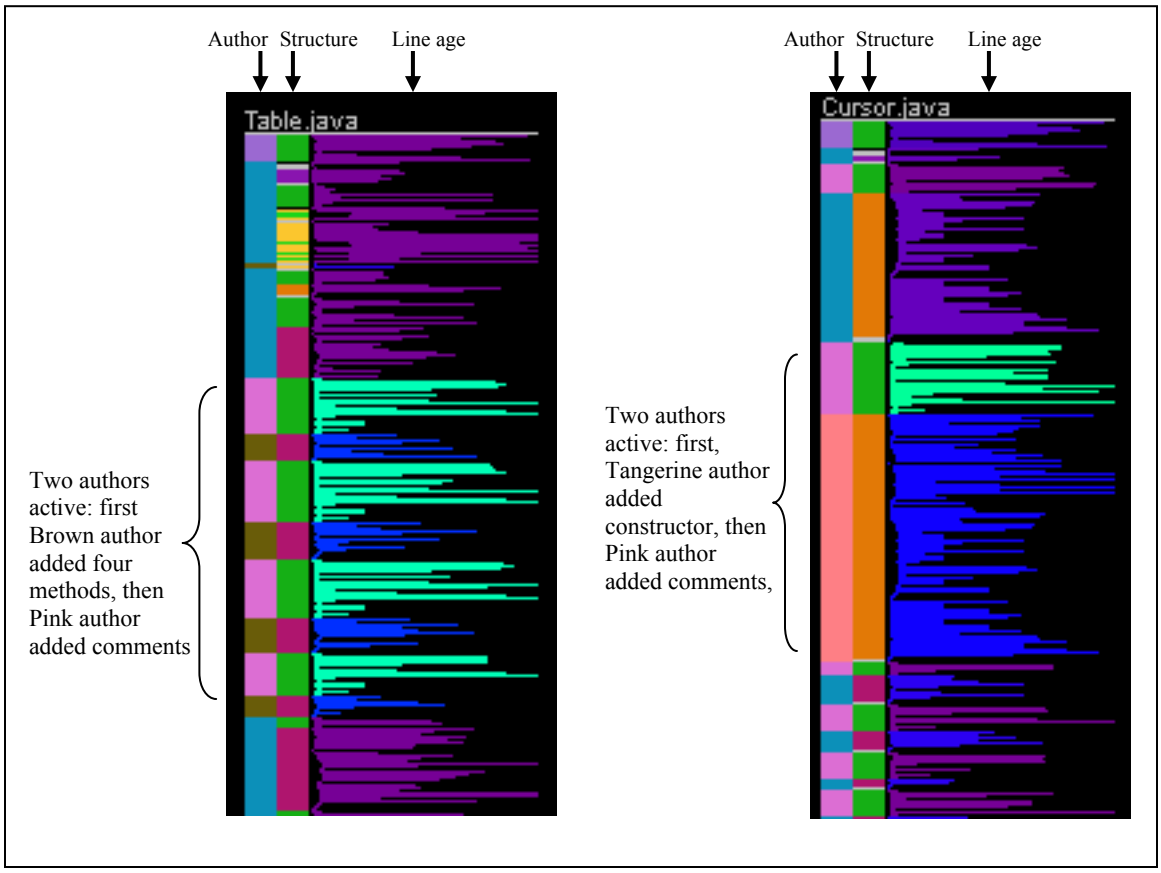

Figure 3.4 Method addition patterns
(left) Two authors have recently modified the left file: first, the Brown author added four methods, then later, the Pink author added comment headers to these methods. (right) Unlike the left file, the recent modifications to the file on the right involve a constructor: first, the Tangerine author added a large constructor and then the Pink author added a comment header to this constructor.

Also note that in these two files, source code was modified in larger structural units than at the line level. It is not just that a collection of lines were checked in, but that these lines make-up larger structural units (e.g. a method, a constructor, or a comment block). These types of inferences are only possible through the unification of annotation columns and source code.

3.2.2.2 Example 2: Java Interface Patterns

The two files portrayed in Figure 3.5 have structural patterns that are consistent with the attributes of a Java interface. Most object-oriented programming languages have the functionality of interfaces (sometimes called protocols) built into the language itself. Typically interfaces define a set of methods or static variables that can be implemented by any class in the class hierarchy. Because of this generality, interfaces are usually very well commented to ensure that their use is well-defined and understood by developers.

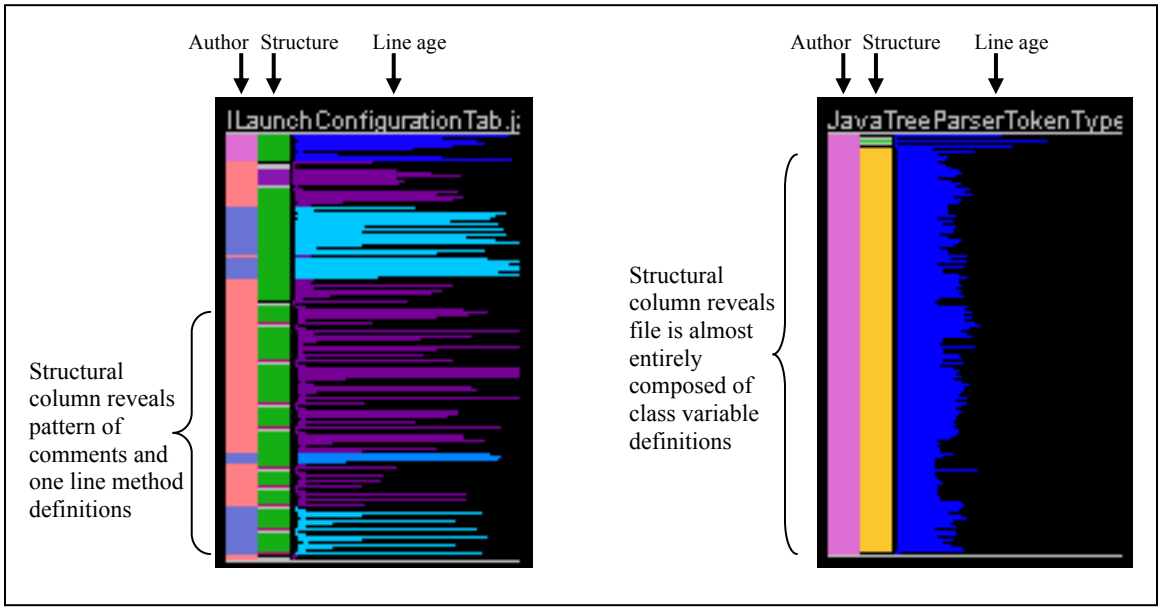

Figure 3.5 Java interface patterns
(left) The left file contains the visual attributes of an interface: in the structure column there is a pattern of alternating comment blocks followed by one line method definitions. (right) Similarly, the file on the right also has an easily identifiable type—it is almost entirely composed of class variable definitions (another attribute of an interface in Java).

The files in Figure 3.5 seem consistent with this description. The left file is almost entirely composed of alternating medium-sized comment blocks and one-line method definitions, while the right file is composed almost exclusively of static variable definitions. In fact, both files are interfaces. The left file, ILaunchConfigurationTab.java,

is an interface from the Eclipse[5] project for defining launch configuration tabs in their UI while the right file, JavaTreeParserTokenTypes.java, is a parser interface for defining Java token types as static integers.

Structural patterns like the ones shown in the structural annotation column of Figure 3.5 allow the user to detect file types visually rather than having to focus in on the filename or textual composition. This allows the user to, for example, quickly identify interfaces in a project, which are not always obvious from reading the filenames alone.

Of course, even more finely grained observations can be made about these files. For example, one can observe that the static variables in the right interface in Figure 3.5 were all added at the same time by the Pink author—the only author in the file. Or that, in the left file, there are three authors contributing code, but that one of them (Tangerine author) added a majority of the Interface's method definitions.

3.2.2.3 Example 3: Structure Modification Patterns

The last example, shown in Figure 3.6, relates how the structural columns allow us to characterize modifications on the basis of severity. The left file in Figure 3.6 has a series of small changes. Notice, however, that they are all located in comment blocks. These modifications are fairly benign compared to the multi-line modifications of methods in the right file.

[5] http://www.eclipse.org

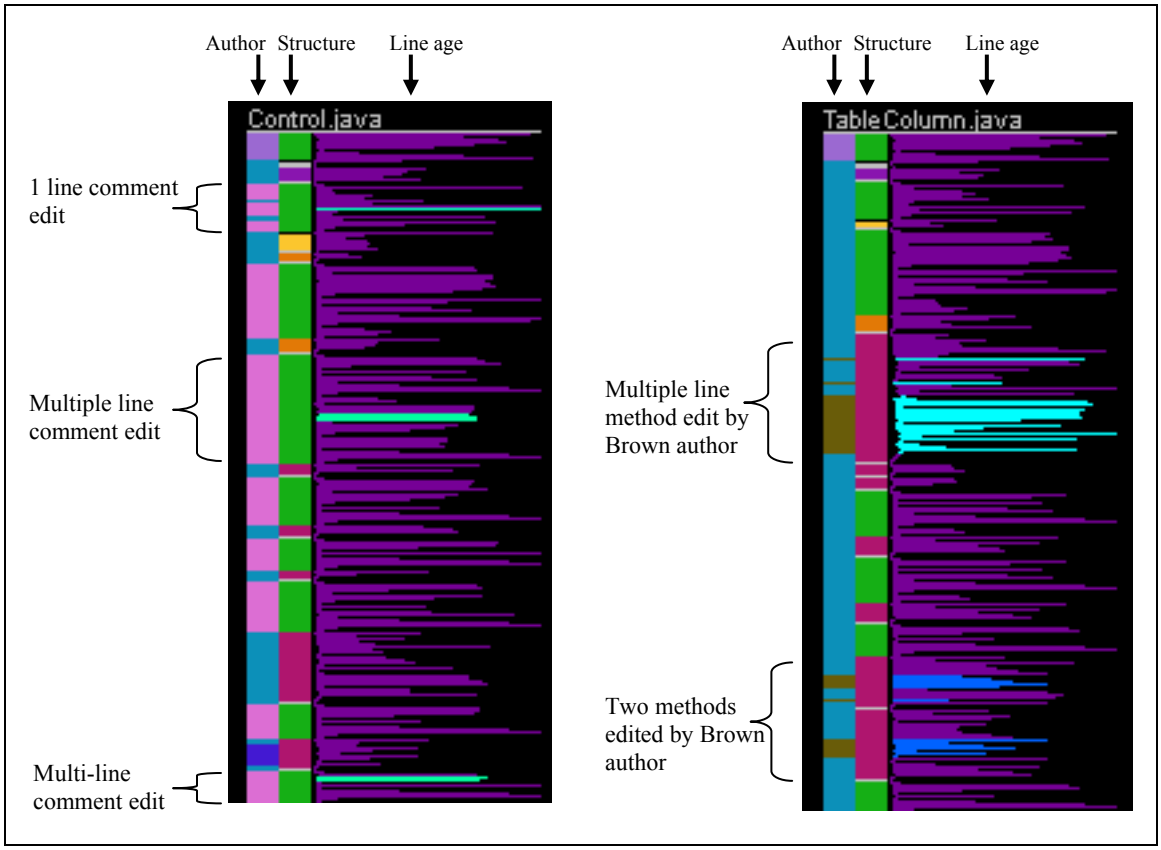

Figure 3.6 Structure modification patterns
(left) The recent activity in the left file is simply modifications to existing comment blocks. (right) Conversely, the activity in the file on the right has modifications to existing method blocks—these modifications were conducted by an author different than the one who originally programmed the methods. The modifications in the right file would be seen as potentially more invasive than the modifications in the left file.

If we ignored the two annotation columns and just focused on the spatial attributes of the temporal pattern in the right file, we might infer that the bright sky-blue block made up an entirely new method when, in fact, it is modifying an existing one. Clearly there is a distinction between "adding a new method" (as we saw in Example 1, Figure 3.4) and "modifying an existing one" (as we see in this example). In addition, we can determine that this particular method modification was done by an author different from the original method author (i.e. so, two authors have co-edited this method).

Questions about author relationships like, “who is working with who and when?” or, at least, “which developers are modifying the same file and when?” are more easily answered through Augur’s secondary visualizations (particularly the social networking graph visualization, see Section 3.2.7.4).

Although the previous three examples emphasized the activity/artifact patterns in Augur’s line-oriented file views, they are not the only visualizations in the system. The next few sections provide an overview of the Augur graphical user interface and the secondary visualizations.

3.2.3 The Augur Graphical User Interface

The basic Augur interface is shown in Figure 3.7. Each pane displays a different aspect of the system being examined: changes in one view are immediately reflected in the others. The large central pane shows the line-oriented view of the source code. Figure 3.7 is using Augur’s default color mappings: the color of each pixel line indicates how recently it was modified (coloring by line age); this allows a developer, at a glance, to see how much activity has taken place recently and where that activity has been located.

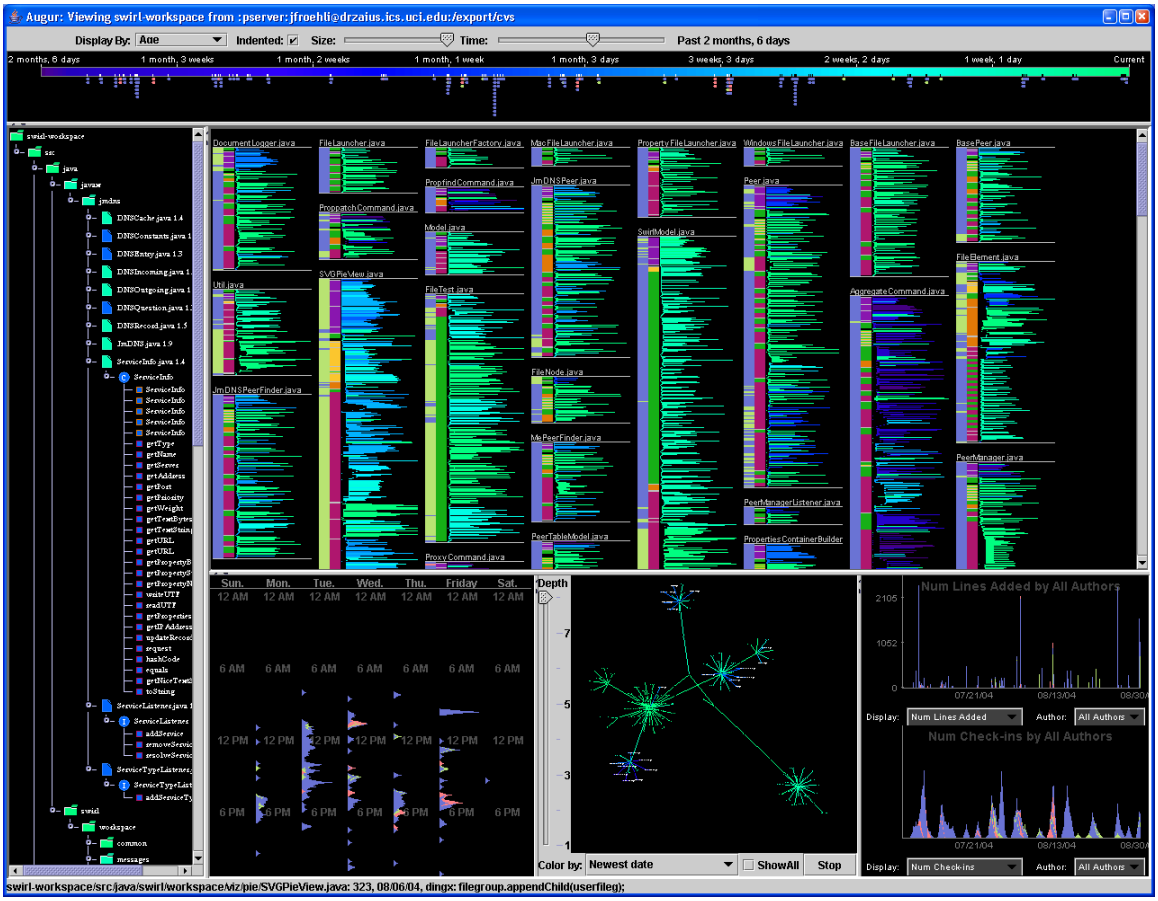

Figure 3.7 Screen capture of the Augur graphical user interface

Augur uses three techniques to integrate information about the artifact and its associated activities: annotation, interaction, and triangulation.

Annotation. In order to bring views of structure and activity together, the primary line-oriented display is annotated with subsidiary information in two extra columns that run down the left-hand side of each file block (as described in Section 3.1 and 3.2).

Interaction. The interaction design extends this relationship between structure and activity. When the developer clicks on any text line in the line-oriented display, Augur highlights two other sets of lines: first, the other lines of code checked in at the same

time, allowing the developer to see the extent of the check-in and the relationship between different parts of the system, and second, the structural blocks (e.g. methods) within which those lines are embedded. So, while the relationship between artifacts and activities is initially conveyed through the visual representation, it is reinforced by the application's response to user interaction.

Triangulation. The third mechanism by which the relationship between activity and artifacts is made clear is through the use of multiple, coordinated displays. While the line-oriented view occupies the central area of the interface, a number of other panels accompany it displaying related views of the system under examination. These views are largely graph-oriented, and show cumulative breakdowns of information to accompany the main display. For instance, in the view in which lines are colored by modification date (age), the accompanying panels show the files according to their overall change history and a detailed change graph for the currently selected file. Similarly, when "author" is selected as the primary attribute, the graphs display information concerning each author's history. These views are also interactive; selecting specific objects or events in the secondary displays will also cause information to be displayed or highlighted in the primary view. This allows developers to "triangulate," moving back and forth between displays to narrow in on the details they want to find.

The primary visualization window contains the line-oriented file view. The secondary visualizations provide unique views of data (line graphs depicting project growth, check-in activity graphs, social network graphs, etc.). The filetree Explorer view presents an overview of the file/folder hierarchy. Each of these views make generous use of the color mappings (defined in the color legends) and are tied together by the Augur event system.

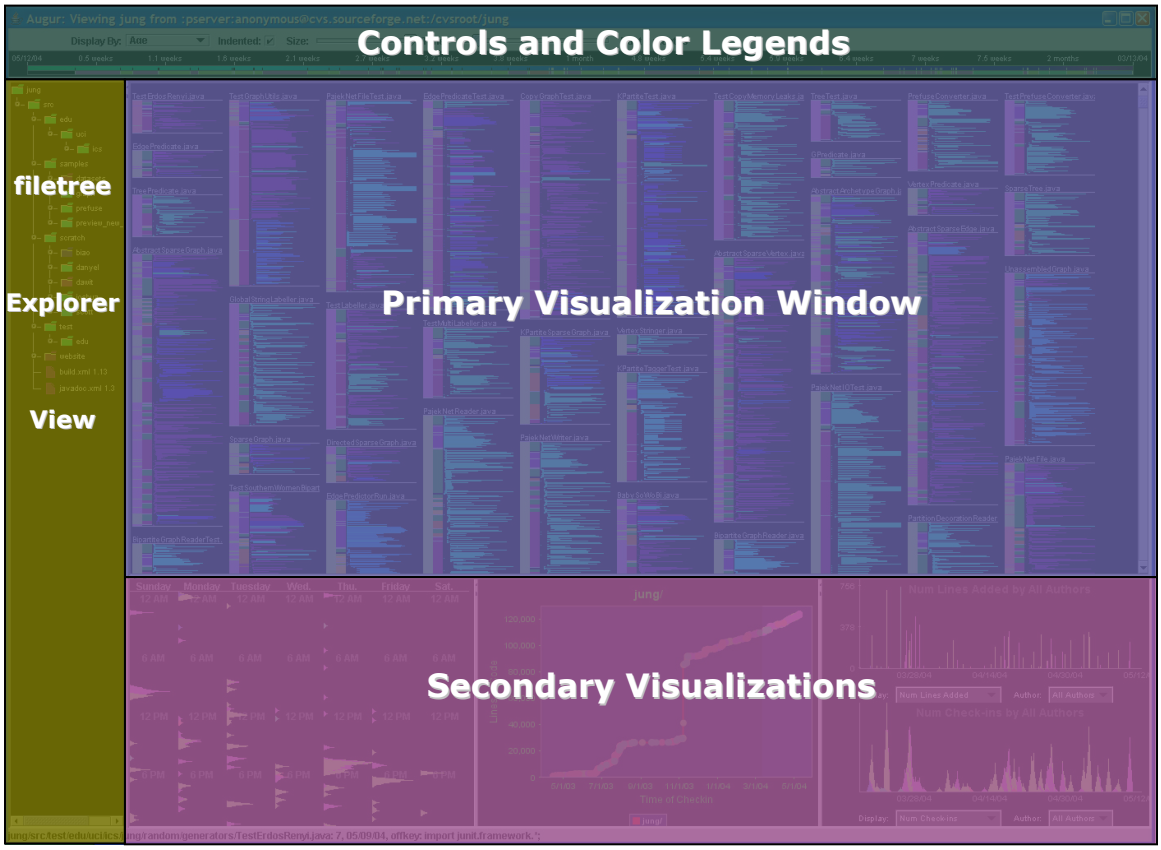

Figure 3.8 The Augur graphical user interface split into four sections

The Augur display can be partitioned into four areas (see Figure 3.8): the primary visualization, the color legends, the secondary visualizations, and the filetree explorer view. The line-oriented augmented SeeSoft view is located in the primary visualization window. The color legend is at the top of the interface and provides information on color mappings. The secondary visualizations are located in three resizable panes at the bottom of the interface. And, finally, the filetree explorer is located on the left side of the interface and presents a selectable view of the loaded project's file/folder hierarchy. The following sections examine these displays in more detail.

3.2.4 Primary Visualization

The primary visualization window is composed of the line-oriented file panels. By default, the file panels are arranged by modification date. The most recent active files are displayed at the top while the least active files are displayed at the bottom. Other layouts organize the file panels alphabetically, or by path, number of authors, or number of revisions, etc. The filetree explorer selects which path (or set of paths) is being viewed in this window. By default, the top level folder in the filetree is selected, which includes every path in the loaded project.

A magnifying glass can be used to reveal the actual contents of the code in the file panels themselves. A zooming mechanism is also available (via the size bar), which changes the size of the file panels. Zoomed all the way in, the file panels actually become readable text files where each line is "highlighted" by a background strip of color information. Zoomed all the way out, the file panels are approximately five pixels in width such that they appear only as abstract rectangular blocks. The line colors in the blocks, however, are still discernible. Therefore, zooming out is a particularly powerful way of using Augur as it can display extremely large amounts of code on the screen simultaneously (between ~50,000 and 100,000 LOC depending on the arrangement and size of the window).

3.2.5 Color Legends

The color legends serve an essential role in the Augur interface by relating color mappings to the user. There are currently two types of color legends: continuous and discrete. The continuous color legends relate coloring information for continuous data sets (like time). Similarly, the discrete color legends relate coloring information for

discrete data sets (like authors, structure types, line types, etc.). Figure 3.9 displays both a continuous color legend (top) and a discrete color legend (bottom).

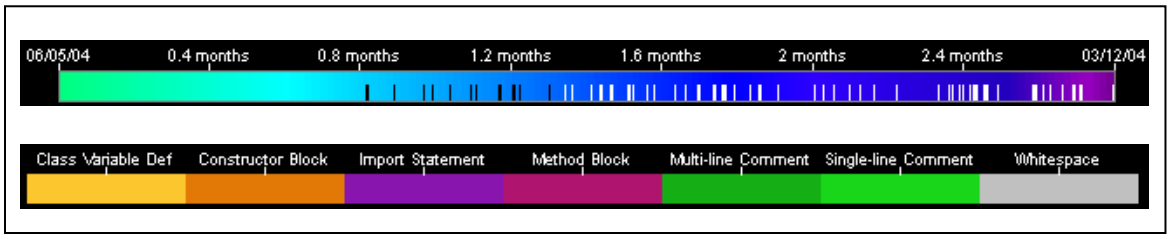

Figure 3.9 Augur color legends
(top) Example of a continuous color legend for line age; the left side of the legend (in light green) is the most recent date while the right side of the legend (in purple) is the oldest date. (bottom) Example of a discrete color legend for Java structure elements.

The continuous color legend shown in Figure 3.9 displays color mapping information for a user-defined date range (in this case, June 5th, 2004 to March 12th, 2004—or approximately three months). The dates are mapped on a continuous scale with the green to blue to purple gradient where the left side of the legend (in light green) is the most recent and the right side (in purple) is the least recent. The small inline tick marks protruding from the bottom of the legend represent actual file check-in dates. Note how, in this example, there were no check-ins in approximately the past 0.8 months (i.e. there are no tick marks on the left quartile of the legend).

The discrete color legend shown in Figure 3.9 displays color mapping information for Java structure elements. Only comments and whitespace that fall outside method and constructor blocks are highlighted in their respective colors (green and light gray). Legends can also respond to mouse clicks. For example, if the user selects “Constructor Block”, all constructor blocks will be highlighted in the line-oriented file view. So, the legends can serve as both passive information displays and active interaction windows depending on their use.

3.2.6 Filetree Explorer

The filetree explorer allows the user to:

- Understand the line-oriented file panels in the primary visualization window in terms of their folder/file hierarchy
- Constrain information displayed in other visualizations by selecting only one or two folders to view
- Identify the location of recent activity in terms of the file hierarchy

The filetree display looks and functions much like the Explorer view in Microsoft Windows. The file and folder icons are arranged in a hierarchical fashion according to their arrangement on the configuration management server. Files that have been parsed and include information about their structural make-up can be expanded to reveal their inner-structure (e.g. a list of methods—see Figure 3.10). The icons themselves are colored by the current display mode. For example, if the current display mode is "Color by Author," then the file and folder icons are colored by the most recent author in the given file or folder (this allows users to, at a glance, observe who has been working where recently).

The filetree also responds to user interaction. Multiple files or folders can be selected at a time. For example, when a file is selected in the filetree (by left mouse clicking on the file icon), its corresponding line-oriented file view is located and highlighted in the primary visualization window. Other visualizations may also respond by displaying different sorts of information on the selected file (e.g. change history, file growth, etc.). Similarly, when a folder is selected in the filetree, all visualizations respond by constraining their displays to only reveal information relevant to that folder (e.g. the

primary visualization window will only display files in the selected folder and sub-folders). This allows users to investigate patterns at different levels in the file/folder hierarchy.

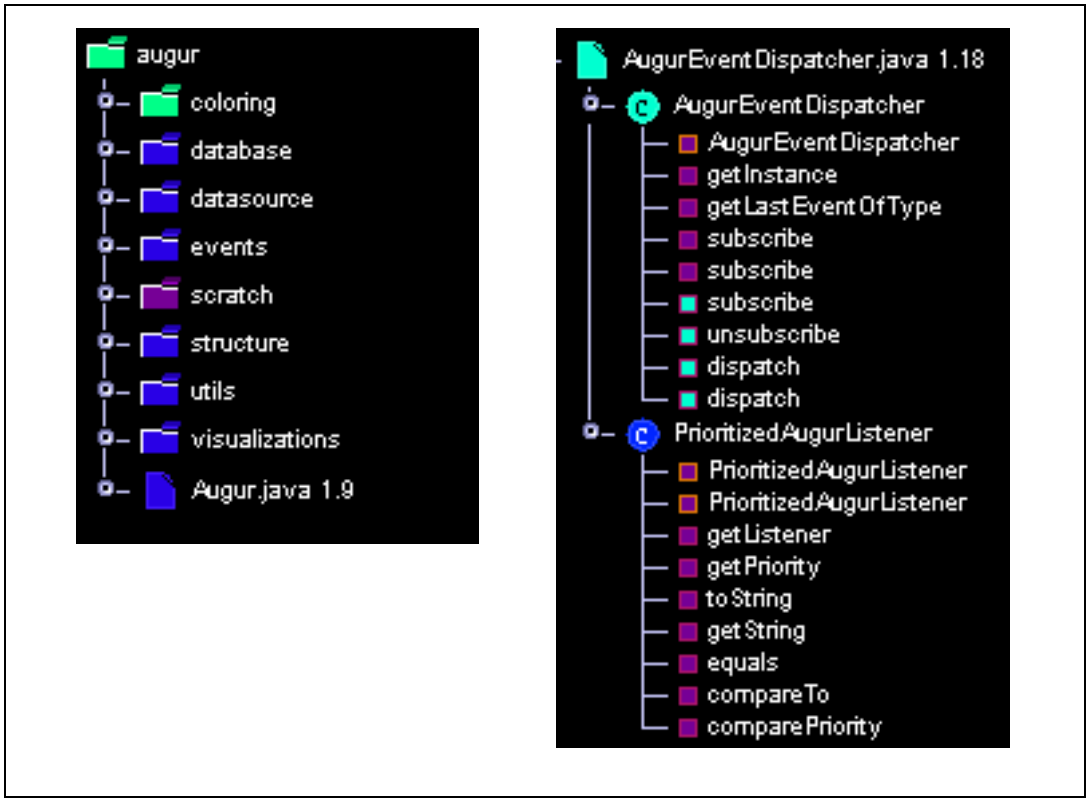

Figure 3.10 The filetree explorer
(left) Folder view colored by age. From this view we can easily discern that, for example, the folder coloring has had recent activity while the folder scratch has not. (right) File view colored by age. The expanded file, AugurEventDispatcher.java, contains two classes: AugurEventDispatcher and PrioritizedAugurListener (denoted by circles with the letter 'C'). The class AugurEventDispatcher has been updated recently—four of its methods were modified. The subtle orange and maroon outlines in the structure block squares represent constructor and method respectively.

Figure 3.10 demonstrates two views of the Augur filetree. In both views, the display mode is "Color by Age" such that the icons are colored by their most recent modification date. The left figure shows the unexpanded filetree view; note the spectrum of colors ranging from green to purple in the folder icons. The coloring folder (in green) contains one or more files with recent modifications. Conversely, the files in the scratch folder (in

purple) have not been modified in a while. This high level information allows the user to quickly ascertain where the recent activity is concentrated in the codebase and drill-down by selecting folders of interest. Thus, the colored icons serve as sort of an overview map describing where recent activity has occurred.

The right view in Figure 3.10 illustrates the low-level structural view available in the filetree explorer. Here we see that the file, AugurEventDispatcher.java, has been modified recently (its file icon is colored green). More specifically, we can see that the AugurEventDispatcher.java file contains two classes: AugurEventDispatcher and PrioritizedAugurListener (denoted by the two circular icons with the letter 'C'). We can also determine that the AugurEventDispatcher.java file icon is colored in green because four of its methods were modified recently (subscribe, unsubscribe, and two dispatch methods). To investigate these modifications in detail, the user can select either the file, the class, or the method icons in the filetree and the appropriate section of code will be highlighted in the primary visualization window.

3.2.7 Secondary Visualizations

The secondary visualizations are located in three window panes at the bottom of the Augur interface. They are meant to provide interesting views and abstractions of the data that are not available through the primary visualization window. Each visualization must react both to user-interactions and to the Augur event system such that a date selected in one, for example, is also selected in the others (see Section 4.3.3). This, along with consistent use of color mappings, presents one, seamless interface to the user. Each window pane has a roll-over drop down menu that allows the user to select any of the AugurComponent visualizations in the system (see Section 4.3.1).

There are two notable aspects of the Augur secondary visualization system that should be mentioned. The first is extensibility; new secondary visualizations can be created and "dropped-in" to Augur without much development effort (see Section 4.3.1). This extensible design allowed us, as developers, to quickly prototype and experiment with new visualizations while building Augur. The true benefit, however, in providing this extensibility is supporting third-party development (i.e. so that programmers can develop and integrate their own custom visualizations). The second notable feature of the Augur secondary visualization system is in its relationship to the line-oriented file view in the primary visualization window. Secondary visualizations often portray completely unique views of the data and/or utilize higher-level visual abstractions than the primary visualization window. These views are all linked through the Augur event system. A user investigating an interesting pattern in a secondary visualization is able to see how this pattern relates to the source code itself simply by interacting with the system. For example, in a line graph depicting project growth, a user is able to select a "check-in" point and instantly see all the code that was checked-in at that particular time in the primary visualization window.

Approximately twenty secondary visualizations have been developed for Augur. The following section provides a brief overview of a selected subset of these secondary visualizations.

3.2.7.1 Project Growth

A number of Augur secondary visualizations are simply traditional quantitative displays like line graphs, bar graphs, and histograms. These displays carry the benefit of familiarity for the user and, therefore, require very little acclimation to understand them.

In addition, they are easy to generate through the use of the open source JFreeChart[6] graphing library subsystem. Augur attempts to leverage graphing libraries whenever possible (Augur uses Prefuse[7] and JUNG[8] as well). The chart shown in Figure 3.11 is a line-graph oriented display depicting project growth.

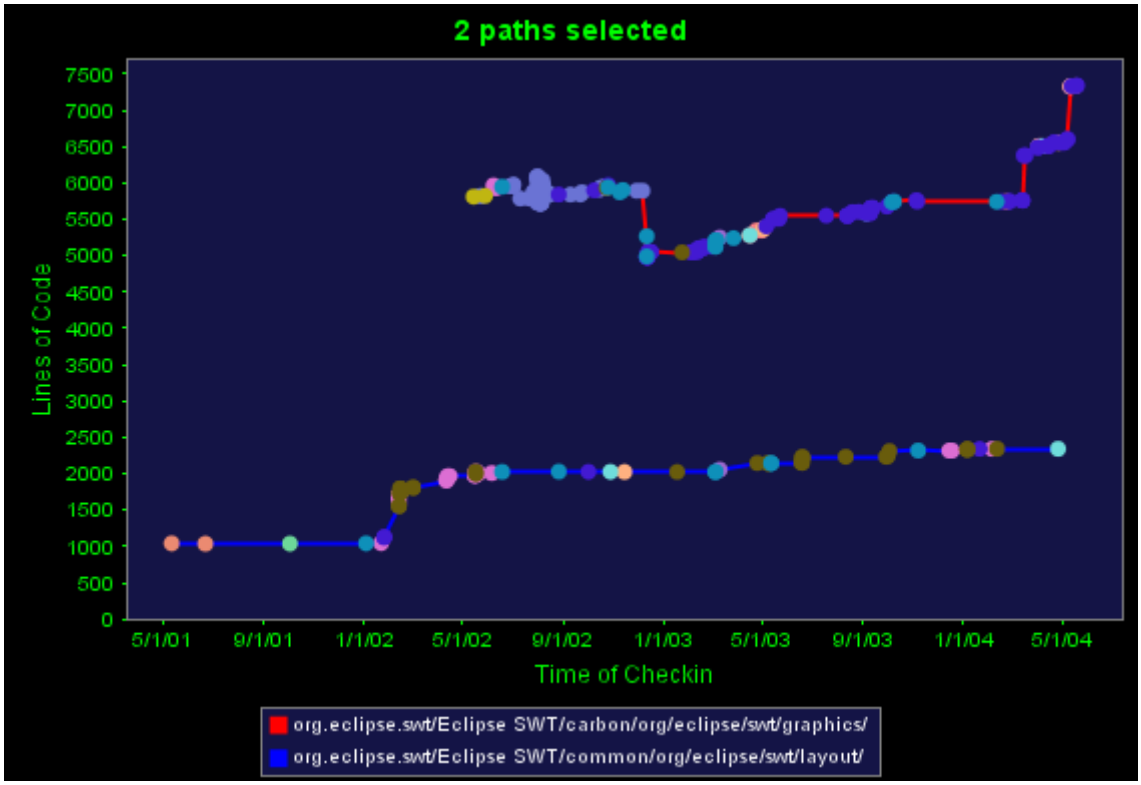

Figure 3.11 Line graph depicting project growth
Two source folders are selected: the graphics folder (red) and the layout folder (blue). The lines represent folder growth over time; the dots represent actual check-in times; the dot colors represent the author of that check-in.

In this particular case, the graph displays "number of lines of code" data for two separate folders selected in the filetree explorer view (see Section 3.2.6). There is no limit on the number of paths that can be viewed simultaneously. Files and paths can be selected and viewed together, which allows the user to compare how those elements have

[6] http://www.jfree.org
[7] http://prefuse.sourceforge.net
[8] http://jung.sourceforge.net

evolved. The lines themselves are colored by path or file and the dots represent check-in times. The dot color represents author (which, again, is used consistently in each visualization)

In Figure 3.11, for example, we can ascertain that the "layout" Eclipse directory (in blue) began nearly one year before the "graphics" Eclipse directory (in red). Also, the Brown author appears to be a frequent contributor in the layout directory, while the Dark Blue author appears to be the primary contributor in the graphics directory (at least in the past year: 05/01/03 – 05/01/04). The layout directory has seen relatively steady growth from the onset while the graphics directory has fluctuated more seriously (and appears to be growing in large spurts in the past few months).

This graph is also interactive: a line dot can be clicked-on by the user. This sends a DateSelectedEvent (see Section 4.3.3) to the other visualizations allowing the user to investigate how a line-graph pattern relates to a specific check-in in the codebase.

3.2.7.2 Commit Log

Although most of the visualizations in Augur have very little text, the commit log visualization is text-based. It allows the user to browse the configuration management commit log for any of the check-ins in the given project. This visualization was developed as a result of our preliminary investigations with Augur (see Section 5.1). The users wanted a way to view the commit log in response to their interactions with the codebase. When a check-in date is selected in the system (e.g. by clicking on a line), the commit log details for that date and two dates before and after it are displayed in a scrollable pane.

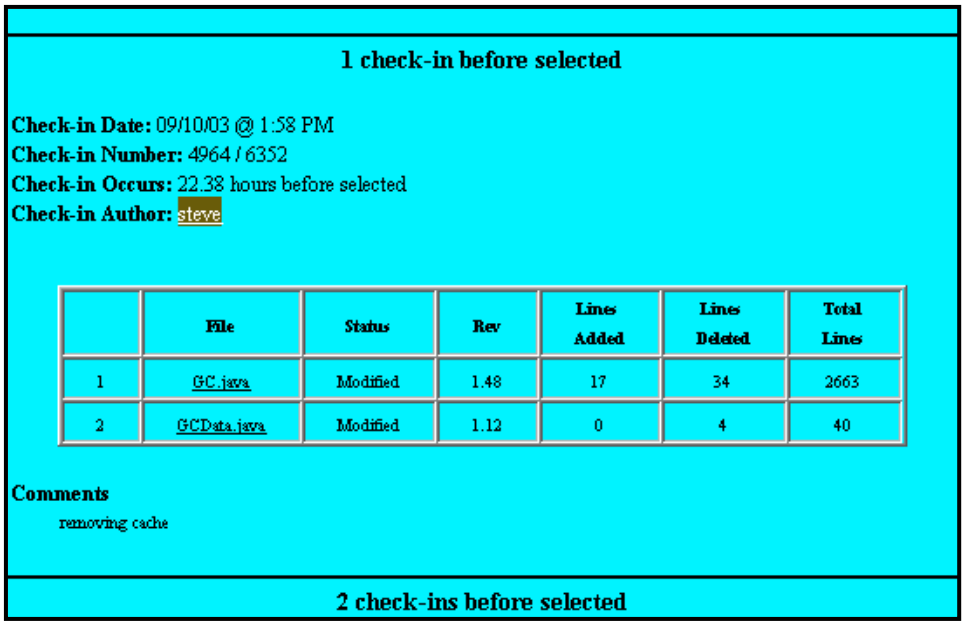

Figure 3.12 Commit log relating check-in information
This commit log visualization reveals information about a specific check-in. It shows the check-in date and time, the check-in number, author, comments, and files modified, added or deleted.

The commit log visualization reveals check-in information in a html formatted window. The check-in date, check-in author, and file names are all clickable. The background color of the check-in pane depends on the check-in date (uses the same color mapping as line age: green to blue to purple).

Figure 3.12 demonstrates the commit log view. The background of each check-in pane is colored by date. The check-in date, check-in author, and filenames are all selectable via the left-mouse button.

3.2.7.3 Temporal Patterns

The temporal related visualizations focus on revealing interesting time related activity dynamics in the codebase to the user. The weekly calendar activity diagram (or "actogram") shown in Figure 3.13 and the developer activity histogram shown in Figure

3.14 provide different views of the temporal data. The weekly calendar view shows the relationship between an author's check-in activity and time of day. The developer activity histogram reveals check-in activity patterns in a linear time view. For example, if the user manipulates the time window to display the past 24 hours, the weekly calendar view would show which authors are active in that time period relating to time and day (e.g. Jim has checked-in code twice in the past 24 hours: once at 11AM and once at 4:30PM and Andrea has checked-in code once in the past 24 hours, at 2PM). If the user then selected the past 12 months, the developer activity histogram could be used to show spikes in activity (e.g. when the developers were preparing for a release).

Both of these temporal visualizations respond to Augur events. For example, if a DateSelectedEvent is received, that date is highlighted in the view. Similarly, if an AuthorSelectedEvent is received, all check-ins submitted by that author are highlighted. For an example of how a display might change after receiving an AuthorSelectedEvent, see the bottom views in Figure 3.14. For more one Augur events, see Section 4.3.3 .

Figure 3.13 presents two weekly calendar activity diagrams from two different open source projects. The top figure is visualizing data from a commercially backed open source project; the developers are paid employees. The bottom figure is visualizing data from a highly successful sourceforge.net hosted open source project (no commercial sponsors). The time window used in both figures is set to maximum, which displays temporal information for the entire lifespan of the project.

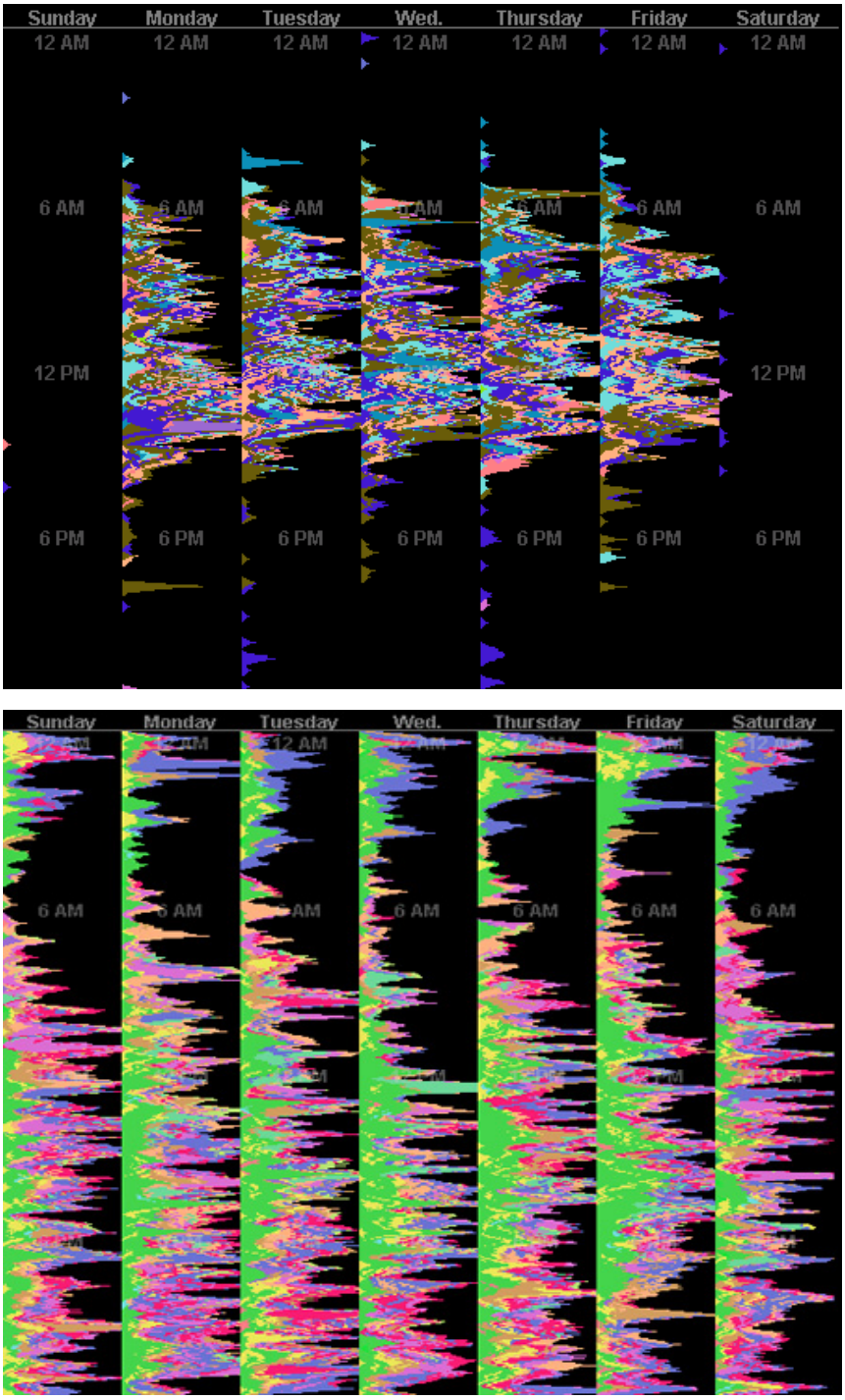

Figure 3.13 Weekly calendar activity diagram
This visualization shows check-in activity by week; in both the top and bottom views, the entire project's time span is visualized. The colors correspond to authors. (top) Note very little activity in the top view outside of 6AM to 5PM weekdays. (bottom) Conversely, this view has fairly steady activity (even on late nights and weekends). This view allows the user to see who has been working and when.

Notice, first of all, that the temporal patterns in the top view differ significantly from those in the bottom view. The majority of check-ins in the top project occur between 6AM and 5PM on weekdays. In comparison, the bottom project has no strong check-in activity distinctions between weekday and weekend. Even more striking, the developers appear to check-in code at all hours of the day and night. In the top project, if there is check-in activity after 7PM on the weekdays or at anytime during the weekends, it is most likely done by the Blue author. No such observation is possible with the bottom project.

These temporal views are meant to increase developer awareness of their colleague's activity such that they are better able to answer questions like "was Kerry able to finish and check-in that new code yet?" At a glance, the user can observe whether or not Kerry has been active recently and, if she was, the user can click on that check-in date in the temporal view and the code that Kerry modified or added will be highlighted in the primary visualization window.

Figure 3.14 displays an instance of the developer activity histogram.The time window for the project being viewed in Figure 3.14 is set to its maximum extent (that is, the time window is set for the entire lifespan of the project). In this case, that means viewing about a two year long time frame (from May 2002 to May 2004). The top view titled "Num Check-ins by All Authors" combines all author activity data into one display. This allows the user to see broad "group" fluctuations in activity over time. In this case, the two obvious spikes in activity correspond to the time just preceding a release of the software (that of April 2003 and May 2004).

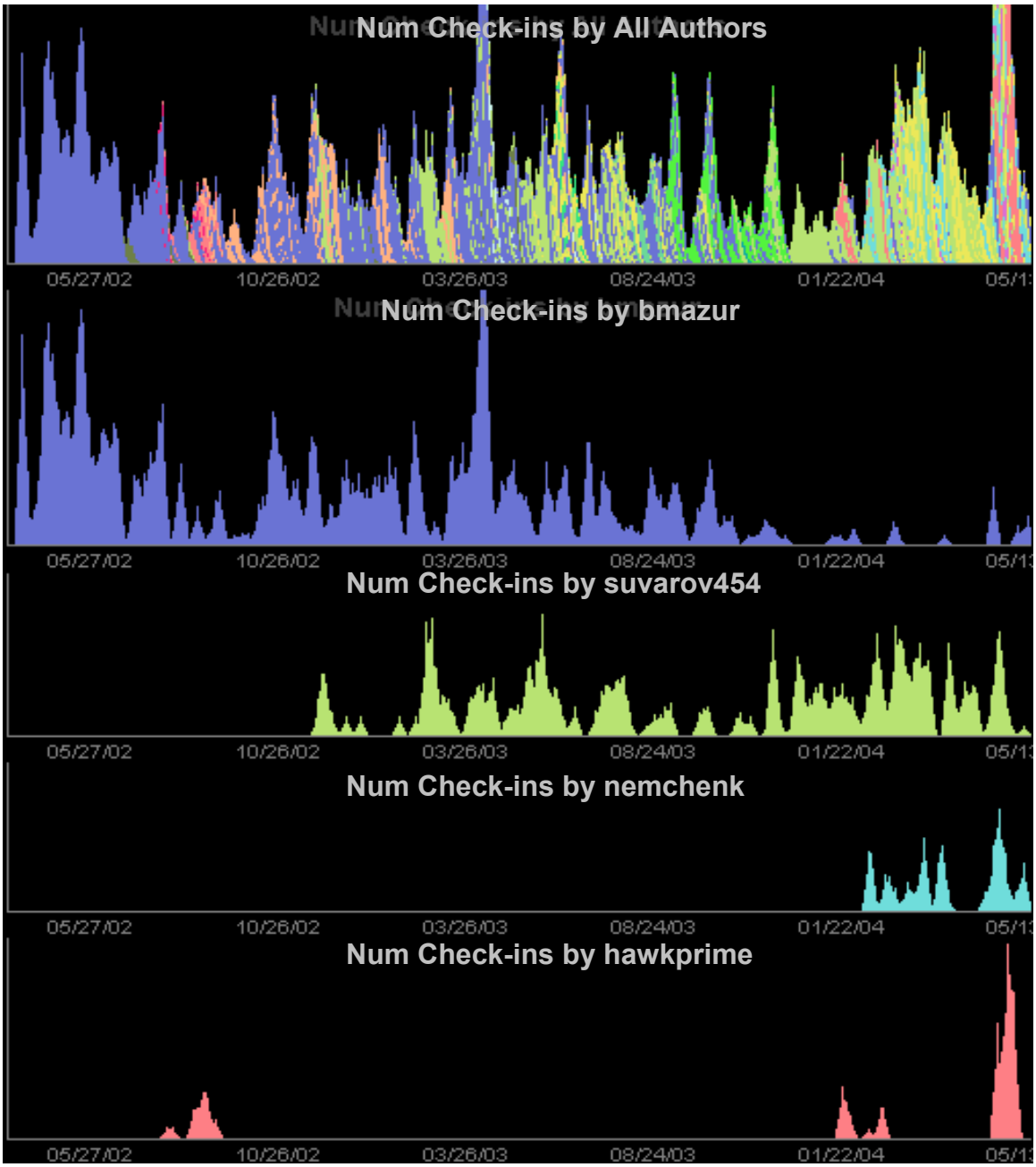

Figure 3.14 Developer activity histogram
This is a horizontal histogram-like view of the check-in activity data for the selected date range (which, in this example, corresponds to the entire time span of the project). Like in Figure 3.13, the colors correspond to authors and the height to check-in activity. The top view "Num Check-ins by All Authors" shows combined activity data for all authors, the other four views are for individually selected authors. Note that from this graph it is clear who started the open source project (bmazur), who comes and goes (hawkprime) and who is just beginning work (nemchenk). This view allows the user to see who has been working and when.

The bottom four views in Figure 3.14 display check-in activity of only one author at a time. This allows the user to compare the check-in activity of all the developers in the project. For example, we can easily determine that the author "bmazur" was the creator of this software (owns the first set of check-ins to the project). We can also see that "suvarov454" began working on the project about six months after it was introduced and that "nemchenk" is a relatively new contributor to the project with only about four months of check-in activity. Most interestingly, we can discern that the author "hawkprime" seems to have a highly sporadic contribution pattern. Hawkprime's first spurt of activity occurs around June of 2002; however, it is not until January of 2004 until another check-in is made (an absence of nearly 18 months). He again disappears in February sometime only to check-in code about three months later in May 2004.

These temporal views allow developers to relate interesting patterns of check-in activity (e.g. activity bursts) to major events in the development process (e.g. refactorings, branchings, releases, etc.). They can select an interesting date in these temporal views and explore the activity more closely in other visualizations (e.g. they can look at the commit log to see the comments about the activities or they can look at the source code itself that was modified during this time in the primary visualization window). Finally, these views are helpful in relating who is currently contributing source code. This is particularly helpful in open source projects as developers often come and go (like "hawkprime" in Figure 3.14). New developers still adjusting to a project can use these views along with the primary visualization window to gain a better understanding of who did what and when.

3.2.7.4 Social Patterns

We have been exploring visualizations that reveal relationships between authors based on their activity in the source code. Figure 3.15 is one such visualization. It ties authors together based on file modification data; if two authors modify the same file, an edge is drawn between them. The more files they work on "together", the thicker the edge. This "social network" view is time sensitive. Just like in the temporal views, the user can manipulate the time window to affect the display of the graph. For example, if the user selects the "past 7 days", the graph will only display author vertices and connections that have been active in that time period. Figure 3.15 has its time window maximized, such that the time period used to display this visualization includes the entire lifespan of the project. By reacting to selectable date ranges, this visualization reveals how relationships between authors change over time.

The vertex size is based on the amount of check-in activity and its color is the assigned author color. Each vertex is outlined by a date color representing the last time they were active in the system. Edges are also colored based on time; the color represents the most recent modification date by either author in a file touched by both.

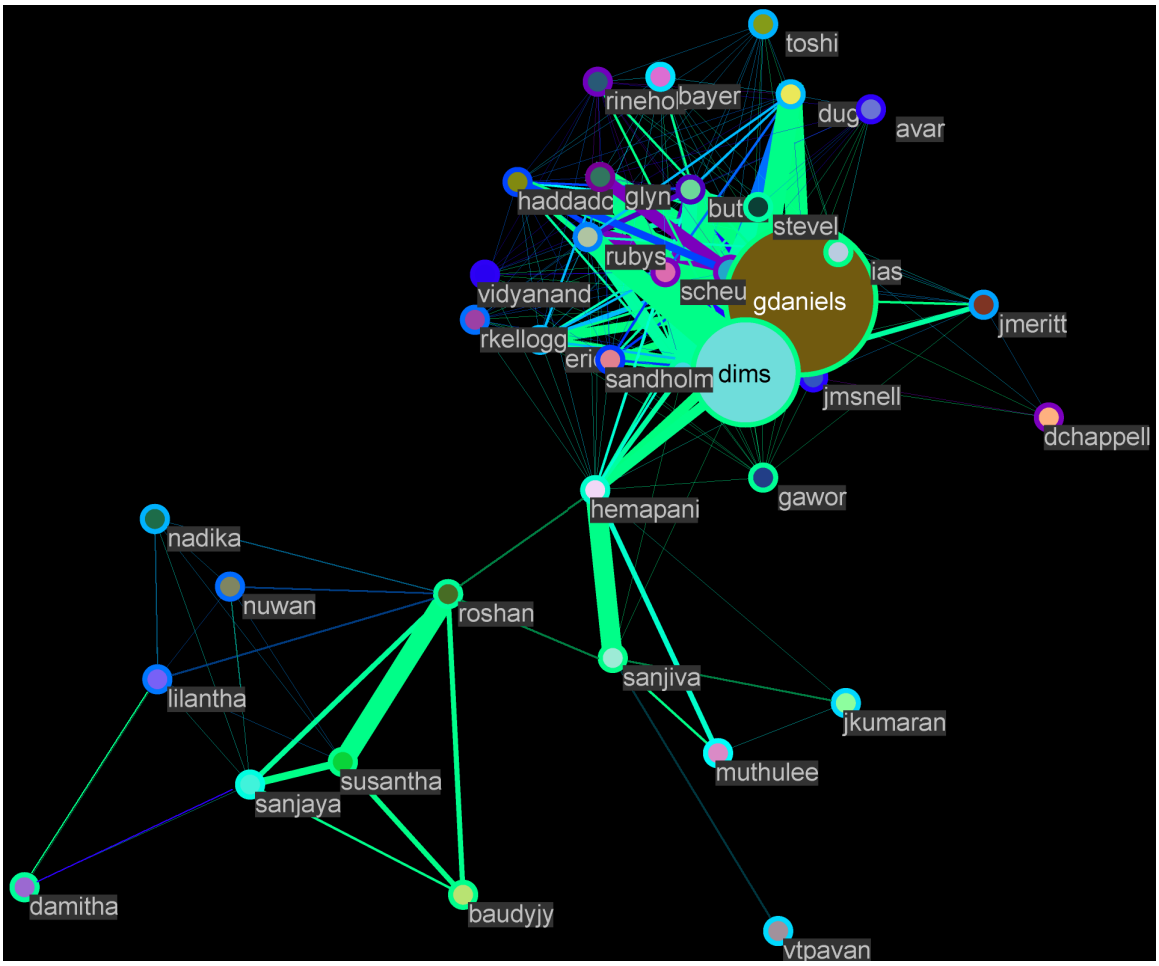

Figure 3.15 Author network diagram
This is a social network view of the authors in the project. In this particular example, the entire timeline of the project is select. Vertices are authors; vertex size corresponds to the number of check-ins by that author; vertex color is colored by author; edges between two authors represent that they have modified one or more of the same files; edge thickness is how many files; edge color represents the most recent date of activity between them. This view is sensitive to the date range selected and allows the user to see who has been working together within that date range (e.g. past 24 hours or, like in this view, the entire timeline of project)

Figure 3.15 has a particularly interesting, if not busy, structure. The graph would not look nearly so complicated if a smaller time frame was selected (e.g. past 2 days, past 1 week, past 1 month, etc.). However, by looking at the entire project lifespan, we are able to gain some additional insights. For example, there are clearly three clusters of

developers here: the primary cluster around the principal authors “gdaniels” and “dims” and the two secondary clusters bridged by authors “hemapani,” “roshan,” and “sanjiva” Though most of the development involves files that “gdaniels” and “dims” have touched, there is a contingent of authors (apprxoimately 11 developers) who always work elsewhere. So, the author network diagrams begin to reflect the modularity in the source code and how different groups of authors work on different subsystems. These graphs also reveal who the “bridge authors” are; those developers that tie the groups together.

In addition to this graph, we have been experimenting with a variety of social networking and structural analysis techniques to reveal different sorts of group structures and relationships. For example, visualizing social dependencies based on the call-graph dependency analysis, class hierarchy structures, and interface implementations. This is an active part of our ongoing work (see 6.1.2).

3.2.7.5 Social and Temporal Patterns

These secondary visualizations are not viewed in isolation or individually, but together in a combined view provided by the Augur user-interface. This allows the user to investigate interesting patterns in one visualization and relate it to another. For example, as explained in the previous section, the author network diagram views are time dependent. By including the temporal related graphs (Section 3.2.7.3) alongside the author network graphs, these time dependencies are further exposed.

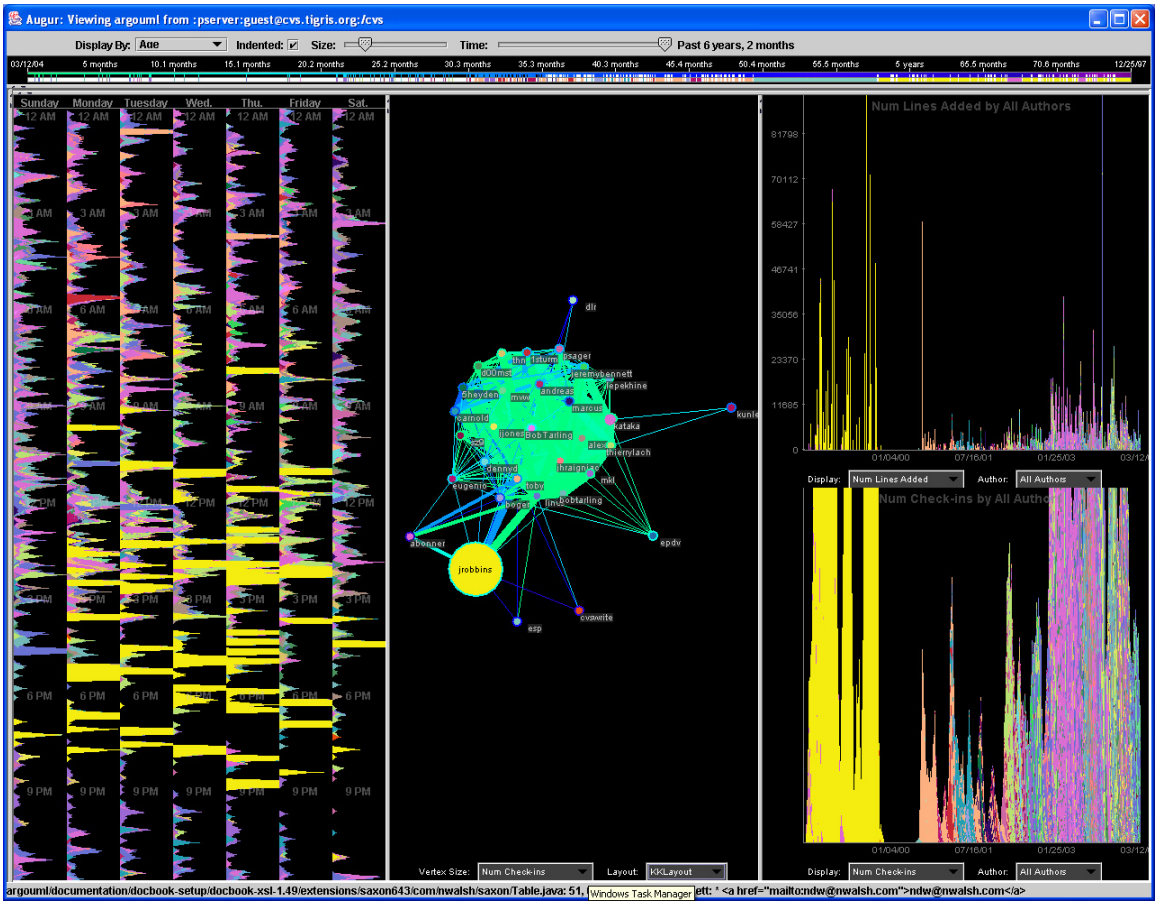

Figure 3.16 Combining social and temporal views
This figure illustrates the three secondary visualization panes maximized in the Augur application. Here we are seeing a combined view of the temporal displays and the author network diagram. By displaying these views together, the user is able to better understand who has been working with who and when.

For example, in Figure 3.16, we see two temporal views and an author network view. If we focus on the author network graph, we see that the Yellow author is only connected to a few other developers. This is unexpected. Typically, a chief contributor of source code (reflected by vertex size) is part of the central core of development and is, therefore, connected to many authors (as in Figure 3.15). The temporal views provide us with an explanation. The right temporal view allows us to discern that the Yellow author is not only a primary contributor, but also the originator of the project itself. The Yellow author

appears to have worked vigorously for the first year of development and then abruptly stopped. Since that time, only a few authors have worked on the original codebase developed by the Yellow author—all the other authors have expanded functionality elsewhere. The Yellow author is not part of the current core development team because he has not contributed code in over four and a half years.

The basis for these interesting visual patters can be explored by opening the primary visualization window and looking at the source code itself.

3.2.7.6 File Check-in Patterns

Finally, the last secondary visualization highlighted in this thesis is the file check-in activity network. This view allows the user to see how files are related through check-in activity. For example, if every time file A is modified, file B is also modified, there could potentially be some sort of functional dependency between them. This view allows the user to compare file check-in activity relationships with the structure of the system architecture. This evolutionary coupling is also explored in [Bieman et al. 2003] and [Boehm and Bose 1994].

Figure 3.17 demonstrates an example of the file check-in activity network. An edge is drawn between two files, if those files have been modified at least N times together (this is settable via a slider control by the user). In this case, N is set to one. So, in Figure 3.17, an edge is drawn between two files if they have been checked-in together at least one time. The edge thickness represents how many times the files were checked-in together and, finally, the edge color represents the most recent date in which both those files were checked-in together.

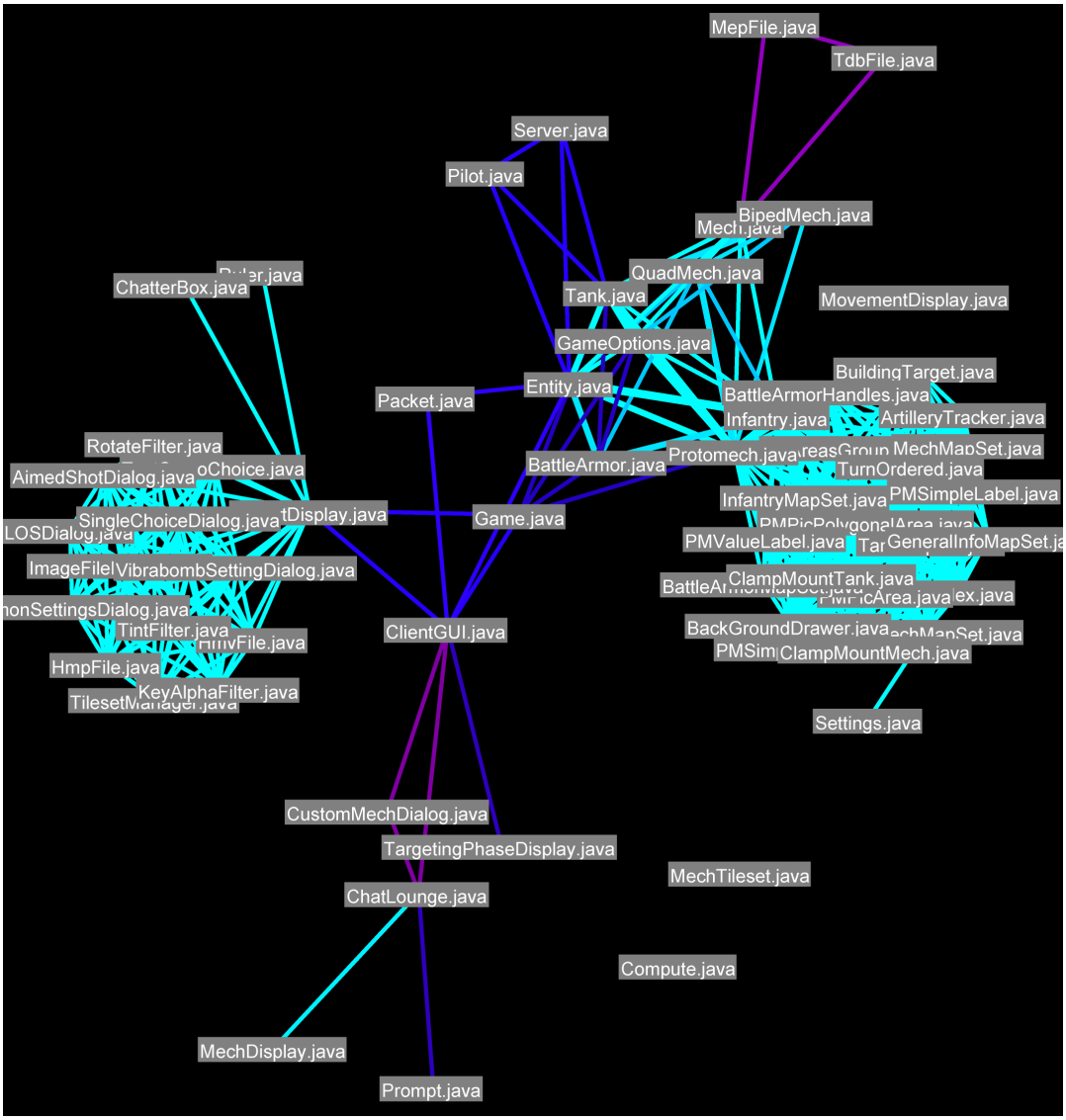

Figure 3.17 File check-in activity network
This view shows which files have been checked in together. The vertices are files; an edge between two vertices represents that they have been checked in together; the edge thickness represents the number of times the files have been checked in together; the edge color represents the most recent date the files were checked in together. Typically this view allows the user to see how functionality (modular dependencies) correspond to activity (i.e. "is a file both functionally related and also related by check-in activity?")

This visualization is still in an experimental phase. Eventually, we hope to incorporate authors and files into a bipartite network graph to reveal more information about the concentration of author activity in the source code.

Just as in all other visualizations, this graph responds to Augur events so that, for example, a user can click on an edge in this graph and see all the lines of code that refer to that check-in in the line-oriented view. Emphasis is placed on relating the secondary visualizations back to the source code.

3.3 Augur Interface and Interaction Conclusion

The central feature that Augur uses to tie together multiple forms of information is the spatial organization of the source code in the primary visualization window. This spatial arrangement is familiar to all developers and common across different individual perspectives. Providing a unifying framework of this sort allows different types of information to be synergistically combined; it supports rapid movement back and forth and the simultaneous combination of information (e.g. through the primary and secondary attributes in the line-oriented view.) While Augur extends this spatially-oriented view with secondary graph depictions of cumulative statistics about source code and activity, it is the common frame presented by the spatial arrangement of the code that ties everything together.

Chapter 4 ARCHITECTURE AND IMPLEMENTATION

Augur's architecture is best understood in terms of its two primary architectural components: the backend, which centers on data retrieval, analysis, and storage, and the frontend, which consists of the Augur event system and the visualizations framework. Figure 4.1 below provides a high-level diagram view of the architecture.

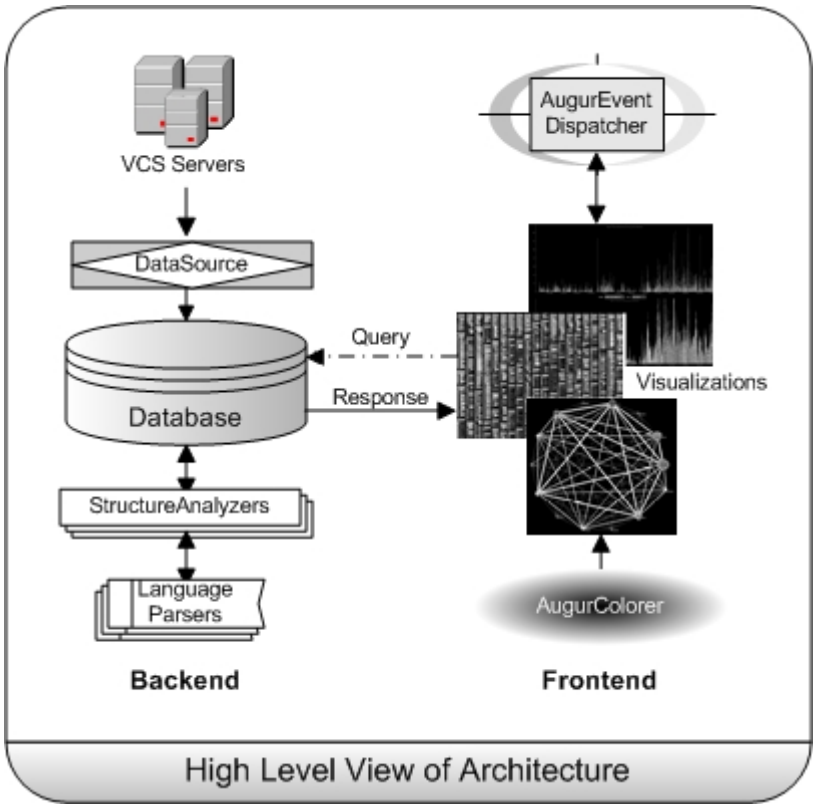

Figure 4.1 A high level diagram of the Augur architecture
The backend handles configuration management connections, data retrieval and analysis, and populating and managing the Augur database. The frontend consists of the event system, the colorization system, the visualizations framework, and the visualizations themselves.

A fundamental criterion in designing Augur was that it be flexible enough to accommodate different types of development situations, scenarios, and technologies. Emphasis was placed on designing an extensible and flexible architecture with a broad

use of interfaces that provide a standardized extensible framework in both the back and frontend architectures. In addition, a flexible, partially open-ended query system was developed for the database, allowing database clients (e.g. visualizations) to dynamically build their own queries.

As a whole, Augur supports four particular forms of extensibility: in repository protocols, in analytic tools, in visual displays, and in custom queries. These elements provide a framework to extend support for multiple configuration management systems, programming languages, visualizations, and novel data queries respectively.

4.1 BASIC DATA FLOW

The following three steps provide a walkthrough of the basic data flow in Augur.

4.1.1 Step 1: Server Connection and Data Download

Upon program startup, a version control system connection dialog is displayed to the user. The user enters his/her connection parameters (e.g.. host, project name, login credentials) and begins a data connection to the configuration management[9] server. Augur then initiates multiple threaded input/output streams to download files, file annotations, and file logs for the selected project (see

Figure 4.2). Only out-of-date data is downloaded from the server—the rest is retrieved locally from a disk cache (as provided by the Augur local caching service).

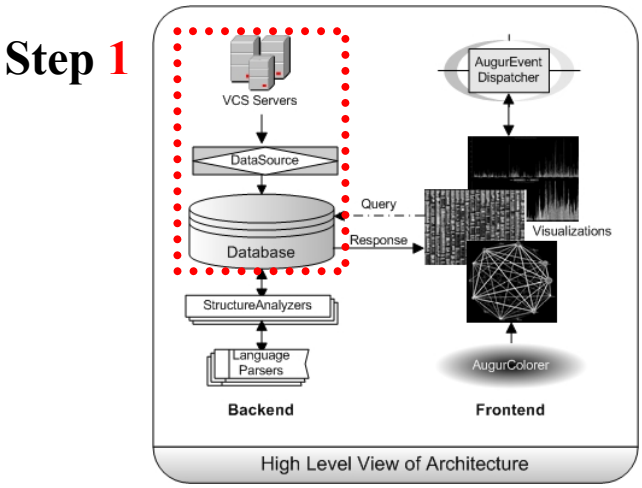

Figure 4.2 Step 1: Server connection and data download
First, a network connection is made to a CM server. Project data is retrieved and loaded into the database. A wide range of CM systems can be supported through implementation of the DataSource interface.

[9] "Version control systems" and "configuration management systems" are, in this paper, used interchangeably. Configuration management is sometimes abbreviated as CM.

4.1.2 Step 2: Data Analysis

The loaded files are then analyzed by a structure analyzer class, which controls and facilitates any number of structural analysis procedures (see

Figure 4.3). One primary means of structural analysis is language parsing. If the structure analyzer determines that a language parser exists for a given file type (e.g. a Java language parser exists for a Java file), the file is parsed and its structural data is extracted and recorded. If no language parser exists, this step is skipped.

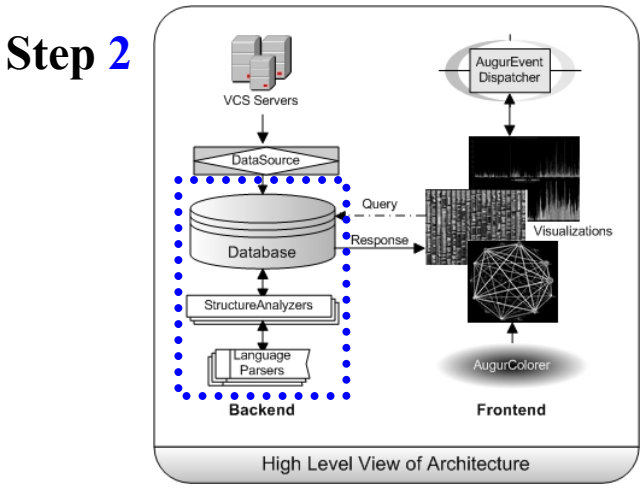

Figure 4.3 Step 2: Data analysis
Then, the source code passes through a series of structural analyses as dictated by the StructureAnalyzer class. New language parsers can be registered with the StructureAnalyzer's addLanguageParser method.

4.1.3 Step 3: Visualization System Displayed

After the structural analysis completes, Augur initializes the visualization framework and displays the main Augur window. Visualizations retrieve data to display using the Augur Evaluator Query system. Certain data units (e.g. authors, dates) are colored consistently throughout the application; this is managed by the AugurColorer. The rest of the data flow is shaped by user interactions with the interface. These interactions produce events handled by the Java Abstract Window Toolkit (AWT) event thread, and the Augur event handling system.

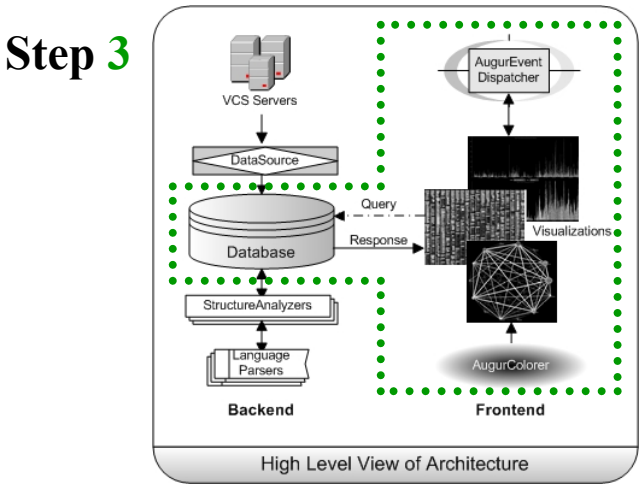

Figure 4.4 Step 3: Visualization system is displayed
After the structural analysis completes, the visualization system is initialized and displayed. New AugurEvents and AugurComponent visualizations can be easily added (see Section 4.3.1 and 4.3.3). From this point on, data flow is dependent on user interactions with the interface.

The remainder of this chapter looks more closely at the backend and frontend architectures.

4.2 BACKEND ARCHITECTURE

The backend architecture describes a layer of classes built to retrieve, analyze, and store CM data. This is broken down into three main components: the DataSource, the StructureAnalyzer, and the VersionDatabase (see Table 3.1Table 4.1). A graphical depiction of the backend architecture is shown in Figure 4.5 (this is a more detailed view of the left-side of Figure 4.1).

Table 4.1 Three Main Components of the Backend Architecture

Section	Component	Extensible	Responsibility
4.2.1	DataSource	Yes	-Configuration management connections -Data retrieval -Caching CM data locally (via CachingService)
4.2.2	StructureAnalyzer	Yes	-Parsing source code -Building Abstract Syntax Trees (AST) -Building StructureBlock Trees (SBT) -Analyzing source code dependencies
4.2.3	VersionDatabase	Yes (query system only)	-Data storage -Database query system -Query cache

The DataSource provides a standard interface for downloading data from version control systems. Currently, CVS is the primary version control system supported by Augur; however, a proof-of-concept Subversion implementation has been developed as well. The StructureAnalyzer mediates the structural and semantic analysis of the source

code. It is designed to support a number of different source code analysis techniques, of which language parsing is the most basic. Finally, the central architectural storage component, the VersionDatabase, provides a hierarchical storage perspective of CM data and a standardized query scheme for flexible data access.

4.2.1 The DataSource

Augur defines a DataSource layer, which provides a set of streamlined interfaces for downloading configuration management data. The DataSource interface separates the download phase from the database loading phase, allowing implementing classes to develop their own CM communication schemes. This flexibility is necessary because configuration management systems often differ in their internal storage models and data access protocols. For example, CVS stores commit logs at the file level while Subversion stores them at the project level. As a consequence, Augur downloads N commit logs for N files with CVS, but only 1 commit log for N files with Subversion.

Supporting common network-based configuration management systems (such as CVS or SubVersion) for its data enables Augur to be easily incorporated into existing projects. In other words, Augur requires no additional configuration cost if a project is already using a supported configuration management system. If the CM system is not local to the machine executing Augur, the DataSource is subject to network delays. For example, a five-to-ten minute overhead can be expected for first-time downloads of projects with 700-1000 files where the CM server is accessed via the internet[10]. Augur mitigates these delays by utilizing multiple download threads and providing a local caching mechanism.

[10] This is dependent, of course, on the local network connection and the server network connection.

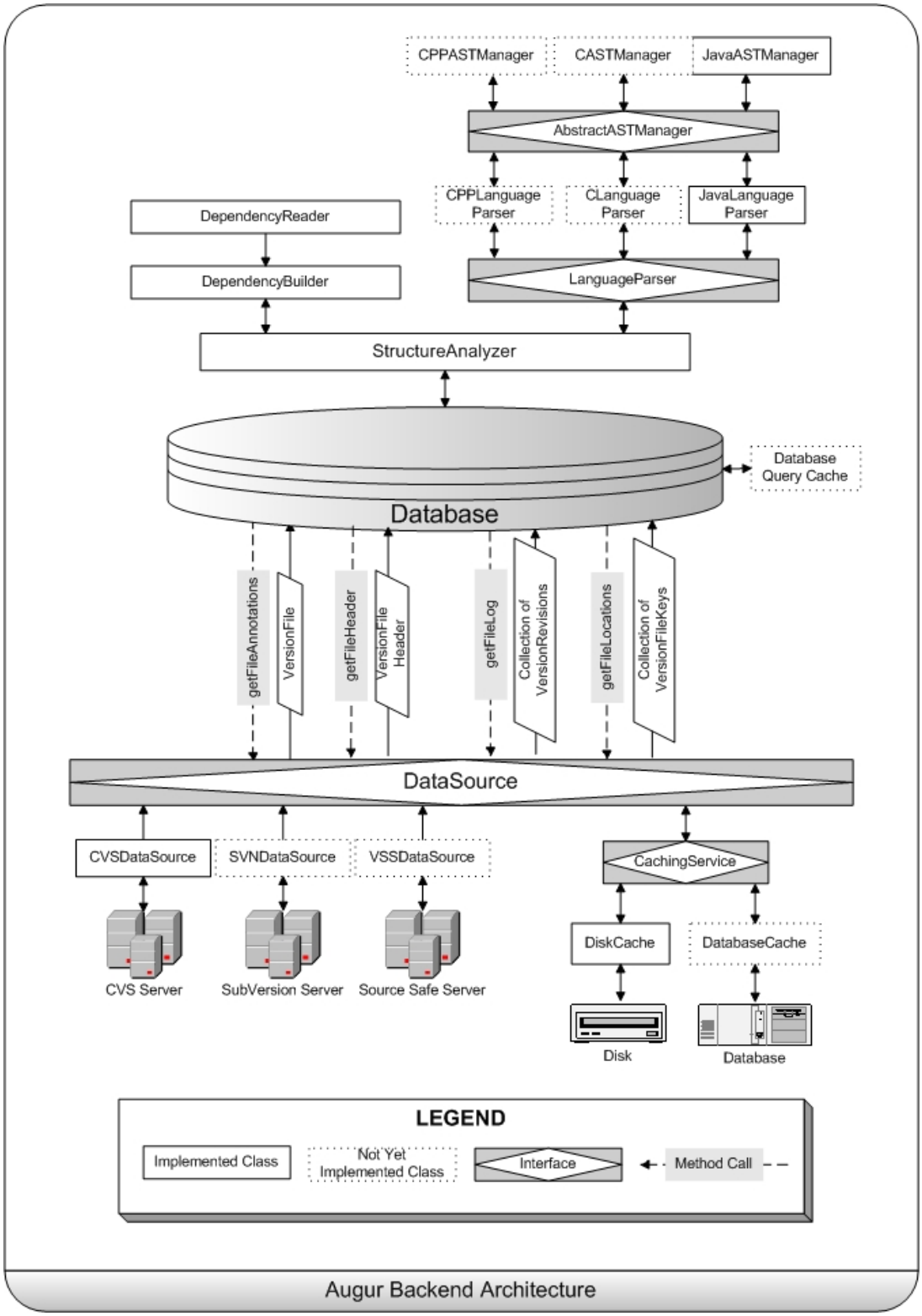

Figure 4.5 A detailed view of Augur's backend architecture

The caching mechanism (CachingService) allows the DataSource to first check a local storage system (file disk, SQL database, etc.) for up-to-date CM data before downloading the out-of-date data from the configuration management server.

4.2.2 The StructureAnalyzer

Once the data has been downloaded by the DataSource, structural analysis of the source code is performed by the StructureAnalyzer component. The most basic structural analysis is language parsing, which includes Abstract Syntax Tree (AST) construction. While this is a very simple form of analysis, it provides a foundation for more rigorous and complex analyses (as most source code analysis requires an AST). Augur delegates language parsing to ANTLR[11], an open source, Java-based parser generator, and so can support multiple programming languages. While our work has primarily focused on Java source code, we have also looked at systems written in other languages in the C family.

Currently, the StructureAnalyzer uses ASTs to generate two sorts of information (which are then recorded as annotations on the LineRecords). The first is line-type data, which distinguishes between different types of source lines (e.g. comment lines, method definition lines, variable definition lines, method call lines, etc.). The second is structure block information, which allows us to relate specific lines to larger structural units (e.g. lines are contained by method blocks, constructor blocks, class blocks, etc.) Structure blocks are valuable because they contextualize line data and provide a hierarchical perspective of the source code. This is used, for example, to determine the extent of specific check-ins (that is, to generalize from the lines that have been modified to the

[11] http://www.antlr.org/

methods that have been modified). For more information on structure blocks, refer to Section 4.2.4 .

The StructureAnalyzer also provides mechanisms to extend analysis of the source code beyond language parsing. For example, Java dependency analysis modules were added to the existing system to analyze static dependencies in Java source code. This allows users to examine interesting dependency relationships in the source code, e.g. “Which modules depend on those recently updated?”, “What authors call my code?”, or “Whose code do I call most often?”). Other approaches could be implemented that reveal different sorts of interrelationships between lines, structure blocks and activity data.

4.2.3 The VersionDatabase

The VersionDatabase stores CM data in hierarchical data structures and provides access to this data via a query/response framework. A key feature of the VersionDatabase is its flexible query system: one that is both multi-modal and multi-level. By “multi-modal,” we mean a query system that can support queries predicated on more than one data element. Similarly, “multi-level” is used to describe a query system that allows queries at different levels of data storage units (e.g. at the line level versus the file level).

For example, a single modal query might be “who are the *authors* in this file” or “what are the three newest check-in *dates* in this file." Both of these queries consist of a single subject; either “author” or “date,” but not both. (Notice that in this case, a “file” is not a subject. The query system distinguishes between storage units and subjects; they are treated independently). However, a multi-modal query system can support multiple subjects, for example, “who *authored* the three newest check-in *dates* in this file.” Augur defines no upper-bounds on the number of subjects used per query.

Multi-level query support provides even more flexibility by minimizing query unit constraints (where a unit is a line, method, class, file, directory, etc.). Often, a constructed query is relevant at varying levels of storage unit granularity. For example, instead of "who are the authors in this file" we have "who are the authors in this method" or "who are the authors in this directory." Augur allows the generic query, "who are the authors in _____" to be applied to any unit of storage.

Finally, in combination, a multi-modal, multi-level query might look like, "who are the authors in this directory active in the past 2 months" or "which classes implement the Serialization interface that have been modified by the authors Frank, Peter, and Andrea in the past week." Notice how these queries combine subjects and different levels of storage granularity; this can become quite intricate and complex, but it is this complexity that allows for interesting and meaningful queries.

Initial prototypes of Augur were very effective at supporting single mode queries or simple multi-mode queries. The query system was managed by the VersionDatabase and controlled through a series of rigid get retrieval methods—for example, getAuthors(), getFiles(), getFile(filekey). By building the query system around a set of well-defined data retrieval methods, the VersionDatabase query interface was simple and efficient. However, tying our queries directly to the interface (the get method definitions), resulted in a fairly inflexible query system. New queries would require changes to the interface or the creation of new methods; in fact, the number of get methods seemed to grow linearly with the number of new unique queries.

One option would have been to reduce the collection of get methods to one simple key-based get method. For example, get(String key). However, given our complex

query requirements, this would necessitate a sophisticated database query syntax in which the client would use to build a query and the server would parse to build a response. In addition, a single get method would require careful class casting of the result. The Augur Evaluator Query Framework was developed as an alternative solution.

The Augur Evaluator query/response framework provides a set of open-ended query interfaces that create a flexible and dynamic query system (see Section 4.2.5). These interfaces support multi-modal and multi-level queries and, most notably, are shared across all levels of data storage.

4.2.4 Data Storage

Along with the query system, data storage is the most crucial part of the Augur backend architecture. A fundamental problem that Augur must solve is to relate two different views of a program—a spatially-oriented view, focused in particular on the lines of source code that make it up—and a structural view that describes the relationship between its elements. Internally, Augur uses a spatially-based representation as the primary organizing principle.

Line text data is housed in hierarchical data structures (beginning with LineRecords), which are all integrated into the Augur Evaluator Query Framework. Meta-data describing various features of the text (e.g. structure, author, time, etc.) can be added at any level of the storage hierarchy, but exists most prominently at the line level as this is the finest granularity available from most configuration management systems.

The storage hierarchy was developed to reflect the arrangement and context in which lines of text exist in a development project. One simple perspective on the arrangement of text data is that it is ordered by line and organized and stored by file; a collection of files

make-up a project. This three-layered arrangement (line > file > project) was the basis for Augur's initial storage system (see Table 4.2).

Table 4.2 Initial Augur Data Hierarchy (from lowest granularity to highest)		
Layer	**Class**	**Function**
1.	LineRecord	Contains line text and annotations
2.	VersionFile	Contains set of LineRecords within a file
3.	VersionDatabase	Contains set of VersionFiles within a project

This architecture allowed query clients to retrieve data at the line, file, and project level; however, it prohibited queries at the structural (e.g. method or class) and path level. To overcome these limitations, a more finely grained approach was developed. Two additional layers were added to the existing storage hierarchy: the StructureBlock layer and the Directory layer (line > structure block > file > directory > project). Table 4.3 and Figure 4.6 reveal the new hierarchical arrangement of data. (Note: layer 2 and layer 4 are the new layers.)

Table 4.3 Current Augur Data Hierarchy (from lowest granularity to highest)		
Layer	**Class**	**Function**
1.	LineRecord	Contains line text and annotations
2.	StructureBlock	Contains set of sub-StructureBlocks Contains pointer to LineRecords
3.	VersionFile	Contains StructureBlock root Contains set of LineRecords
4.	Directory	Contains set of sub-Directories Contains set of VersionFiles
5.	VersionDatabase	Contains set of Directories

With the StructureBlock layer, line data can be understood by its surrounding context. This context is essential because its corresponds with the ways in which we think about and process lines in documents. For example, while reading this thesis, you do not simply see a collection of lines in a document, but rather a collection of lines that are organized by paragraphs, then by sections, then by chapters, and finally, by the whole document itself. This hierarchical structure lends itself to better information processing and, therefore, comprehension on the part of the reader. There is a similar hierarchical arrangement to source code.

In Java source code, for example, lines of text are organized by method, then by class, then by file. This hierarchical structural arrangement is familiar and useful to developers; it provides a way of thinking about and structuring their development (e.g. it is not that developers think about code simply as sequences of lines, but instead, at higher level functional units like methods, classes, packages, etc.). It is important, then, that this information be available in our source code visualizations (see Section 3.1 and 3.2) as it supports this understanding—the StructureBlock layer makes this possible.

The StructureBlock layer not only provides a structural context for the line data (i.e. the structure block annotation column in Section 3.2.1) but also allows for structure specific queries. For example, “give me all the Java methods that were modified in the past 24 hours.” Obviously, as this query includes a structural reference, an adequate response would be impossible without the StructureBlock layer or, at least, some structural understanding of Java source code.

The Directory layer allows queries to be further bounded by path. For example, given the query above, “give me all the Java methods that were modified in the past 24 hours” could be further specified as “give me all the Java methods that were modified in the past 24 hours, but only in the directory named ‘utils’.”

Both the Directory layer and the StructureBlock layer are tree-based classes as they each have a natural tree hierarchical arrangement. One can navigate both layers by traversing these trees (e.g. with getParent(), getChild(), getLeft(), getRight()).

The StructureBlock layer is dependent on the StructureAnalyzer component, which creates StructureBlocks and a StructureBlock hierarchy (StructureBlock Trees) through language parsing. Without a suitable language parser, no StructureBlocks can be created. This dependency also affects the orientation and format of the StructureBlock layer across programming languages, as different languages have different inherent structures. A C language StructureBlock Tree, for example, would not contain classes or interfaces. Conversely, in Java, methods are always the children of a “Class” StructureBlock.

At a high-level, however, these cross-language structural differences are quite minor. This is particularly true for modern programming languages, which, for the most part, have converged on a fairly similar look and feel (e.g. methods, variable definitions, conditional loops, and comment blocks). This is a result of both shared syntactical elements across languages and shared formatting conventions.

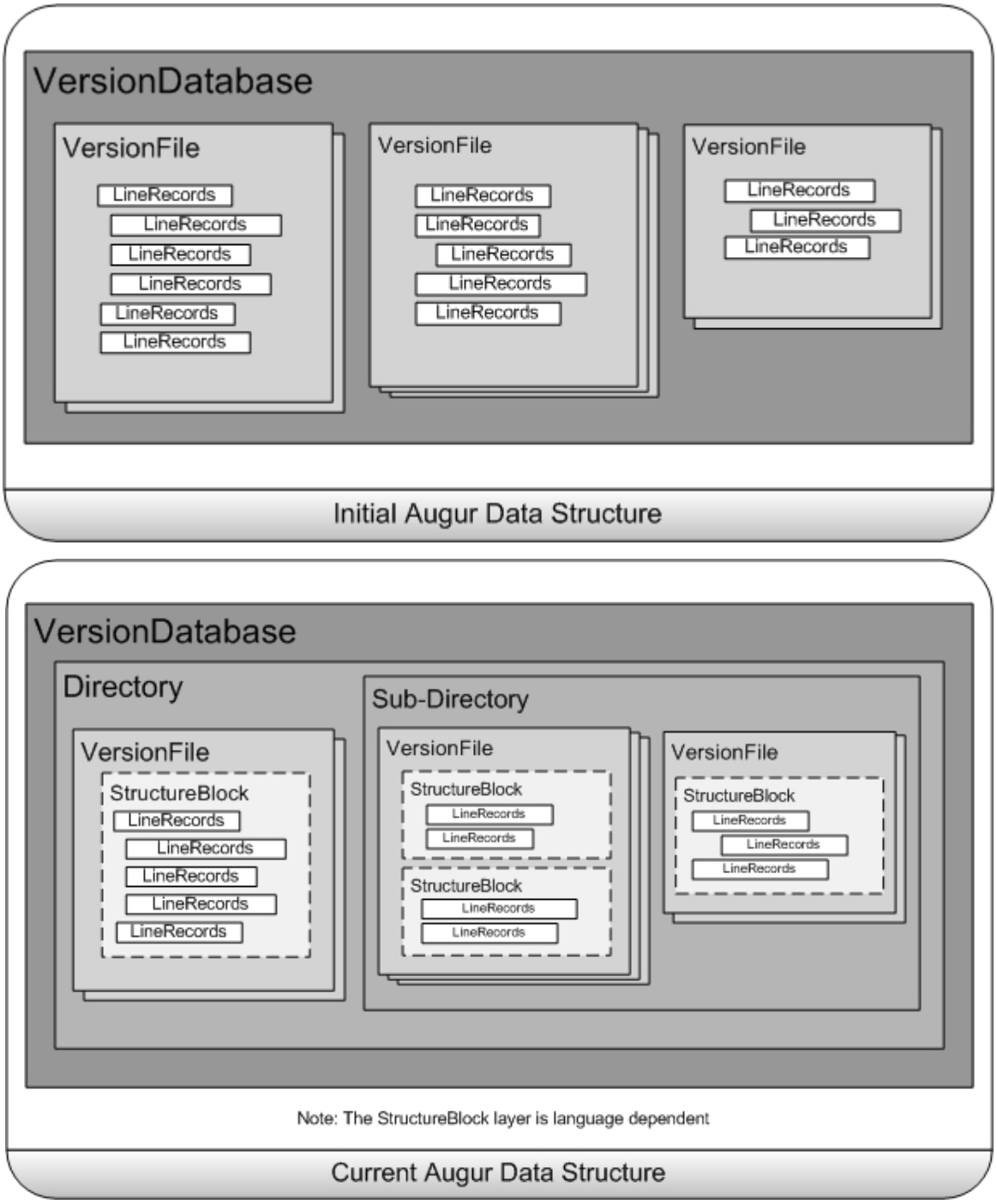

Figure 4.6 Augur's data storage hierarchy
(top) Augur's initial data storage hierarchy; this is analogous to a CVS view of data. (bottom) Augur's existing data storage hierarchy includes two new layers, which allow for more flexibility and detail in the query system

4.2.5 Augur Evaluator Query Framework

The Augur Evaluator Query Framework was developed to meet the varied information needs of the Augur query clients (e.g. the visualizations). The framework provides a set of standardized, but flexible, query interfaces that define how data can be accessed from the VersionDatabase. Central to this data access model is the concept of "Evaluators."

Evaluators were introduced to enhance the flexibility of the query framework. They allow for dynamic, open-ended queries at each level of storage (from StructureBlocks to Directories to the VersionDatabase itself). Evaluators are similar to the predicate classes in the Apache Jakarta Commons-Collection[12] in that they both defer computation to user-defined classes in order to filter subsets of data. Evaluators are classes that define a set of comparison-based criteria in order to filter response data from queries.

For example, the GetDateNewest evaluator is used to return the most recent date in any given data structure implementing the QueryableDates interface; this allows the evaluators to apply across different levels of data storage. The GetDateNewest evaluator, then, can return the most recent date at the project, directory, file, or structure block level. This broad applicability to different levels of data storage structures and the allowance for user-defined evaluator classes is what makes the framework so powerful. New evaluators can be easily developed and configured to form new queries or expand on existing ones without modifying the query system itself.

Evaluators are used in accordance with the Augur Evaluator Query Interfaces, which are presented in Table 4.4. There are three important things to note here: one, that no data

---

[12] http://jakarta.apache.org/commons/index.html

storage class implements all query interfaces; two, that the implemented query interfaces are a reflection of the storage hierarchy (e.g. the Directory class implements the file query interfaces, but the StructureBlock class does not. This is because directories contain files, structure blocks do not); three, query interfaces are also limited by the underlying configuration management data (e.g. StructureBlock classes do not implement check-in related statistic query interfaces because these statistics are stored at the file and directory level in CM systems and are unavailable at finer granularities).

Table 4.4 Augur Evaluator Query Interfaces

Interfaces	Data Storage Structures				
	Version Database	**Directory**	**Version File**	**Abstract Structure Block**	**Line Record**
QueryableFile	✓	✓			
QueryableFiles	✓	✓			
QueryableCheckIn	✓	✓	✓		
QueryableCheckIns	✓	✓	✓		
QueryableStructureBlock	✓	✓	✓		
QueryableStructureBlocks	✓	✓	✓		
QueryableAuthors	✓	✓	✓	✓	
QueryableDates	✓	✓	✓	✓	
QueryableLines	✓	✓	✓	✓	
QueryableLine			✓	✓	
QueryableAuthor					✓
QueryableDate					✓

4.3 FRONTEND ARCHITECTURE

The frontend layer communicates with the backend layer through the Augur Evaluator Query Framework. The frontend uses data retrieved from the backend to create its visualizations and is, therefore, primarily concerned with features that are visible to the user (both the interface itself and the way it can be manipulated—user interactions). The frontend architecture can be partitioned into three components: the visualization framework, the AugurColorer, and the Augur event system (see Table 4.5).

Table 4.5 Three Main Components of the Frontend Architecture

	Component	Extensible	Responsibility
4.3.1	Visualization Framework	Yes	-Overall Augur interface -Provides framework for adding new visualizations
4.3.2	AugurColorer	Yes	-Manages coloring of data -Ensures that colors are consistent across all visualizations -Can prevent colors from being assigned to more than one data element
4.3.3	Augur Event System	Yes	-Ties visualizations together through events -Handles events generated by user interactions

The visualization framework controls the overall look-and-feel of the Augur GUI. It provides an interface called AugurComponent which all Augur visualizations must implement in order to be added to the Augur visualization system. Each AugurComponent relies on the AugurColorer for its color mapping data. The AugurColorer was developed to decouple color management from the visualizations themselves (so that, for example, color mappings could be maintained across

visualizations). Finally, AugurComponents are tied together by the Augur event system, which handles events generated by user interactions.

4.3.1 Augur Visualization Framework

The Augur Visualization Framework encompasses the graphical user interface, the AugurComponent system, and a variety of GUI support and utility classes. Augur's user interface allows multiple visualizations to be viewed concurrently; this is facilitated by a series of resizable split panes. The primary visualization (the augmented SeeSoft view) is located in the central viewing pane while the three secondary visualizations share the bottom pane. New secondary visualizations can be added to Augur by, first, implementing the AugurComponent interface and, second, adding the visualization's unique identifier (the AugurComponentType id) to the AugurComponentFactory.

The AugurComponent interface defines a set of methods that every Augur visualization must implement. The AugurComponent was designed as an interface rather than an abstract class so that developers could select any base class for their visualizations (be it a JComponent, JPanel, JTree, or even a user defined class) and not something strictly Augur specific.

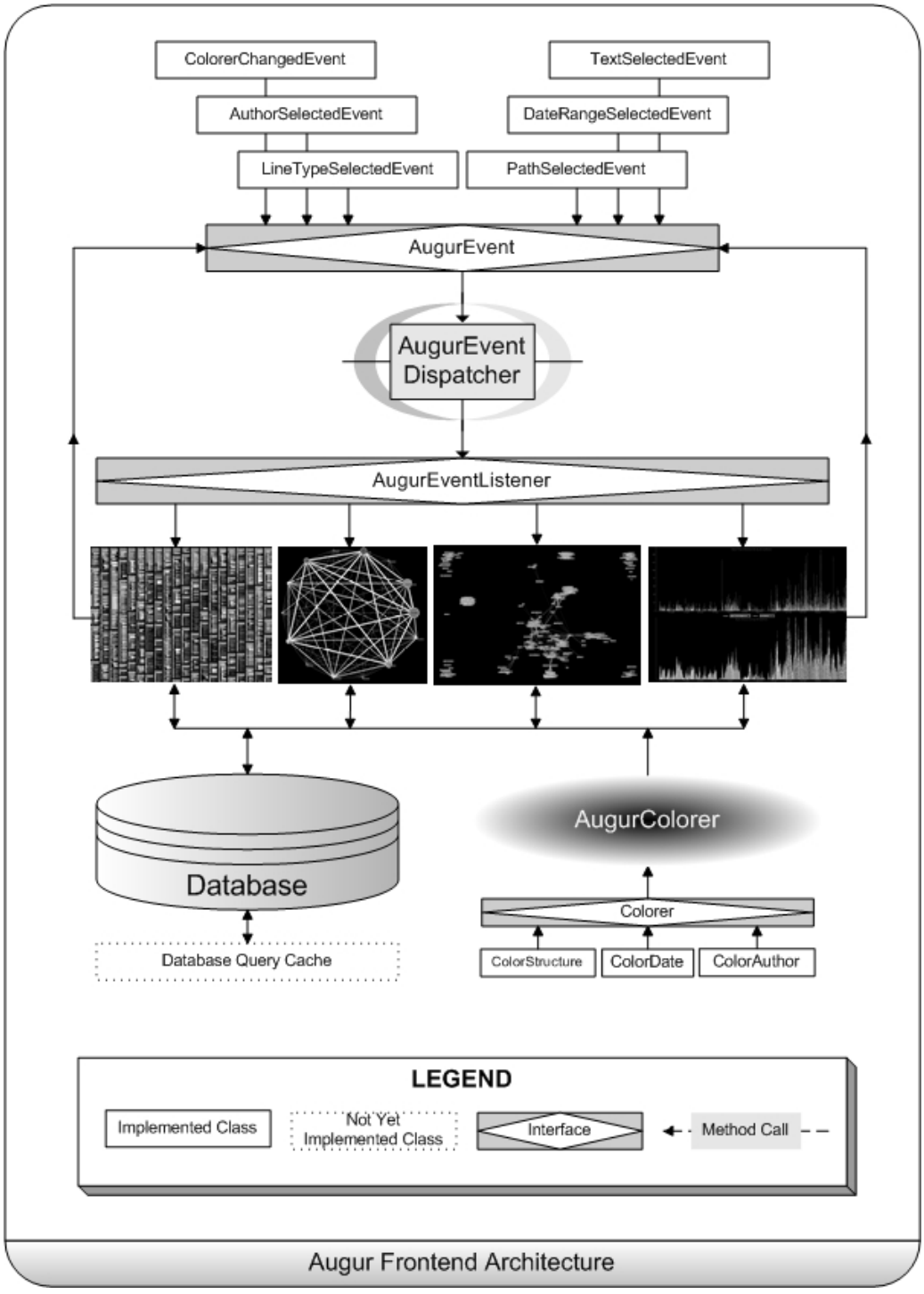

Figure 4.7 A detailed view of Augur's frontend architecture

4.3.2 AugurColorer

One critical shared aspect of all AugurComponent visualizations is color information. Data like author, date, and structure are consistently colored across visualizations so that, for example, a representation of the author "Sandy" colored in red in the social network view will also be colored red in temporal view. Coloring is therefore managed independently of the visualizations via the AugurColorer component.

The AugurColorer contains a collection of Colorer classes, which maintain color mappings for specific data sets. Two distinct Colorer implementations exist: the DiscreteColorer (see Table 4.6) and the ContinuousColorer (see Table 4.7). The DiscreteColorer is used for discrete data sets (e.g. the ColorAuthor class maintains color mappings for author data). The ContinuousColorer is used for continuous data sets (e.g. the ColorDate class maintains color mappings for time data). New data colorers can be added to the AugurColorer via the addNewColorer method.

The discrete and continuous colorer classes are not directly available to be invoked and called by the visualizations, but can only be accessed through the AugurColorer. This centrality reduces coloring complexity, as only one point exists for color information. It also eases the load of adding a new colorer to the system (as a new colorer does not need to be systematically added to the visualizations themselves).

The meaning of the color mappings are revealed through a series of color legends located at the top of the Augur GUI (see Section 3.2.5).

Table 4.6 Discrete AugurColorer Classes			
	Discrete Colorers	**Description**	**Mapping**
1.	ColorAuthor	Maps colors to author.	Mapping is set once for each author in a given project. Mapping is retained across executions.
2.	ColorHistory	Maps colors to before date, current date, after date.	Mapping changes with selected date.
3.	ColorJavaStructure	Maps colors to Java structure blocks (e.g. method blocks, constructor blocks, comment blocks, etc.).	Mapping is static.
4.	ColorLineType	Maps colors to line types (e.g. variable assignment line, method definition line, multi-line comment, etc.).	Mapping is static.
5.	ColorSelectedText	Maps colors to selected text strings.	Mapping changes with text selections.

Table 4.7 Continuous AugurColorer Classes			
	Continuous Colorers	**Description**	**Mapping**
1.	ColorDate	Maps colors to time.	Mapping changes with selected date range.
2.	ColorHistoryContinuous	Like ColorHistory but maps before, current, after colors on a continuous scale.	Mapping changes with selected date.

4.3.3 Augur Event System

Finally, an event mechanism is used to tie the visualizations together. The Augur event system synchronizes information sharing across visualizations via a publish/subscribe event model. This allows data selections and other user-triggered events that occur in one visualization display to be reflected in the other visualization displays as well. For example, if an author is selected in a visualization, other visualizations may react by presenting information specific only to that author. The line-oriented display, for example, would highlight all lines developed by the selected author while graying out the others.

Currently, eight distinct Augur events exist in the system, though more can be added by implementing the AugurEvent interface and updating the dispatch table in the AugurEventDispatcher class. Table 4.8 provides a list of the current Augur events.

Table 4.8 List of Current Augur Events			
	Event	**Description**	**Example**
1.	AuthorSelectedEvent	Event for new author selected	Author “ned” selected
2.	ColorerChangedEvent	Event for new colorer selected	“color by author” or “color by date” selected
3.	DateRangeSelectedEvent	Event for new date range selected	“09/09/1978 @ 9:30PM – 09/09/1999 1:03AM” selected
4.	DateSelectedEvent	Event for new date selected	“10/26/1951 @ 8:51AM” selected
5.	LineTypeSelectedEvent	Event for new line type selected	“class definition” selected
6.	PathSelectedEvent	Event for new paths, files, or structure blocks selected (or any combination of the three)	“File1.java and the compareTo(Object) method in File2.java” selected
7.	SBTypeSelectedEvent	Event for structure block type selected	“constructor ” selected
8.	TextSelectedEvent	Event for text selected	“any text string” selected

4.4 Architecture Conclusion

Augur is designed to support a wide range of extensions, from expanding the backend system to support additional configuration management systems, structural analyses, and programming languages, to extending the frontend to include new visualizations and color mappings. Augur's architecture provides a generic framework for analyzing source code and meta-data in a visualization system. It provides suitable abstractions such that developers interested in expanding Augur to visualize their own data (e.g. profiling data or test analysis data) are not are not consumed with low-level issues like communicating with the CM server or Java language parsing.

Chapter 5 VALIDATION AND USER EXPERIENCES

Augur is designed to improve coordination in software development teams. There are two approaches to team coordination. A centralized coordination strategy attempts to match each user's activity to a common reference point. Most process models take this approach; the process is the basis of coordination, and each user's activity is mapped onto the common process definition. Alternatively, a distributed strategy attempts to support separate coordination between individuals without requiring a common perspective or shared understanding across the whole team. While this approach seems less efficient, it can be more effective for a number of reasons. First, it is more easily introduced into existing settings, operating alongside other software development practices. Second, it recognizes that developers play different roles and have different concerns, so that, for example, their interpretations of others' actions will differ. Third, it more easily accommodates change and evolution in the development process.

Augur supports decentralized coordination. It aids coordination not by bringing everyone into alignment with a common perspective, but rather by providing developers with an enhanced understanding of the work of others and of the group, allowing them to make appropriate decisions about their own activity. So, while we aim to support development work in teams, our focus for validation is on individuals using Augur to visualize and examine the work of others.

Effective evaluation of Augur cannot be conducted in the laboratory. Augur is designed to support the ongoing coordination of development teams, and so true validation requires longer-term deployment and an analysis of the impact of the system on collective development practices. Although logistically difficult, we are currently

pursuing this goal (see Section 5.2). In the meantime, however, we felt it important to seek some more informal validation of our approach.

5.1 CASE STUDIES

Visualization systems work by allowing users to perceive meaningful patterns and regularities and also exceptions and variations in the images they present. This is a skilled task; users must first acclimate themselves to the system before extensive interpretations become possible. In demonstrating and experimenting with Augur, we have found that the system is much more compelling when viewing a codebase that is known to the viewer. Artificial experiments in which groups worked with unfamiliar software systems and unfamiliar partners, then, would be an inappropriate means of investigation or validation. Instead, to gain some initial feedback on Augur's effectiveness, we have conducted informal evaluations with developers engaged in active development of multi-authored systems. These studies have been conducted while the tool has been in development, and the interface has changed somewhat between revisions, but the core functionality has remained largely stable.

5.1.1 Case #1: "J" and Three Apache Projects

J is an active member of the apache.org open source community. He explored three Apache projects. The primary one we report on here is a core portability layer; it consists of 78,180 Lines of Code (LOC)[13] in 332 files, with a total of 32 authors, 16 of whom have more than 1500 LOC currently checked-in under their name. The first check-in was August 17th, 1999 and the project is still active. J's projects are written in C, which at

[13] LOC including whitespace and comment lines

the time was not supported by the structure analysis component, and so he did not use those views.

J used views of project history to reveal activity patterns; he observed, "One of the interesting things you can get here is project growth over time. You can see there are a lot of files that are still gray." Scrolling forward through time revealed which lines of code were added to the project and even when/what files were added. This exploration was combined with activity graphs, indicating major changes: "All of the sudden something happened on this date where the file went back down. Something got refactored." Structural details provided context to these explorations. "Those are big preprocessor definitions. It's a big conditional statement, really nasty."

Perhaps because this project is so large and complex, J concentrated his attention on the sets of authors. One feature that stood out was the number of multi-authored files. In fact, though a majority of code was composed by three or less authors, there was a surprising amount of files with eight, nine, or even fifteen separate authors. Unusual cases stood out; noticing a large file (over 500 LOC) with all but two lines by the same author, J commented: "Look at this windows file: what happened here was that [author1] is a windows person so he writes all the windows code and this poor guy – [author2] – just added two lines. What the heck did [author2] do?" Then, using the magnifying box, he could answer, "Ah, yep, later he did the include and license line."

5.1.2 Case #2: "D" and the Open Source Graph Toolkit

D is one of four developers on an open source project for modeling and analyzing graph and network data. The project was first registered at sourceforge.net in early February 2003 and, since that time, has seen steady growth from 1620 LOC in 10 files to

its current state of 35,000 LOC in 268 files. The project made its first 1.0 public release in early August, 2003 (approximately 4 months after the first source check-in).

D tended to use multiple visualizations in coordination. First, he would manipulate the graph views until an interesting pattern emerged. Then, he would drill down using the line-oriented view. These multiple views allowed him to see relationships in his code at both the broad and detailed level. D would fairly rapidly brush the mouse over a sequence of files, stopping only when he perceived an interesting revision pattern: "So now when I highlight this file, this pattern or this shape says to me, I checked it in, I made only trivial tweaks – one or two lines – and then I stopped playing with it on that date."

This strategy also allowed D to see how the selected activity relates to other files in the project: "This is intriguing now, in that I am really enjoying the idea of seeing different sorts of [line graph] patterns. You know we had the static check-in earlier – the file was just checked-in once and left. In contrast, over here we have [file1.java] with a small amount of activity, followed by a surge, followed by a ton of stuff." He found a second file with an unusual growth pattern: "…that file was a point of contention in which there was some debate or discussion about its proper role. Judging by the fact that it kept growing; people kept sticking stuff in it and then, in a burst, a whole bunch more stuff was put in, twice. In that final deletion, however, 60 or 70% was cut out."

The broad view of activity history allows users to relate the views they see to the "natural history" of the project, understood in terms of major transitions and events; D noted, "Here is the check-in with the copyright notice headers, but a lot of other stuff at the same time too. It seems that the major check-in of this day was the heading, which was five lines per file. There were a few other lines in some files, where it seems that

[author1] added more than just the license. Also, it looks like some entire files were added that day… Sort of undisciplined of us, wasn't it?"

The combination of structure and author views revealed different coding styles among the developers. For instance, one developer had a different indentation style ("he uses a different development system that automatically formats his code") and was the only author to use switch/block statements, which prominently stood out when looking at the line-types. This view also exposed that the developers "seem to have a variety of import styles, sometimes narrow, sometimes long (which is often a reflection of how involved a file is – 'ah, imports, this is a file that uses lots of stuff'). Some classes have two constructors some have many more… "

5.1.3 Case #3: "S" and the Open Source Graph Toolkit

S is another developer working on the same project as D. He used a slightly updated version of Augur which could display subsets of files based on the repository path selected in the file tree view. After becoming acquainted with the interface, S selected a particular repository path and began investigating. "I selected this repository 'cause I know I wrote all this code… only, oh no, let's see, what is this?" Although he had thought he was the only author of this package, he noticed multiple colors in the author column for a few files. Exploring a little further, he noted, "Oh, I see now, all the changes by this author have only been in the comment sections in the second column." Interestingly, Augur first seemed to contradict S's understanding of other author activity in the displayed files, but then served to reinforce this understanding by noting that changes had only been made to comments.

Displaying by author, S was surprised that the graph pane showed that some authors appeared to contribute more lines of code than he had. “Hmm, well, I’ve definitely written more… well, in my mind I’ve written more than [author] has.” However, correlating this with the line-oriented view showed that [author] checked-in the license header for every file in the project and, therefore, his total line contribution number was a little distorted.

Finally, S commented that he found the combination of multiple attributes in the display “extremely useful,” since “files that are all one color in the first column are the least interesting to me. You want to see where people work on the same file.”

5.1.4 Case #4: “F” and the Open Source Web Project

F is the chief developer and administrator of a large open source project (117,325 LOC) for web-based document authoring, in progress since 1998.

The line-oriented view allowed F to correlate spatial arrangements with structural aspects of the code. “One thing I like with this is seeing the indentation. That gives me a feeling that [file] is too complex – that it should be re-factored, that it should be structured differently.” This was further correlated with structural information (“it’s easy to see that these are all little methods;” “over here, though, these little blocks are wrapper functions”).

F discovered a change he had made to the handling of global data. “Basically, before every function or every class had some kind of error message so I took that out and I put that into a global class.” Using the search function revealed many single-line calls to GlobalData, “Yeah, so you can see that most of these [highlights] are one or two lines.” Displaying search results in the line-oriented view helped F contextualize this

information with author, structure, and time data: “What I’ve changed fairly recently in GlobalData is, oh yes, here we are, is configuration stuff; for storing configuration entries. You can see this is a fairly recent change.”

Much of F’s use relied on the combination of focus and context achieved through the magnifier tool. “Where I always have problems with textual representations is trying to figure out where does this function begin, where does it end? If it goes over the screen size, then you are scrolling back and forth and you are losing context. With this and because you are showing indentation and the [structure] column information, I think it’s much easier to have the context of the functions.”

5.2 Ongoing On-Site Field Study

The first round of case studies, although informal, allowed us to gain insights into how Augur would be understood and used by developers while exploring their own codebase. These case studies employed a qualitative-based approach relying on eight sessions of one to one-and-a-half hour long interviews with developers of open source projects. In contrast, our second study deploys Augur into an actual distributed software development environment and makes use of both quantitative and qualitative methods. This evaluation is ongoing and is scheduled to conclude in early August of 2004.

The focus of our on-site evaluative study is a software development company based in both southern California and Italy; we will refer to them as NetSoft. NetSoft is a relatively new (approximately two years old) startup company with a dozen full-time developers at their California site and about the same number at their Italy site. Most of their development work is in ASP.Net with C# for the “codebehind”. To a lesser extent

they also use Java, Javascript, and Flash. They use CVS for their configuration management server and WinCVS for their CVS clients.

The goal of this study is to see how, in the real world, distributed software development teams use Augur. The evaluation methodology we are using is based primarily on usage logs and user feedback. A specially instrumented version of Augur was deployed to the California site in mid-April, 2004. This version monitors the visualizations, display modes, and Augur events triggered by the user during application execution. A series of interviews with developers towards the end of our study will be used to bolster our understanding of the usage logs and to acquire feedback about how the tool was perceived.

Chapter 6 Discussion and Further Work

The preliminary case studies in Section 5.1 were positive. Our sample users were interested and engaged, and gained insight into their code and their development practices through their use of Augur. They clearly exploited not just activity information and artifact information, but the relationship between the two in the ways in which they interacted with the views they saw. What is more, as one of them noted, this information comes essentially "for free." Augur generates no new information by itself, nor does it require any extra work from developers, but merely presents a visual depiction of information already available in the repository.

Each subject's use of Augur varied. J examined larger projects with considerably more developers, a number of whom were not known to him. D and S examined a smaller project, currently in development along with a few close colleagues. F's use was more retrospective, and focused on the system's history rather than distribution of author activities. The different uses that the subjects made of Augur seem to reflect these patterns: D and S focused more on the code and the change history as the primary view, F was more concerned with larger evolutionary patterns, and J focused more on the distribution of author activity throughout the system.

That said, there are clearly some commonalities across these experiences. They all made use of multiple perspectives using graph views as well as line-oriented views, and made use of structure, change history, and ownership perspectives. S perhaps made the most use of this, moving back and forth between these views, while F relied most heavily on the coordination between multiple columns. More importantly, they all made use of these views in coordination, triangulating on the information they needed by exploiting

information from different perspectives, and using one view to account for the information revealed by another.

These informal investigations are far from conclusive, but they nonetheless support our two initial hypotheses—first, that combining activity and artifact information in a single view provides developers with information that helps them understand their systems, and second, that the spatial organization of the code can provide a common framework that integrates different forms of information about software development. This common frame provides the coordination that allows developers to exploit multiple perspectives concurrently and deal with the relationship between activity and artifact.

6.1 Further Work

On the basis of this initial experience, we are moving forward to deal with a number of further areas of work. Most particularly, these include incorporating new layout methods for the line-oriented file panels, and more sophisticated tools for analyzing structure, such as dependency analysis [Callahan 1990]. These are briefly explained below.

6.1.1 New File Panel Layouts

During our preliminary case studies, it became clear that we could improve the way in which the file panels were arranged in the primary visualization window. The current system provides the user with a number of simple arrangements: organize the file panels by recent activity, by the number of revisions, by the number of authors, alphabetically, etc. However, all of these arrangements are essentially based on a bin-packing algorithm, which packs the file panels together as densely as possible. So, although there is a general

arrangement to the panels (e.g. arranging by the number of authors places the highest authored files towards the top and the lowest authored files towards the bottom), they are not necessarily arranged in such a way that reveals a higher level oraganizing structure like path.

For certain types of high-level explorations, this is not a problem. Particularly for those explorations in which the user is not certain what he/she is looking for and, instead, is guided by the interesting patterns revealed through the visualization. However, at other times, the lack of a meaningful layout results in confusion.

The downside of imposing a higher-level layout structure is that it will inevitably decrease the visualization's information density. As it currently stands, almost every available pixel in the primary visualization window is appropriated towards displaying information.

We are currently exploring new display techniques that address these layout issues. Three proposed solutions are included below. The first solution looks at clustering file panels into their respective directories and arranging them around a rotatable circle (Figure 6.1). The second solution arranges the file panels in terms of their file/folder hierarchy like in the filetree explorer view (

Figure 6.2). The last solution proposes implementing a high-level inset view of the primary visualization window; this solution could be beneficial independently of the file panel layout method used (Figure 6.3).

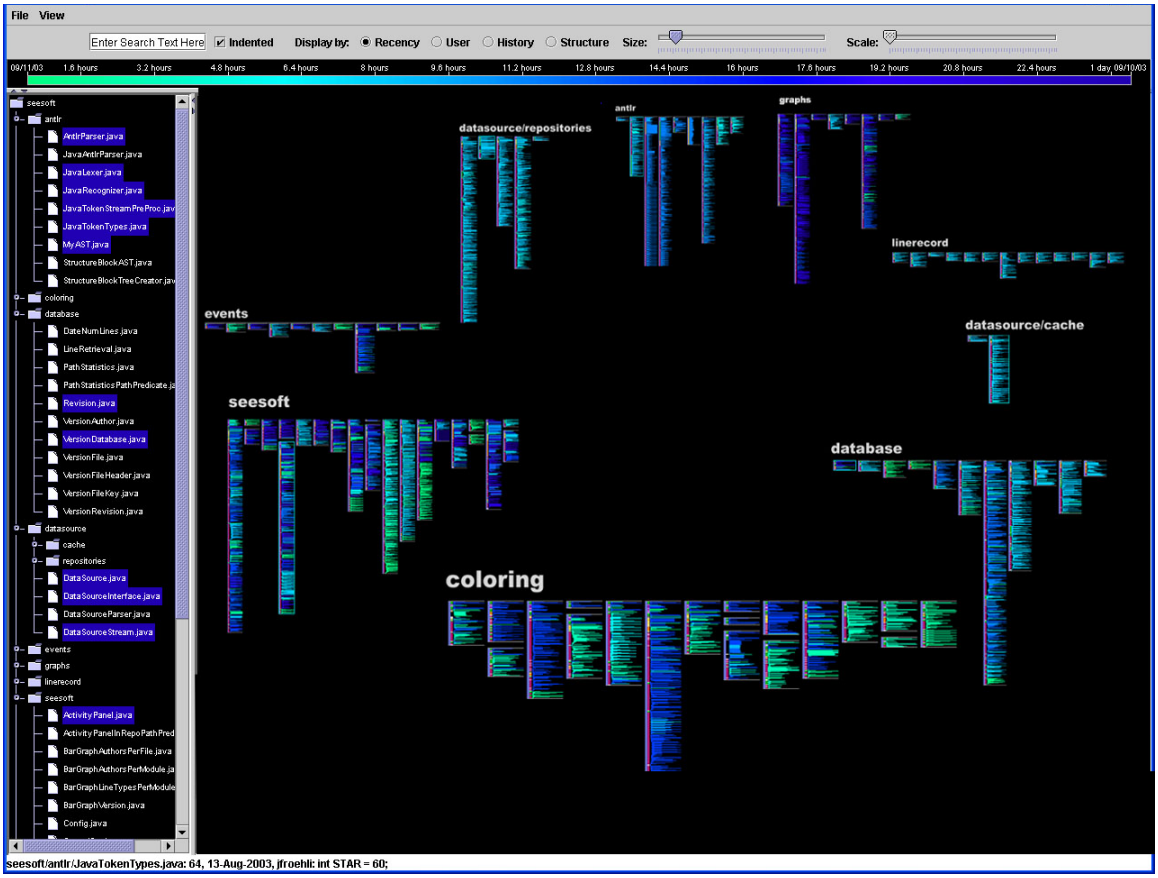

Figure 6.1 Circular file panel layout
This is a sketch view of the circular file panel layout design. The groups are clustered by path. Paths will be organized by activity. More recently active paths will be zoomed in and towards the front while less active paths will be zoomed out and towards the back. The paths can rotate around the perimeter of the circle so that, for example, selecting a path opposite to the coloring path in this figure, will rotate all paths 180 degrees such that coloring (and surrounding directories) is at the top and zoomed out and antlr (and surrounding directories) is at the bottom and zoomed in.

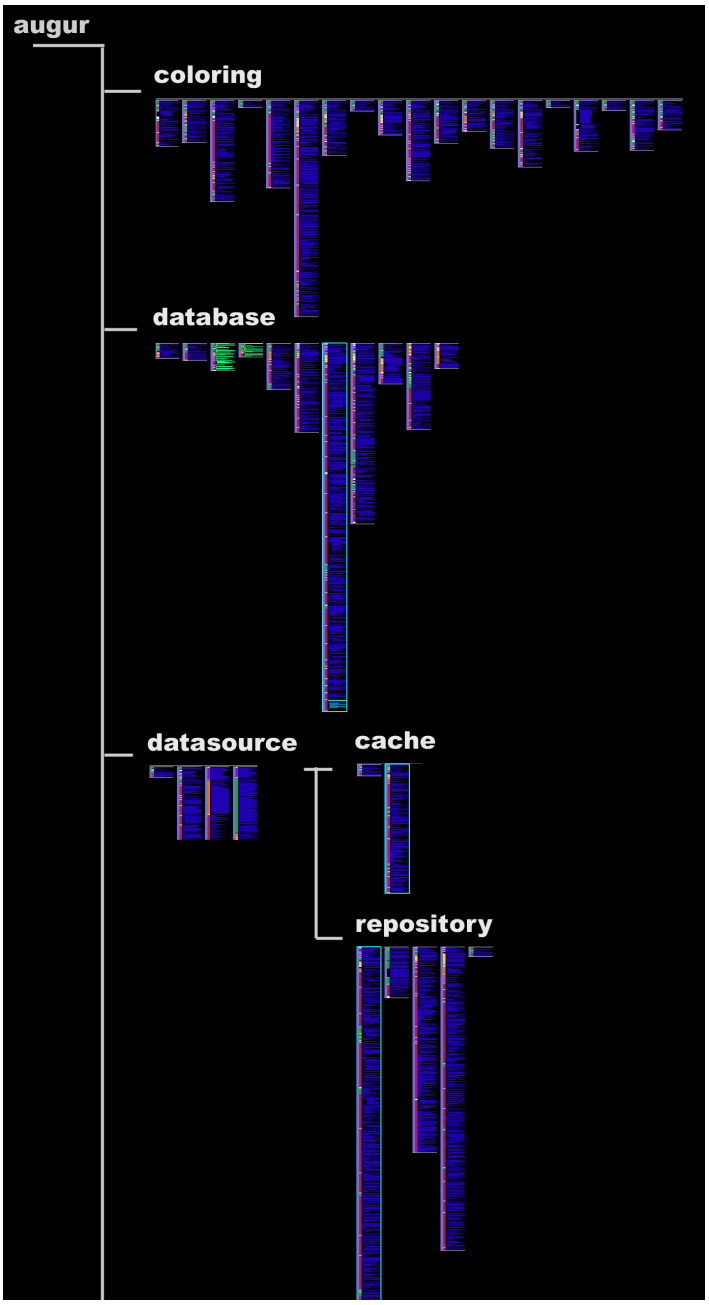

Figure 6.2 Hierarchical file panel layout
This is a sketch view of one possible new file panel layout technique. The file panels are presented in a folder/file hierarchy view like that of the filetree explorer. A zooming mechanism would allow the user to quickly zoom in and out of different areas in the codebase.

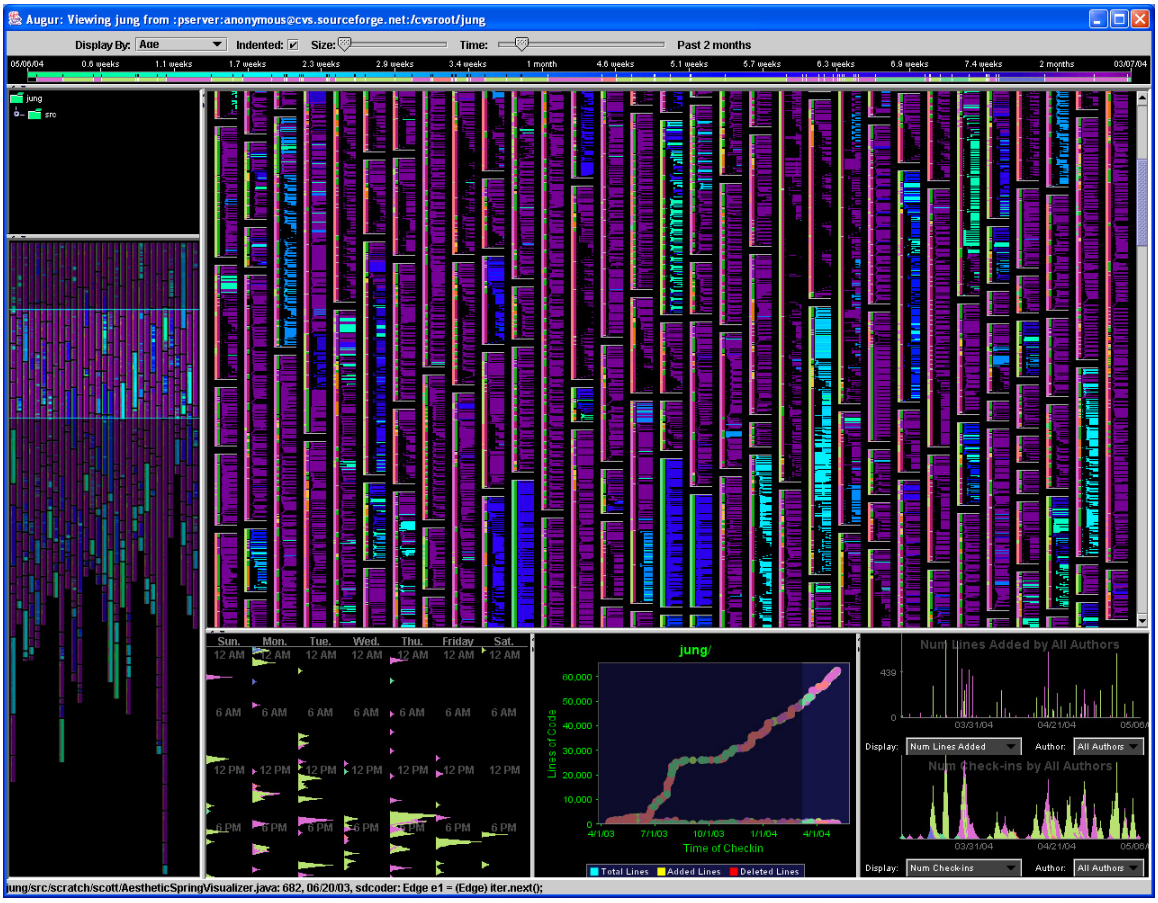

Figure 6.3 File panel layout legend
This figure contains a sketch view of the file panel layout legend located on the left side of the figure underneath the filetree explorer view. This is essentially a miniaturization of the primary visualization window and is designed to server as a guide for the user. The highlighted section in the file panel legend is what is visible in the primary visualization window.

6.1.2 Dependency Analysis

The original version of Augur incorporated only a simple static form of structural analysis, mainly language parsing and structure block hierarchies. This allowed us to classify lines of code according to their type and structural context, and so allowed a developer to see each line of code in terms of the larger structures within which it was embedded. In our more recent versions of the system, we have begun to augment this view with information that explores the dynamic structure of code.

In particular, we have incorporated call-graph analysis[14]. A call graph is a data structure that describes which elements of a software system make use of which other elements. Software systems are constructed in terms of procedures (or "functions" or "methods"), which may in turn make use of the results other procedures, just as, in mathematics, a function can be defined which makes use of the results of other functions (e.g. if f(x) = sqrt(x) + 1, then the function f makes use of the function sqrt.). A call graph lists all the procedures in a software system, and, for each procedure, shows what other procedures it makes use of.

A call graph, then, reveals the potential dynamic structure of a software system, although it can be derived using static analysis techniques (i.e., it can be extracted directly from the source code, without examining a running instance.) More importantly, in demarking dependencies within the code (between one procedure and another), it also begins to suggest dependencies within the development team (between the maintainer of one procedure and the maintainer of another.)

---

[14] Call graph analysis is currently only available in Java using the call-graph analysis functionality in the Soot Java Optimization Framework. Please refer to: http://www.sable.mcgill.ca/soot/ for more information.

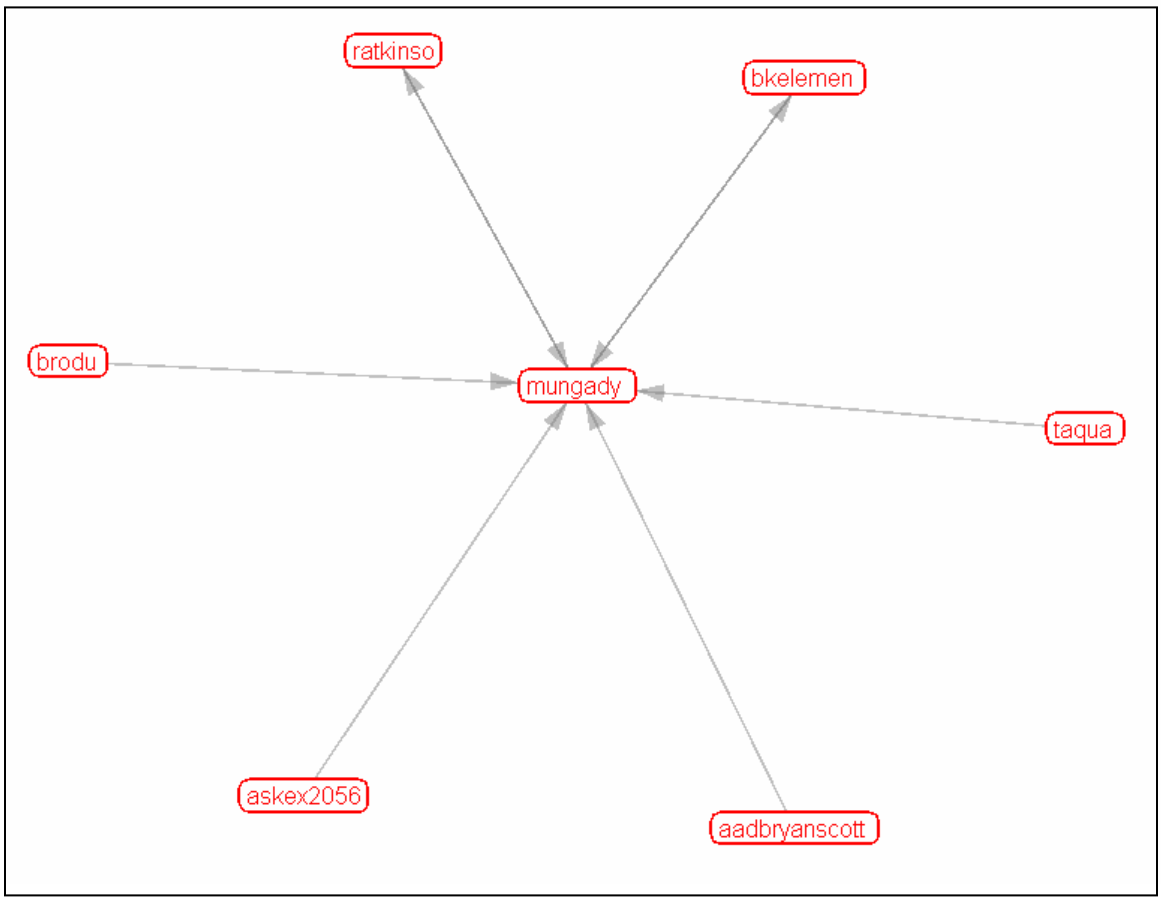

Figure 6.4 Author dependency analysis graph
This prototype graph reveals author source code dependencies. Instead of drawing a module based call graph, we extend this to incorporate information about the author's of these modules. Notice that in case all authors in the project are dependent on code written by the central author, "mungady" (each author node has a directed edge to mungady). Also, authors in the peripheral group do not call each other's code.

This relationship between members of the development team is made more explicit in the network view. In this view, Augur draws views of the network of contributors to a project, relating them according to source code dependencies (see Figure 6.4 above). Each author in a project is represented as a node, directed edges between nodes exist when one author "depends" on another author's code (e.g. author A calls author B's function). The user is able to explore graphs like the one above and then relate these relationship patterns back to the source code itself in the primary visualization window.

For example, clicking on the edge from “brody” to “mungady” in Figure 6.4 would highlight all the code written by brody that depends on mungady in the source code view in the primary visualization window.

Chapter 7 CONCLUSIONS

Software development is complex; distributed software development is even more so as the complexity of collaboration is added to the complexity of the artifact. Most tools focus on one or the other of these concerns. We have been exploring a visualization-based approach that allows developers to understand the relationship between them, embodied by a prototype tool called Augur. Our initial, informal user experiences have been positive. They demonstrate two things—first, that the tool provides meaningful information to developers working on project teams, and second, that the combination of information about activities and artifacts helps provide context that developers can use to understand development processes.

The central element of our approach is to exploit the spatial structure of the source code as the unifying principle for organizing many different forms of information. The source code is the common artifact around which all developer activities take place; its structure unifies their actions. Our approach seeks to take the artifacts that mediate activity and to make them into "inhabited spaces," revealing the actions and activities of the communities who work with them. Our experiences with Augur suggest that this is an effective approach for stitching together the representations of action that many software tools produce.

REFERENCES

Ball, T. and Eick, S. Software Visualization in the Large. Computer, 29(4), 33-43, IEEE, 1996.

Bieman, J., Andrews, A., and Yang, H. Understanding Change-Proneness in OO Software Through Visualization. Proc. 11th IWPC 2003, 44-53.

Boehm, B. and Bose, P. A Collaborative Spiral Software process Model based on Theory W, Proc. of 3rd ICSP IEEE, New York, 1994, 59-68.

Callahan, C., Carle, A., Hall, M., and Kennedy, K. Constructing the Procedure Call Multigraph. IEEE Trans. Software Engineering, 16(4), 483-487, 1990.

Clifton, M. H. A Technique for Making Structured Programs More Readable, ACM SIGPLAN Notices, v.13 n.4, p.58-63, April 1978

Cubranic, D. and Murphy, G. Hipikat: Recommending Pertinent Software Development Artifacts. Proc. 25th ICSE 2003, 408-418.

Dourish, P. Process Descriptions as Organizational Accounting Devices: Notes of the Dual Use of Workflow Technologies. Proc. ACM Conf. GROUP 2001 (Boulder, CO), ACM, New York.

Dourish, P. and Bellotti, V. Awareness and Coordination in Shared Workspaces. Proc. ACM Conf. CSCW 1992, ACM, New York.

Egyed, A. A Scenario-Driven Approach to Trace Dependency Analysis. IEEE Trans. Software Engineering, 29(2), 116-132, 2003.

Eick, S., Graphically Displaying Text. Journal of Computational Graphics and Statistics. Vol. 3, Number 2, 1994.

Eick, S., Steffen, J., and Sumner, E. SeeSoft: A Tool for Visualizing Line-Oriented Software Statistics. IEEE Trans. Software Eng., 18(11), 957-968, 1992.

Eick, S., Graves, T., Karr, A., Mockus, A., and Schuster, P. Visualizing Software Changes, IEEE Trans. Software Eng., 28(4), 2002, 396-412.

Finkelstein, A., Kramer, J., Nuseibeh, B. Software Process Modelling and Technology. RSP Ltd, 1994.

Froehlich, J. and Dourish, P. 2004. Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams. Proceedings of the International Conference on Software Engineering ICSE 2004 (Edinburgh, UK), 387-396.

Grinter, R. Using a Configuration Management Tool to Coordinate Software Development. Proc. COOCS'95, ACM, New York, 1995, 168-177.

Griswold, W., Yuan, J., and Kato, Y. Exploiting a Map Metaphor in a Tool for Software Evolution. Proc. ICSE 2001 (Toronto, Ontario), 2001, 265-274.

Harris, R. Information Graphics: A Comprehensive Illustrated Reference. Oxford University Press. 2000.

Herbsleb, J., Atkins, D., Boyer, D., Handel, M., and Finholt, T. 2002. Introducing Instant Messaging and Chat into the Workplace. Proc. Conf. CHI 2002 (Minneapolis, MN), ACM, New York, 171-178.

Hill, W. and Hollan, J. History-Enriched Digital Objects: Prototypes and Policy Issues. The Information Society, 10(2), 1994.

Hill, W. and Hollan, J. Edit Wear and Read Wear. Proc. Conf. CHI ACM, New York, 1992, 3-9.

Hutchins, E. Cognition in the Wild. MIT Press, Cambridge, MA, 1995.

Jones, J., Harrold, M. J., and Stasko, J. Visualization of Test Information to Assist Fault Localization. Proc. ICSE 2002, ACM, New York, 467-477.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect-Oriented Programming, Proc. ECOOP (Lecture Notes in Computer Science 1241), Springer, Berlin, 1997.

Maletic, J., Marcus, A., and Collard, M. A Task Oriented View of Software Visualization. In Proceedings of the 1st IEEE Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2002), S. 32-40, Paris, France. 2002.

Mockus, A. & Herbsleb, J.D. Expertise Browser: A Quantitative Approach to Identifying Expertise. In proceedings of International Conference on Software Engineering (ICSE 2002), pages 503-512, Orlando, FL, May 19-25, 2002.

Murphy, G., Notkin, D., and Sullivan, K. Software Reflexion Models: Bridging the Gap Between Source and High-Level Models. Proc. Symp. FSE (Washington, D.C.), ACM, New York, Oct. 1995.

Murphy, G. and Notkin, D. Reengineering with Reflexion Models: A case study. Computer, 39(8), 29-36, 1997.

Naumovic, G., Avrunin, G., and Clarke, L. Data Flow Analysis for Checking Properties of Concurrent Java Programs, Proc. ICSE 99, pp. 399-410, May 1999, Los Angeles, CA.

Price, B.A., Baecker, R.M., and Small, I.S. A Principled Taxonomy of Sofware Visualization Journal of Visual Languages and Computing 4(3):211-266. 1993.

Richardson, D., Aha, S., and O'Malley, O. Specification-based Test Oracles for Reactive Systems, Proc. 14th ICSE 92, May 1992.

Robertson, G., Card, S., and Mackinlay, J. Information visualization using 3-D interactive animation. Communications of the ACM, 36:57-71, 1993.

Scaife, M. and Rogers, Y. (1996) External Cognition: How do Graphical Representations Work? International Journal. Human-Computer Studies, 45, 185-213.

Storey, M.-A. and Mueller, H., Manipulating and documenting software structures using SHriMP views. Proc. ICSM IEEE, 1995, 275-285.

Sarma, A., Noroozi, Z., and van der Hoek, A. Palantír: Raising Awareness among Configuration Management Workspaces. Proc. ICSE 2003 (Portland, OR, May), 444-454.

de Souza, C., Redmiles, D., and Dourish, P., Breaking the Code: Moving between Private and Public Work in Collaborative Software Development. Proc. Conf. GROUP, ACM, New York, 2003.

Teitelman, W. Interlisp Programmer's Manual. Bolt, Beranek and Newman, Cambridge, MA, 1974.

Tufte, E. The Visual Display of Quantitative Information. (2nd ed.). Graphics Press. 2001.

Zimmermann, T., Diehl, S., and Zeller, A. How History Justifies System Architecture (or not). Proc. 11th IWPC (Portland, OR), 2003