Repository Views for Rapid Exploration and Developer Insight

Michael Van Hilst
Farquhar College of Arts and Sciences
Nova Southeastern University
Davie, Florida USA
mv518@nova.edu

Shihong Huang
College of Engineering and Computer Science
Florida Atlantic University
Boca Raton, Florida USA
shihong@fau.edu

***Abstract*—The process of developing and maintaining software systems involves many artifacts. Developers create and change these artifacts to adapt and maintain the system. This work is often done with little knowledge of the artifacts' prior history and context. Online tools that are used to manage these artifacts leave clues to this history in the form of event records in a repository. Over the past 10 years, there has been considerable work in the field of repository mining to recover history from repository records. More recently, work has focused on simplifying the process for average workers to mine this information for themselves. This paper presents work on large scale projects in a leading telecommunication industry to reconstruct both history and context from repository records, and present the information in novel views for easy exploration and useful insight. Using these views, developers are able to gain valuable insight both into each artifact's role in the system, and about the experience of earlier developers attempting similar changes to the system. The results provide objective snapshots of the development process that help developers and managers make informed decisions about changes to the system. This project is a continuation of our previous work on non-invasive process metrics and analysis.**

Keywords— software engineering, repository mining, software maintenance and evolution, data visualization

I. INTRODUCTION

Our experience over many years of working with industry partners on the development of complex systems is that during the development process, there are 4 times as many alteration and repair tasks as new development tasks. One third or more of the effort is devoted to these alteration and repair tasks. Much of the trouble in delivering a reliable system is associated with artifacts that have the richest history of alteration and repair [1] [2] [3] [4]. Thus a substantial part of the effort in software system maintenance and development is devoted to understanding existing system artifacts, including their roles, history, and rationale.

Since 2002, we have been working to assist the development process by extracting various kinds of information about artifacts and processes and presenting it in useable forms. The work involves not only collecting the information, but combining it into views that have immediate value to developers or managers, and enabling them to explore and navigate the data to answer queries with a minimum of effort. Always, our emphasis has been on placing the least burden possible on those doing the actual work, while delivering to them the most possible value [5]. A subset of that work, focusing on developer views and ease of navigation, is presented here. In the experimental results of our previous work, we had encountered problems of presenting too much information with not enough focus [2]. In the latest work, the level of interactivity is greatly enhanced. The types of focus and the types of display have also been changed to provide better insight.

The developers' view of the system, when working on it, is through a narrow focus on a small subset of the artifacts. Each of those artifacts has important context and history that can help the developer decide what to change and what to look out for, and also where to find additional information. With the help of history, a developer might ask the following questions: 1. In what parts of the system does the artifact have a role? 2. Which tests revealed issues for which the artifact had to be changed? 3. Who else has knowledge of this artifact? 4. When was the artifact made? 5. How often has it been fixed or altered, and in each case, when, why, and by whom?

In the project described here, our goal was to extract the context and history of every artifact from common historical records and make it available to architects and developers for inspection and exploration in the easiest possible form. There is already a rich history of work on exploring artifact history. The work here differs from, or extends, that work in its focus on defining forms of synthesis and display, and mechanisms of interaction, for use in the everyday context of system developers. In this work, we focus on presenting facts and leave subjective forms of analysis to the developers, or possibly other tools. The data itself comes from a corporate repository, spanning many projects and product variants. Much of the published work in this area involves open source projects.

This work was partially funded by a grant from Motorola Mobility.

In this paper we report on work to create interactive views of information that is directly useful to developers and which cannot be otherwise obtained.

II. RELATED WORK

As the software system evolves over time, it needs constant change and update to meet changing requirements and new

978-1-4799-0053-4/13/$31.00 ©2013 IEEE

business challenges. Understanding the system development history is critical to make informed decision about the system. The research in the area of software maintenance & evolution [6] and mining software repositories has provided quite good theory and practice foundation in this domain. Changes and updates to the system could be responses to any of the three types of software maintenance:

- Corrective maintenance – change a system to correct deficiencies in the way it meets its requirements
- Adaptive maintenance – adapt software to a different operating environment
- Perfective maintenance – add or modify system's functionality to satisfy new requirements and improving the implementation

Canfora and Cerulo [7] use open source projects, such as KED, Gnome, Mozilla etc. to derive the set of source files impacted by a proposed change request. They use information retrieval algorithms to link the change request description and the set of historical source file revisions impacted by similar past change requests. Their method was evaluated by applying it to four open-source projects. However, the issue remains that the development paradigm of open source and proprietary software development are different. It is hard to tell how applicable the results from open source can benefit real-industry development.

Zimmermann et al. demonstrated the value of co-change analysis for predicting sites for further change. [8] Co-change analysis identifies items that are frequently changed at the same time. Their work was much more fine grained than ours, working at the level of individual source code lines, and was designed to derive a statistical analysis of the method's accuracy. The accuracy was shown to be high, at about 75%. But the computational cost was substantial. The resulting matrices of numbers are extremely fine grained and difficult for developers to interpret. We lead the developer to the artifact and give them a list of actual changes on which the analysis is based.

Grant et al. do a kind of cluster analysis on co-change lists to identify distinct topics in source code repositories. [9] In our case we are not trying to extract the overall architecture or even the structure of a system. We are just uncovering affinities between artifacts, in the form of easy to understand objective counts, to broaden and deepen the view taken by a developer working on an artifact within a larger system.

Lawrence et al. modeled the behavior of programmers navigating source code to find likely sources of a bug, based on information extracted from the bug report. [10] Our work does not address the initial search for the location of a bug. Rather it adds insight about factors concerning a chosen artifact that can be helpful before one starts to make a change, including strategies for navigating outward to additional sites.

A number of works have addressed the need for automating the process of mining software repositories. A recent example is BOA by Dyer et al. [11] These tools simplify the tasks such as cleaning the initial data, managing large heterogeneous sets of data, and creating queries against the data. We perform similar operations, but on the fairly stable repositories generated by management tools in a single enterprise. The users are not as concerned with cleaning data or writing new queries. In our case, the scripts for data acquisition, cleaning, and reduction only need to be written once. Our emphasis is on end users for whom the usage effort must be much lower still.

Voinea and Telea reported work with a similar goal of using visualization to address questions about a large repository. [12] But the questions they address – what is the contribution style, who are the main developers, what are the high level building blocks, how maintainable is the project, and what is the maintenance risk – are very different that those addressed here. Their questions call for a kind of statistical analysis and a result that calls for a very different kind of visualization.

Perhaps most similar to our own work is that of Murphy, et al. Like the work presented here, their Rationalizer tool also has views for when, who, and why. But their published work focuses more on history than context and navigation. In Bradley and Murphy [13] the information in their view assumes that the developer is already looking at a line of code. It displays the history of that code within the development environment. By contrast, our tools are used earlier in the development task and support rapid navigation across artifacts and artifact records. The two views are complementary.

A very nice study by Fritz, Murphy, and Hill [14], confirms our assumption that a developer who created an artifact or has made numerous changes to that artifact is likely to have the most knowledge about that artifact. Thus our graphical and tabular views, described in Section V, of which developers changed an artifact, how often, and when, can quickly indicate who has the most knowledge of that artifact.

In a more recent paper on answering developers' questions, after surveying the literature, Fritz and Murphy collected 46 questions. [15] Of those 46 questions, 27 concerned artifacts and other developers. The rest concerned builds, test cases, APIs, comments, and management scheduling. Of those 27, 3 concern APIs, 1 asks about code clones, 2 concern future work assignment, and 2 concern the project in general. The topic of those last 2 was addressed by our earlier tool. The remaining 19 can all be answered by our current tool. Like our work, Fritz and Murphy emphasize speed and ease of answering questions. Although they do not present a view of their tool, the description of its use involves query composition and filters, which would make it considerably more complicated than the process presented here. With our approach, answering questions about the history of artifacts, developers, and tasks requires only a few clicks, as we describe in the remainder of this paper.

III. SOURCES OF INFORMATION

As stated in the introduction, when making modifications to the system, the developers' view and focus is on individual artifacts. For example, to add a function or correct malfunctioning behavior, the developer must inspect a number of source code files and ultimately make changes to a few of them. While making a change, its impact on other parts of the system must be examined. Relevant tests should also be run. Our goal here is to help developers make better decisions about

where to seek information, what to change, what to inspect, and how likely they are to encounter problems, by giving them quick access not only to the events in each artifact's historical record, but also the broader context and immediately related events. The challenge is to present it in an easy to use form to quickly answer the developer's basic questions about an artifact, as described above.

The information to be shown is extracted from the repositories of two sources of information, i.e., task tracking and version control. In task tracking, each task is tracked for scheduling, status, and progress using online management tools. The records from such tools give us start and end dates for each task, the type of task, and additional clues to the purpose of a task, such as linking it to a particular test or requirement. In version control, every change to every artifact is recorded. These records include the identity and location of the artifact, the associated product version, the event date and time, and the person or persons responsible. As reported in earlier papers [1] [3], we link these two sources of information by associating each event with the task for which it was performed and combine the different pieces of information to produce an accurate and detailed record of every task. In the previous paper [1], the focus was on using the information to measure effort. In this paper, we describe using the same information to generate developer views of artifact history and context.

IV. DATA SYNTHESIZING

We presented a set of 5 questions in Section I, the introduction. Now, let's look at each of the question in turn, to see how we can use the information we gathered from the repository to answer those questions.

A. *In what parts of the system does the artifact have a role?*

Parts of a system can be a physical group of parts or a conceptual function. An artifact has a role if changing the artifact's information has a direct impact on the correct behavior of that part of the system. There are two ways to answer the question of whether or not there is a role. One way to answer the question is to look at the neighborhood of artifacts that changed together with the artifact in question. A functional relationship can be inferred among artifacts that change together in a task that created or fixed a feature or behavior. For example, as can be seen in Figure 4, each of the three times that an artifact called ae_fctry.c was changed, the artifact called ae_memo.c was also changed. This history indicates that the correct functioning of ae_fctry.c is likely to depend on ae_memo.c, or vice versa. We can list out the co-changed artifacts along with the number of co-changes, and what fraction of all changes to either artifact that represents. In our display, we list all of the co-change artifacts, and provide simple counts of change and co-change occurrences to help the developer decide which ones are worth exploring further. In addition to providing context, co-change artifacts are places the developer should also look to make sure that any change is consistent with the artifact's role or roles and that historical relationships are not violated. In our example, a developer making changes to ae_memo.c should look for a possible effect on ae_fctry.c.

The other way to answer the question about an artifact's role is to look at the tasks for which the artifact was changed. We can infer that if the artifact had to be modified to achieve the goal of a task, the artifact must have some role in that task. For example, the artifact ae_memo.c was involved in 4 development tasks. For one of those development tasks, almost 10% (16/178) of the recorded change events were specific to ae_memo.c, even though 39 other artifacts were also involved. From the additional count information, we can infer that ae_memo.c plays a major role in the requirement that motivated that task. We can list out all of the tasks for which the artifact was changed. For development tasks, the developer can often trace back to the requirement that generated that task. Task tracking records often include a reference to the corresponding requirement. If the requirements documents are themselves kept as managed artifacts, changing the status of the requirement automatically generates an identifiable repository event. Similarly, for repair tasks, the developer can often find the change request, bug report, or failed test that motivated that task. The key contribution here is our ability to recover correlation keys between different sources of information (as described in two earlier papers [1] [3]). Connecting the task and artifact data greatly expands the options for exploration and navigation.

The second question in our original set is answered in the same way as for the requirements. Which tests revealed the reasons for altering the artifact? For confidentiality reasons, we did not include an artifact-to-tasks view. But the tabular and graphical versions look much like to those shown in the other figures.

B. *When was the artifact made?*

The version control system, by itself, contains a wealth of history about how and when the artifact changed and in what ways for different versions of the system. In our approach, the developer does not need to construct a query. The answer is a click away. The developer selects the artifact from a list and then sorts its history by date.

We add context by tying changes to the specific tasks involved. Sometimes the history of an artifact is confounded by its role in more than one system or product. In our data, every change is associated with a task. Reference to the tasks, and that tasks' associated products, can help clarify the context of its creation, as well as that of each successive change. In our example of ae_memo.c, the tasks are associated with at least two different products.

C. *How often has it been fixed or altered, and in each case, when, why, and by whom?*

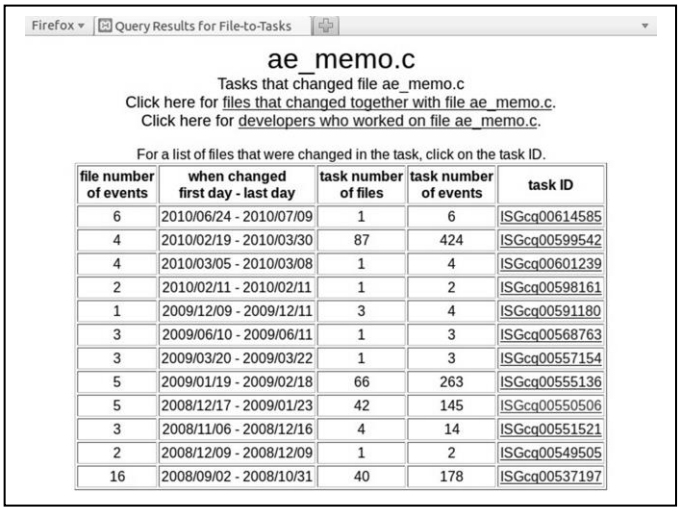
The frequency with which an artifact has changed says a lot about how it should be treated. Artifacts that change often are hot-spots in the system and should be handled with extra care. Our example artifact, ae_memo.c, as involved in 8 repair tasks, indicating that additional care, for example assigning a developer with more experience, is probably warranted. The nature of the earlier repairs should be considered. Only two of the repairs involved multiple artifacts. Those other artifacts should also be inspected. The discussion above already addressed questions about when and why.

We can also reveal by whom, and how much of their effort was devoted to this artifact. In our example of ae_memo.c, there were many different developers involved in changes to that artifact. In the system under study, the large number of developers touching one artifact is somewhat unusual, possibly indicating that its role in the system is more central than many of the other artifacts. In the figures shown below, we use numbers of recorded change events as a rough indicator of relative effort. We have experimented with other measurements of effort, and also with calibrating the event counts with each developer's behavior. (See our earlier paper for a discussion of effort. [1]) As a rough indicator, the simple metric is sufficient and consistent with a keep-it-simple approach.

When the person is no longer around, the history shows the other tasks and artifacts on which they have worked, which might provide some insight into their interests and expertise. Continuing with our example of the artifact ae_memo.c, one of the developers was a21729. Looking at the artifact history of developer a21729, we can tell from the number of requirements documents they have touched that they are fairly senior, and from the types of artifacts touched, that they often work on issues involving communications protocols. That might indicate something about the types of concerns they may have had to address, or conversely, an area where they are less likely than others to have made a mistake.

V. Information Display with Multiple Views

Synthesizing the information is half of our challenge. The other half is to present it in an interactive easy to use form. To achieve this goal, we provide two complementary views, one tabular and the other graphical. Both are interactive.



file number of events	when changed first day - last day	task number of files	task number of events	task ID
6	2010/06/24 - 2010/07/09	1	6	ISGcq00614585
4	2010/02/19 - 2010/03/30	87	424	ISGcq00599542
4	2010/03/05 - 2010/03/08	1	4	ISGcq00601239
2	2010/02/11 - 2010/02/11	1	2	ISGcq00598161
1	2009/12/09 - 2009/12/11	3	4	ISGcq00591180
3	2009/06/10 - 2009/06/11	1	3	ISGcq00568763
3	2009/03/20 - 2009/03/22	1	3	ISGcq00557154
5	2009/01/19 - 2009/02/18	66	263	ISGcq00555136
5	2008/12/17 - 2009/01/23	42	145	ISGcq00550506
3	2008/11/06 - 2008/12/16	4	14	ISGcq00551521
2	2008/12/09 - 2008/12/09	1	2	ISGcq00549505
16	2008/09/02 - 2008/10/31	40	178	ISGcq00537197

Figure 1. Tabular view of artifact task history.

Figure 1 shows a tabular view of an artifact's task history. The tasks are listed in chronological order starting from the most recent. For each task, the dates when the artifact in question was changed are shown along with the number of artifact change events. The table also shows the total number of artifacts changed and the total number of change events. The latter numbers enable the developer to gauge the scope of the task and the size of the artifact's role within that task. Although not shown, we can also color code the task IDs to distinguish between repairs and new development. In the example shown, the artifact was involved in several large development tasks, and also in many small repair tasks in which it was the primary focus of the work being performed. Clicking on the task ID allows the developer to explore all artifacts or developers involved in that task.

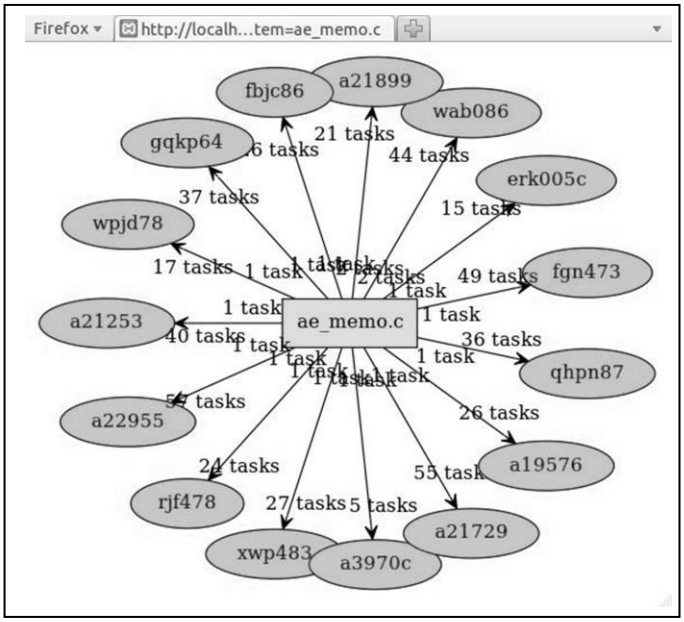


Figure 2. Graphical view of prior artifact developers.

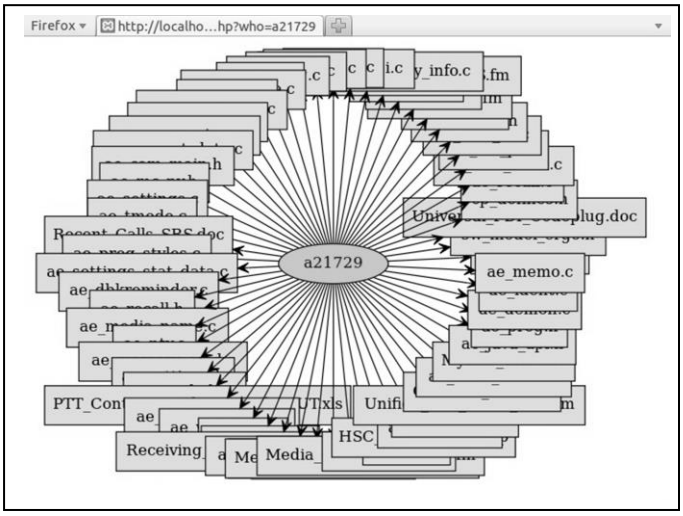


Figure 3. Graphical view of developer artifact history.

Figure 2 shows a graphical representation of the developers who have worked on that artifact, e.g. ae_memo.c shown in Figure 2. Graphical views are used primarily for navigation. In the example we added annotations for the frequency of interaction. The number closer to the artifact is the total number of tasks in which the developer worked on that artifact. The number closer to the developer is the overall number of tasks on which the developer is known to have worked. In practice, developers have expressed a preference for having views without the annotations. Such information, and more, is better shown in a tabular view. Developers can easily switch

between graphical and tabular views by clicking on the artifact's name in the center of the graphical view and at the top of the tabular view. Clicking on a developer brings up a view like that shown in Figure 3.

Figure 3 shows a graphical view of every artifact which a particular developer has changed. From the corresponding tabular view, one can also access the developer's task history. Although the current work is much better at providing focused views than our earlier effort, it is still possible to get graphical views where the forest hides the trees. We have experimented with sorting items clockwise and/or front to back, based on various criteria. But, as with the annotations, adding meaning within a graphical view was found less useful than simply switching to its tabular form.

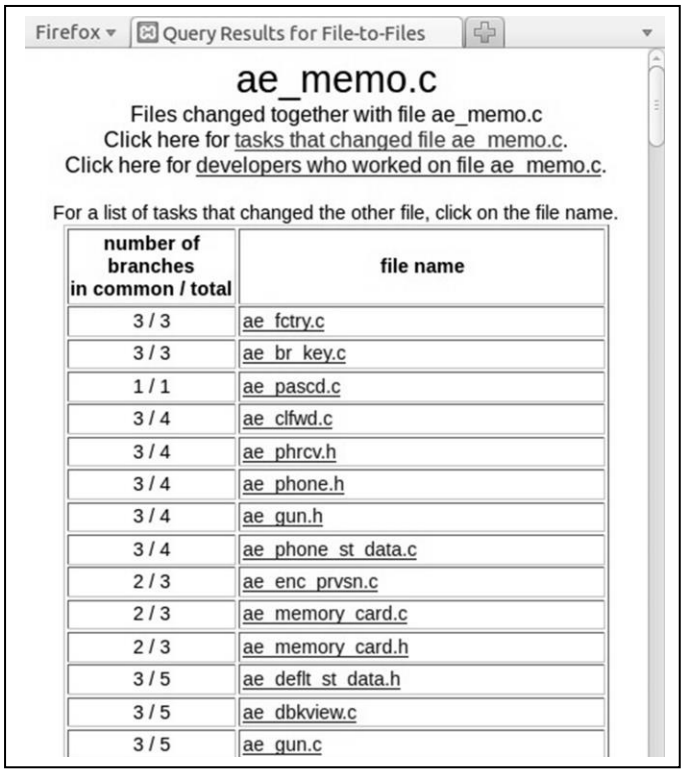


Figure 4. Tabular view of prior co-change artifacts.

Figure 4 shows a tabular view of the co-change artifacts. In this case, the table also shows the number of branches in which both artifacts were changed versus the total number of branches for just the co-change artifact. This give a sense of the frequency with which the two artifacts change together. In the example shown, the artifacts at the top always changed together with the artifact in question. Further down the list the relationship becomes less close.

The distinction between branch data and task data is subtle but significant. Tasks are managed units of work with a purpose and result directly tied to the system's production. In the version control repository, all tasks appear as branches. But there are additional branches that reflect exploratory work done by developers who are themselves trying to understand relationships within the system. Adding the non-task branches increases the overall amount of information. Since non-task branches are usually smaller and more focused than tasks, branch co-change is a tighter metric than simply task co-change.

In any view, clicking on any task, developer, or artifact brings up a new view where that item is the center of focus. Every tabular view includes additional options to select any other view in which the same item is the center of focus. In this way, any view of any item can be reached by simple navigation. Using relationships as the path of navigation facilitates and encourages meaningful exploration. In tabular views, clicking on a table heading sorts the table rows by the values in that column. In long tables, sorting can help in search. But even in a long list, items can be found just as easily with a standard page search operation.

VI. Implementation and Experience

Our tools and capabilities are used to perform navigation and exploration by managers and developers who are already busy. Thus it is important that they be highly interactive. It needs to be easy to navigate among views and to different parts of the system. New queries must be easily generated, ideally with a single click, with no text to remember and type. The more quickly and easily the user can explore the data, the more insights they can accumulate and consider.

Using Web technology made the implementation surprisingly easy. We used HTML with embedded PHP for display, query, and interaction. The PHP code queries the database and generates tables. With only a little extra code in the PHP script, items in the table are turned into hyperlinks. We used stored procedures in MySQL to simplify the PHP code and improve performance.

For the graphical views, the PHP-generated table rows are piped on the fly through the GraphViz application to generate SVG encoded graphics. Again, a small amount of decoration turns rendered objects into hyperlinks. By making everything a hyperlink, no additional text direction is needed for most forms of navigation and exploration.

The queries are run against a small number of tables where the data has been preprocessed to minimize the amount of work needed for each new query. For example, the various counts shown in the tabular views were pre-computed per developer, task, branch, and artifact, and stored in a special table. The table of counts makes it possible to show global counts, while performing queries only on the smaller amounts of local data needed to show relationships. Query time has a negative impact on user experience. Fortunately most queries complete in 3 seconds or less on modest hardware, even though the original repository had more than 30,000 events.

The entire system can be accessed online or, for sensitive projects, delivered on a thumb drive as a LAMP virtual machine. The deployment strategies are again consistent with our goal of minimizing costs to the user in terms of time, effort, or complexity. The keep-it-simple philosophy applies to all aspects of the user experience.

As explained earlier, when it was in use, developers showed a clear preference for using the graphical views to navigate. Thus, even though the tabular views show more information,

the simple graphical views play an important role in facilitating system and data exploration and the discovery of new insights. Developers seemed pleased with the ease and speed with which exploration was possible and genuinely interested in using the information that was learned.

Compared to our earlier efforts to provide similar views of the data, the new system is much easier to use [2]. The earlier system allowed the user to explore regions of a large graph presenting all relationships among artifacts, tasks, and developers. It presented a network view and, to manage density, supported filtering by time. It was best used for general queries about the system as a whole, and exploring hot spots. [4] The new views present only the relationships specific to a single artifact, developer, or task. Instead of the earlier network layout, the graphical views now use a radial, hub-and-spokes layout. With the focus on relationships to a central entity, the tables and graphs are now manageable in size, eliminating the need to filter by time. Interaction is also very fast. As a result, developers have shown much more interest.

VII. Conclusion and Future Work

We presented our work on using repository mining to assist developers making changes and additions to large and complex software systems. The focus was on presenting simple objective information to augment the developers' view of system artifacts on which work will be performed, and providing a context map that is easily navigated. The intended use is to provide valuable insights such as identifying relevant sources of information from the artifact's prior history, to identify other artifacts of the system that should be inspected at the same time, and to provide relative indicators of the likelihood that work on a given artifact will lead to additional problems.

The work was carried to a level where it is usable by actual managers and developers. The initial response has been favorable. In this paper we presented a working example, with different views of the type of information we are able to synthesize and the types of insight that these views support. A novel aspect of our work is that navigation and exploration is supported entirely by clicking on objects in the display with no need to input text, select a filter, or construct a query.

Our work differs from much of the academic work in that it is intended for use on common industry repositories rather than the peculiarities of open-source data, and focuses on the challenges of usability by end users rather than fine grained statistical analysis. Validation comes from the system being accepted for use by everyday developers.

The work is part of a larger project to extract knowledge from existing sources of information in a manner that is non-invasive to developers and managers, and provides valuable views for a variety of development and management activities. Future work will include extension and further validation of the existing tools, simple techniques to increase the quality and value of repository data, and creating additional views of the repository data for project managers.

References

[1] M. Van Hilst, S. Huang, J. Mulcahy, W. Ballantyne, E. Suarez-Rivero and D. Harwood, "Measuring Effort in a Corporate Repository," in *Proceedings of the 12th IEEE International Conference on Information Reuse and Integration (IRI 2011*, Las Vegas, 2011.

[2] S. Huang and C. Lo, "Analyzing Configuration Management Repository Data for Software Process Improvement," in *Proceedings of the 19th International Conference on Software Engineering and Knowledge Engineering (SEKE 2007)*, Boston, 2007.

[3] M. VanHilst, S. Huang and H. Lindsay, "Process Analysis of a Waterfall Project Using Repository Data," *International Journal of Computers and Applications,* vol. 33, no. 1, 2011.

[4] S. Huang, "Assessing Software Process Hotspots via Analysis and Visualization of Software Repository Data," *Journal of Engineering, Computing and Architecture,* vol. 3, no. 1, 2009.

[5] S. Huang, M. Van Hilst, S. Tilley and D. Distante, "Adoption-Centric Software Maintenance Process Improvement via Information Integration," in *13th IEEE International Workshop on Software* Technology *and Engineering Practice*, 2005.

[6] M. Lehman, "Laws of software evolution revisited," in *Proceedings of the Workshop on Software Process Technology (EWSPT'96)*, Nancy, 1996.

[7] G. Canfora and L. Cerulo, "Impact analysis by mining software and change request repositories," in *Proceedings of 11th IEEE International Symposium on Software Metrics*, 2005.

[8] T. Zimmermann, A. Zeller, P. Weissgerber and S. Diehl, "Mining version histories to guide software changes," *IEEE Transactions on Software Engineering,* vol. 31, no. 6, pp. 429-445, 2005.

[9] S. Grant, J. Cordy and D. Skillicorn, "Using topic models to support software maintenance," in *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR 2012)*, 2012.

[10] J. Lawrence, R. Bellamy, M. Burnett and K. Rector, "Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI 2008)*, 2008.

[11] R. Dyer, H. Nguyen, H. Rajan and T. Nguyen, "Analyzing ultra-large scale code corpus with Boa," in *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity (SPLASH 2012)*, 2012.

[12] L. Voinea and A. Telea, "Visual querying and analysis of large software repositories," *Empirical Software Engineering,* vol. 14, pp. 316-340, 2009.

[13] A. Bradley and G. Murphy, "Supporting software history exploration," in *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR '11)*, 2011.

[14] T. Fritz, G. C. Murphy and E. Hill, "Does a programmer's activity indicate knowledge of code?," in *Proceedings of the Joint European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE'07)*, Cavtat, 2007.

[15] T. Fritz and G. Murphy, "Using information fragments to answer the questions developers ask," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE '10)*, 2010.