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

Jon Froehlich and Paul Dourish
School of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425
jfroehli@ics.uci.edu, jpd@ics.uci.edu

Abstract

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.

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 (e.g. 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. [3, 11, 29]) attempt to help teams with the complexity of activities, while techniques and analysis and testing (e.g. [8, 22, 23]) 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 [1, 9]. Beyond previous approaches, Augur's contribution is a set of visualizations that combine information about the structure of both artifacts and activities. It 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 warroom;
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, it is designed to support *end-user visualization* rather than automatic inference to better adapt to different development settings. Second, it favors *online* rather than offline analysis for dynamic integration into the development process. Third, it 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 incorporated into existing development efforts without significant overhead.

In this paper, we first review the background of research into technologies for 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.

2. 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 [20, 21]. 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 [26]; 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 [30].

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 not simply to help developers understand a software system, but also as a tool to support them in coordinating collaborative development work.

Our particular interest is to do so 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 [15, 16] 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 [7]. 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 [12]. 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 [28]. More generally, formal processes can serve an awareness purpose; Dourish [6] 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 [14] uses Instant Messaging technologies to support inter-developer communication that are integrated with their development activities; alternatively, Palantir [27] provides an awareness framework that operates in concert with configuration management systems.

3. Our Approach

Our current research proceeds from the observation that software teams struggle with *both* the artifact 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 code-base, 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. [28] 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 and offline 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 dynamically.

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, SourceSafe, etc.) as well as providing a consistent framework for analytic extensions.

Visualization vs interpretation. The most careful balance to be resolved by the design is that 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. 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 [24]. Research into distributed cognition and external cognition has demonstrated the important role of representations in information processing tasks [17, 25]. 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.

3.2. Augur: Interface and Interaction

We have been exploring these ideas in a prototype system called Augur. 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 provides a set of linked visualizations displaying different characteristics of the software system under examination. The primary view is a spatially-organized view of the software artifact, inspired by Seesoft [9]. Seesoft and related systems (e.g. Tarantula [18] and Aspect Browser [13]) present an overview of a software artifact in which each line of code is represented by a line of pixels colored to indicate some attribute of the line such its author or modification history (these systems will be discussed in more detail below.) The line-oriented display provides an immediate overview of a great deal of information.

The basic Augur interface is shown in figure 1. 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. In this view, the color of each pixel line indicates how recently it was modified; this allows a developer, at a glance, to see how much activity has taken place recently and where that activity has been located.

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

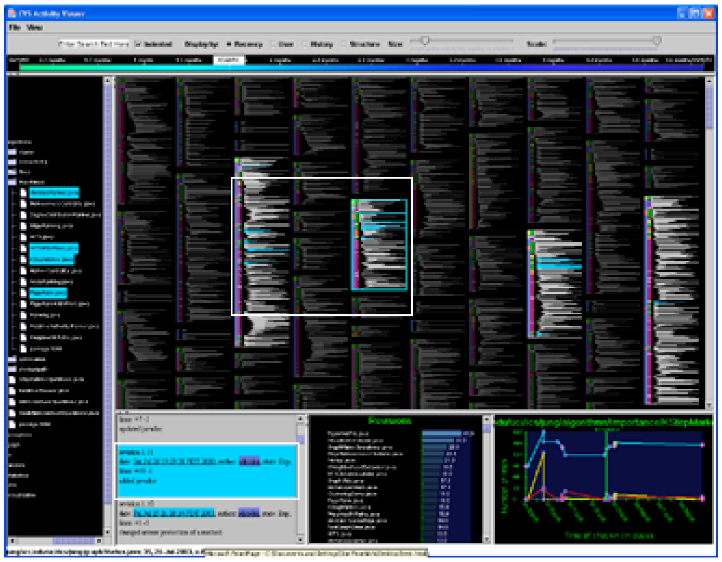
Annotation. In order to bring views of structure and activity together, the primary line-oriented display is annotated with subsidiary information in two extra columns that run down the left-hand side of each module block (see figure 1 top right). The leftmost column indicates which user modified that line of code, while the other shows code structure by indicating line type (block comments, method definitions, method separators, etc.) Juxtaposing these columns allows developers to see at a glance whether recent activity has added whole new methods or modified existing ones, for example.

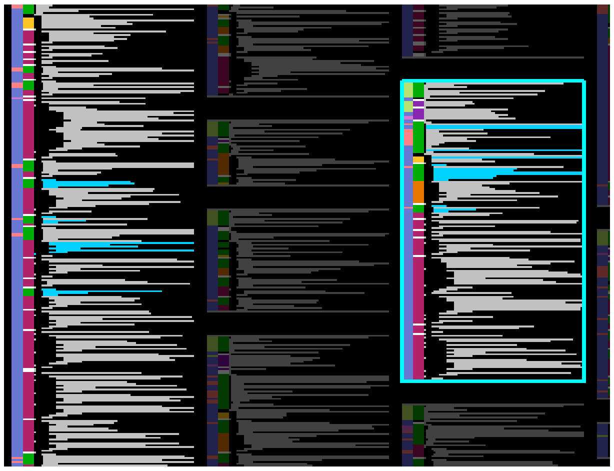
By default, change history is the primary display attribute (mapped to the pixel line color) while users and code structure are subsidiary attributes, indicated in the adjoining columns. However, users can switch back and forth between different configurations of primary and subsidiary attributes, e.g. making user or structure primary, in order to more easily examine the relationship between system structure and development activity.

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, which the developer characterize the kind of work being carried out by placing the work in a structural context. 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 change history, the accompanying panels show the modules according to their overall change history and a detailed change graph for the currently selected module. Similarly, when "user" is selected as the primary attribute, the graphs display information concerning each user'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.

Overall, then, the central feature that Augur uses to tie together multiple forms of information is the spatial organization of the source code. This spatial arrangement is familiar to all developers and common across different





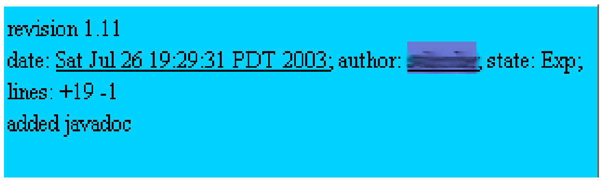

revision 1.11
date: Sat Jul 26 19:29:31 PDT 2003; author: [redacted]; state: Exp;
lines: +19 -1
added javadoc

Figure 1. Augur's multi-paned interface (left) and detailed insets (right). The line-oriented view uses three columns: the two secondary columns on the left are colored by author and structure respectively, and the primary column is colored by check-in date for each individual line (see top right). An inset of the revision history pane (bottom right), which shows the comments for the selected date.

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 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.

Before going on to discuss our experiences using Augur, we first present an overview of its architecture and implementation.

4. Architecture and Implementation

Augur is not a software development tool in itself, but rather is an adjunct to existing development tools. An overriding concern was that it be flexible enough to accommodate different development situations, scenarios, and technologies. Augur supports three particular forms of extensibility – in repository protocols, in analytic tools, and in visual displays. These elements are designed around a central database, from which visual representations are generated, and an event infrastructure through which the elements communicate.

Repository protocols. This information is retrieved from external configuration management systems; Augur is designed to support a range of potential mechanisms for accessing software repositories, from which it extracts

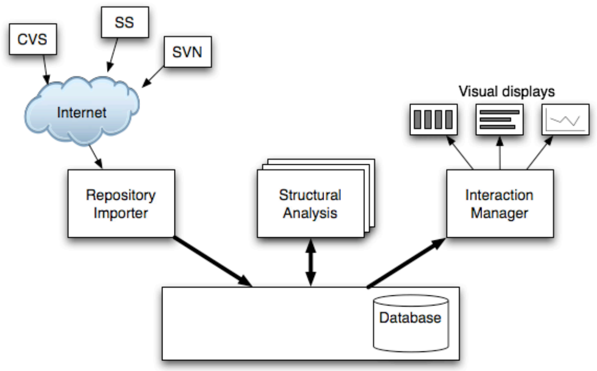


Figure 2: Augur's architecture

appropriate information about modules, check-ins, branches, etc. Compatibility with CVS has been particularly important so that Augur can be used to visualize the activity of major open source projects. Using network-based configuration management systems (such as CVS or Subversion) enables Augur to be easily incorporated into existing projects, but renders the system subject to considerable network delays when dealing with large projects. A caching mechanism provides some relief from this problem by locally storing recently downloaded project data.

Analytic tools. A second source of generality lies in the forms of artifact analysis supported. Once the target system has been loaded from the repository, analytic tools can be engaged to analyze the semantics of the code, perform structural analysis, etc. These results are recorded in the database alongside the raw information.

Visual displays. As well as being open to different repository protocols, the architecture also provides a generalized interface for visual displays. Multiple visual displays are generally available concurrently. The line-oriented view is the central display, but it is accompanied by other views that can compare users, present the history of the project, compare modules quantitatively, etc. Through an event mechanism, Augur ensures that these displays are maintained consistently; for instance, consistent color mappings between the displays allow users to move seamlessly from one to another, and changes (e.g. selections) in one display are reflected in the other displays so that correlations can be explored.

Data Organization. The 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, not least because most software configuration systems operate in terms of lines and files rather than modules and methods. This line-based data structure is then annotated with a range of other properties that augment the basic line information. For instance, a basic LineRecord in Augur is annotated with indentation information, a date, an author, a revision, an Abstract Syntax Tree (AST) node, and a Structure Block Tree node (which provides instant detail on the structural orientation of the line). This allows Augur to see lines not only in terms of their structural organization but also in terms of their temporal and social organization *and* the interplay between them.

Augur's interactive approach is based on multiple, dynamic visual displays that are linked together to form a consistent user experience. As users navigate from display to display, a central database supports the coordination between these views. Since each display visualizes different aspects of the underlying information store, and since they may interact in many different ways, we created an explicit query layer to mediate between the representations and the internal database. This abstraction supports the integration of new structural analyses and interaction models.

Structure Analysis. As the basis of structural analysis, Augur examines the source code that it finds in the repository, and constructs an AST. While this is a very basic form of analysis, it has been sufficient to demonstrate the effectiveness of combining activity and artifact information. Augur delegates source analysis to ANTLR, 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 parser generates two sorts of information (which are then recorded as annotations on the LineRecords). The first is line-type information, distinguishing between different types of source lines (comments, those that define methods or variables, etc). The second is structure block information that allows us to relate specific lines to the larger blocks within which they appear (e.g. method definitions, instance variable definitions, etc.) This is used to determine the extent of specific check-ins (that is, to generalize from the lines that have been modified to the methods that have been modified). Clearly, this same approach can also be used with call graph analysis or other approaches that reveal the interrelationships between lines and blocks of code.

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. 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 their users to perceive meaningful patterns and regularities in the images they present. This is a skilled task and relies on users being able to interpret what they see. In demonstrating and experimenting with Augur, we have found that it 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 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.

Case #1: 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 LOC[1] 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 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: "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 3 or less authors, there was a surprising amount of files with 8, 9, or even 15 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 [user1] is a windows person so he writes all the windows code and this poor guy – [user2] – just added two lines. What the heck did [user2] do?" Then, using the magnifying box, he could answer, "Ah, yep, later he did the include and license line."

[1] LOC including whitespace and comment lines

Case #2: D is one of four developers on an open source project for modeling and analysis of 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 (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: "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 [user 1] 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 user views revealed different coding styles among the authors. 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… "

Case #3: 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 module, he noticed multiple colors in the author column. 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 [user] has." However, correlating this with the line-oriented view showed that [user] 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."

Case #4: 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 refactored, 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."

6. Discussion and Further Work

These initial experiences 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 requires no 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 user 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 user 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 making 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.

On the basis of this initial experience, we are moving forward to deal with a number of further areas of work. Most particularly, these including incorporating richer measures of change severity [27], and more sophisticated tools for analyzing structure, such as control flow analysis [4]. We are also seeking more comprehensive evaluation of the tool through long-term deployment to active development groups.

7. Related Research

There are two primary areas of related work worth noting here. The first is the set of visualization systems descended from the original Seesoft system, while the second looks at other approaches for understanding activities and artifacts in development.

A range of systems have been developed that draw on Eick's pioneering investigations of line-oriented visualization of software statistics. His own 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 [1, 10]. By incorporating many perspectives, their tools can provide an extremely rich picture of organizational software activity. However, since they based their analysis on change records, they are not easily able to analyze the structure of the software system itself and to combine this view of the activity with the artifact (although change records do frequently indicate structural units). 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.

Griswold's Aspect Browser [13] is designed to support software maintenance and evolution. It is designed to help a software developer understand how particular features of a system are distributed through the code. Drawing on work on Aspect-Oriented Programming [19], Aspect Browser is targeted particularly towards cross-cutting concerns – features which 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, in terms of the semantic organization of the code.

A third example of the extended use of line-oriented visualizations is provided by Tarantula [18]. 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. This approach to fault localization draws on a three properties to which a line-oriented visualization 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.

There are two facts 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.

A second set of related investigations concern the relationships between system components or artifacts on the basis of development activities.

Experimental systems developed by Bieman et al. [2] and by Zimmermann et al. [31] also 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 [5] solves a related problem. It is primarily designed to help newcomers to a project to 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.

The relationships between artifacts derived by all three of these systems are possible approaches to incorporate into Augur; new forms of analytic interpretation that can reveal structure. There are three primary differences between the approach we have taken and these alternatives. First, we take a visualization approach that is more open to different sorts of correlations (depending on the different circumstances of development.) Second, we combine multiple perspectives 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, helping to tie the different perspectives more closely together. While this restricts the range of displays that can be provided, it also allows for a richer experience of the many different perspectives at work in any system.

8. 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 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.

9. Acknowledgements

This work was supported in part by the National Science Foundation through awards IIS-0133749 and IIS-0205724. We would like to thank Andre van der Hoek, David Redmiles, and Dick Taylor for their helpful comments on earlier drafts.

10. References

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

[2] Bieman, J., Andrews, A., and Yang, H. Understanding Change-Proneness in OO Software Through Visualization. Proc. 11th Intl Workshop on Program Comprehension (Portland, OR), 2003, 44-53.

[3] Boehm, B. and Bose, P. A Collaborative Spiral Software process Model based on Theory W, Proceedings of 3rd International Conference on the Software Process (Reston, VA), IEEE, New York, 1994, 59-68.

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

[5] Cubranic, D. and Murphy, G. Hipikat: Recommending Pertinent Software Development Artifacts. Proc. 25th Intl. Conf. Software Engineering (Portland, OR), 2003, 408-418.

[6] Dourish, P. Process Descriptions as Organizational Accounting Devices: Notes of the Dual Use of Workflow Technologies. Proc. ACM Conf.

Supporting Group Work GROUP 2001 (Boulder, CO), ACM, New York.

[7] Dourish, P. and Bellotti, V. Awareness and Coordination in Shared Workspaces. Proc. ACM Conf. Computer-Supported Cooperative Work CSCW 1992, ACM, New York.

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

[9] Eick, S., Steffen, J., and Sumner, E. Seesoft: A Tool for Visualizing Line-Oriented Software Statistics. IEEE Transcations on Software Engineering, 18(11), 957-968, 1992.

[10] Eick, S., Graves, T., Karr, A., Mockus, A., and Schuster, P. Visualizing Software Changes, IEEE Transactions on Software Engineering, 28(4), 2002, 396-412.

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

[12] Grinter, R. Using a Configuration Management Tool to Coordinate Software Development. Proc. Conf. Organizational Computing Systems COOCS'95 (Milpetas, CA), ACM, New York, 1995, 168-177.

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

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

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

[16] Hill, W. and Hollan, J. Edit Wear and Read Wear. Proc. Conf. Human Factors in Computing Systems CHI (Monterey, CA), ACM, New York, 1992, 3-9.

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

[18] Jones, J., Harrold, M. J., and Stasko, J. Visualization of Test Information to Assist Fault Localization. Proc. Intl. Conf. Software Engineering ICSE 2002, ACM, New York, 467-477.

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

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

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

[22] Naumovic, G., Avrunin, G., and Clarke, L. Data Flow Analysis for Checking Properties of Concurrent Java Programs, Proceedings of the 21st International Conference on Software Engineering (ICSE 1999), pp. 399-410, May 1999, Los Angeles, CA.

[23] Richardson, D., Aha, S., and O'Malley, O. Specification-based Test Oracles for Reactive Systems, Proc. Fourteenth International Conference on Software Engineering, May 1992.

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

[25] Scaife, M. and Rogers, Y. External Cognition: How do Graphical Representations Work? Intl. Jnl. Human-Computer Studies, 45, 185-213.

[26] Storey, M.-A. and Mueller, H., Manipulating and documenting software structures using SHriMP views. Proc. Intl. Conf. Software Maintenance, IEEE, 1995, 275-285.

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

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

[29] Sutton, S. and Osterweil, L. The Design of a Next-Generation Process Language. Proc. Sixth European Software Engineering Conf. (Zurich, Switzerland), Springer, 142-158, 1997.

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

[31] Zimmermann, T., Diehl, S., and Zeller, A. How History Justifies System Architecture (or not). Proc. 11th Intl Workshop on Program Comprehension (Portland, OR), 2003.