



Institute for Software Research
University of California, Irvine

Initial Successes and Failures Prototyping Socio-technical Visualizations Using a Collaboration Infrastructure



Erik Trainer
University of California, Irvine
etrainer@uci.edu



David F. Redmiles
University of California, Irvine
redmiles@ics.uci.edu

November 2010

ISR Technical Report # UCI-ISR-10-5

Institute for Software Research
ICS2 221
University of California, Irvine
Irvine, CA 92697-3455
www.isr.uci.edu

www.isr.uci.edu/tech-reports.html

Initial Successes and Failures Prototyping Socio-technical Visualizations Using a Collaboration Infrastructure

Erik H. Trainer and David F. Redmiles
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3425
{etrainer, redmiles}@ics.uci.edu

ISR Technical Report #UCI-ISR-10-05

November 2010

Abstract:

Empirical studies, including our own, have concluded that dependencies between source-code modules create coordination needs among developers working on those modules. As described in our previous work, we developed Ariadne, a dependency plug-in infrastructure for the Eclipse IDE and later, Jazz. The decision to extend Eclipse's plug-in model was largely based on the extent of services it provides client plug-ins, from dependency analysis to user-interface widgets, to back-end resource modeling and storage mechanisms. As such, we could architect Ariadne to be general enough to support different programming languages, CM repositories, and visualization representations.

In this work, we report on the choices that drove the integration of our tool with Eclipse, some of the problems we faced prototyping visualizations, and some suggested improvements to both the infrastructure and research tools based on our experiences. We also report for the first time, our experiences integrating data provided by IBM's new software development collaboration platform, Jazz, into Ariadne.

We conclude that although the Eclipse and Jazz infrastructures provide powerful APIs and interfaces for accessing data representative of collaboration activities, current support in the IDE for *socio-technical visualization on top of that data* is inadequate for several reasons. These observations serve as an initial step toward bridging research on visualization with data mined from collaboration infrastructures.

Initial Successes and Failures Prototyping Socio-technical Visualizations Using a Collaboration Infrastructure

Erik H. Trainer
David F. Redmiles

Institute for Software Research
Donald Bren School of Information and Computer Science
University of California, Irvine
etrainer@ics.uci.edu
redmiles@ics.uci.edu

Introduction

The current wave of software engineering tools that has been emerging of late rely on the automatic collection and sharing of information to prepare developers for, and assist in, handling a host of increasingly complex coordination issues, of which developers are generally aware (Grinter 1995; de Souza et al. 2003), but do not always anticipate, including: identifying, providing a "project-level" view for managers to gauge team progress, recognizing and resolving conflicts due to parallel work, and recommending relevant artifacts for development tasks such as evolving code, triaging bugs, and understanding a code base (Trainer and Redmiles 2009). These issues derive from the complex technical and social inter-relationships that emerge as software is designed, implemented, and maintained by software development teams: dependencies between the tasks that share resources, and necessarily between the developers assigned to complete those tasks.

One way to more deeply understand these social and technical, or *socio-technical*, relationships is to examine dependency relationships between developers that originate from the interdependencies between common development activities. Empirical studies, including our own (Morelli, Eppinger et al. 1995; Grinter 2003; de Souza et al. 2004; Cataldo et al. 2006) have provided a basis for the claim that dependencies between source-code modules in a call-graph create dependencies between the developers implementing those modules. We found that, to some extent, developers are aware of these interdependencies--but lack adequate tool support.

In response, we developed a set of scenarios (de Souza et al. 2007) based on field data collected in cooperation with distributed teams at a major corporation that represent more informal, yet critical aspects of development such as finding developers who use the same modules and staying aware of evolving dependencies in order to more smoothly

pace coordination efforts. Although this information is available in the change management system, it is not easily found. For example, consider the case of finding which developers program against a particular interface. Looking at the commit information for the interface allows one only to identify who checked it in, not who actually uses it, which would instead require an exhaustive search of who was involved in commits of code that call methods defined by the interface in question. It is not difficult to conclude that automating this process by way of tool support can significantly speed up such queries. These observations drove key functional design requirements for Ariadne.

A dependency-centric perspective, from which one navigates the socio-technical “maze” of a software project is the concept embodied in our tool, Ariadne. Ariadne is a dependency management infrastructure for exploring the design space of socio-technical visualizations based on information about dependent artifacts and authorship activity contained in a project’s change management repository. Since 2005, we have maintained different versions of the tool in the form of Eclipse (Trainer et al. 2005) and more recently, Jazz, plug-ins. This paper explores our experiences visualizing socio-technical relationships with data mined from traditional development environments such as Eclipse, and more collaborative development environments (Booch 2003) such as Jazz.

In the following sections, we describe requirements for Ariadne itself, the features the Eclipse and Jazz infrastructures provide, and how we traded off these features to prototype our new socio-technical visualizations. We conclude with a discussion of these trade-offs and how they may be managed in a collaborative development environment.

Ariadne’s Requirements

There were three high-level requirements for Ariadne in the early stages of design. First the tool needed to be launchable within, rather than outside the workspace. It needed to have the same “look and feel” as other views in the workspace so that it would not distract from other development activities occurring within the IDE.

Second, more crucially, it needed to extract artifacts and authorship information for those artifacts from data stores associated with the development workspace. In particular it needed to define interfaces to CM repositories and source-code call-graphs. We reported in our earlier work (Trainer et al. 2005) that source-code is just but one type of artifact that can be modeled with a socio-technical network. In a later section of this paper, we show how we extended Ariadne to accept *work items*, or tasks, as another type.

Third, Ariadne needed to provide different visual representations of a project’s socio-technical network. Ultimately, we found existing graphics libraries to be insufficient and eventually experimented with different visualization frameworks. However, this led us outside the Eclipse infrastructure, requiring us to give up a certain amount of integration with existing views in the workspace. In the next sections we enumerate characteristics of the infrastructure that shaped the design of Ariadne and the socio-technical visualizations it creates.

Eclipse’s Architecture: Plug-in Model

Ariadne has taken the form of an Eclipse plug-in since its initial implementation (Trainer et al. 2005). This decision was driven by Eclipse’s large user base and its well-established corpus of user-generated plug-ins at the time (some developed specifically for academic research, e.g. Mylyn (<http://mylyn.net>). The platform has achieved a critical mass of users, and its user base represents the target audience of Ariadne, i.e. software developers, facilitating deployment and evaluation of the tool. Developing Ariadne as an Eclipse plug-in not only makes it easy to distribute in the form of a compact download, but makes it possible for developers, including ourselves, to extend the tool by implementing additional plug-ins such as source-code generators and visualizations that can be used to analyze and explore projects in specific languages using different visual representations.

Ariadne integrates different components according to a layered architectural style. It consists of a core dependency management plug-in and several client plug-ins that extend the Eclipse development platform: dependency generation, authorship annotation, and visualization (Figure 1).

A core plug-in manages the linking of dependency generation and authorship annotation plug-ins to create a social dependency graph consisting of dependencies between developers based on the artifacts for which they have completely, or partially authored (Trainer et al. 2005; de Souza et al. 2007). The core plug-in defines listeners that respond to callback functions from available visualization plug-ins and subsequently displays the data using the specified visual representations.

Essentially, dependency generation plug-ins construct static call-graphs of source-code currently in the workspace for the source-code language each generation plug-in is defined to accept, authorship annotation plug-ins extract who authored what lines of source-code (and in the case of the Jazz SCM, who is involved in the evolution of work items) for the types of repositories they are defined to analyze, and visualization plug-ins provide graphical interfaces to the socio-technical relationships that developing source-code creates.

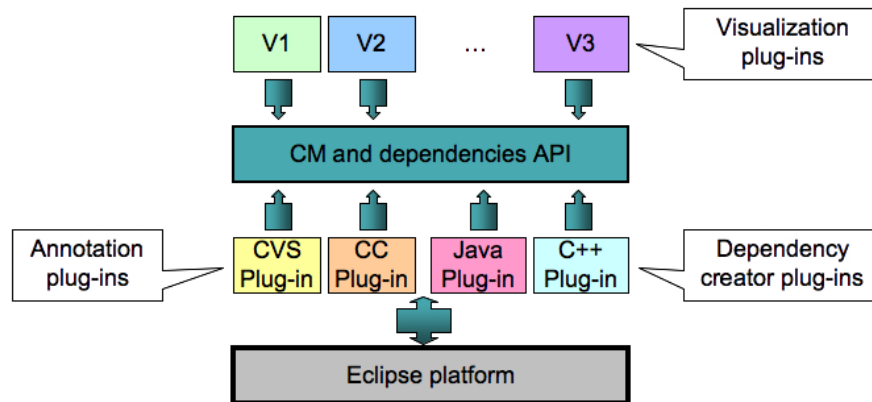


Figure 1. Ariadne’s layered architecture consists of three sets of plug-ins (dependency generation, authorship annotation, and visualization) coordinated and accessed by a core plug-in, the “CM and dependencies API.”

Because the CM and dependencies API can be accessed regardless of the implementation type of each client plug-in, Ariadne is general enough to support various programming languages (e.g., Java, C++ or both), CM repositories, and forms of

visualization. By default, we have provided Java dependency generation, CVS, SVN, and Jazz SCM authorship extractors, social network node-and-edge visualizations (de Souza et al. 2007), and more recently a novel TableGraph visualization (Trainer et al. 2008). Each piece of this functionality was implemented as a plug-in of the corresponding type and integrated into the tool with minimal effort. We believe this to be attributed to the extensibility of Ariadne’s architecture.

Infrastructure APIs

In addition to the plug-in mechanisms provided by Eclipse, the wealth of available APIs lowered the barrier to developing critical pieces of Ariadne’s functionality. For example, we were able to reuse the JDT SearchEngine to calculate dependencies between Java source-code modules (i.e., packages, classes, and methods) as well as the Eclipse Team APIs to hook into different types of CM repositories such as CVS and SVN to determine authorship for the modules. Other hooks into different GUI components such as “perspectives,” “views” and dialogs made it easy to give Ariadne’s Graph Editor and display windows and input forms the same “look and feel” as the ones in Eclipse. Additionally, we were able to reuse the Eclipse Resource Model data structures to access source-code modules and developer attributes, as well as import and export our dependency graphs into the workspace and display them in a custom editor.

Even a good infrastructure will not always provide the APIs and mechanisms tool building requires. Sometimes certain services are not offered because their design has not been finalized, and, as such are subject to frequent change (de Souza et al. 2004). In the design of our CVS authorship plug-in, we had to use unpublished Team APIs provided by the Eclipse team that ended up changing later and breaking our code as a result.

Collaborative “Traces”

Collaborative development environments (Booch 2003) such as Eclipse and Jazz provide traces of collaborative activity, such as lists of developers and the files they modify and when they modify them, as well as records of the work to be completed (and who should be involved) as defined in process descriptions. These “collaborative traces” imply relationships among the developers assigned and/or responsible for completing tasks.

In our previous work (de Souza et al. 2007), we described using source-code modules and authorship as “socio-technical” collaborative traces to infer coordination dependencies between developers. In our newest work using the Jazz environment and an anonymized data set from IBM, we extended Ariadne’s authorship annotation plug-in to support authorship extraction from the Jazz SCM, and more critically, added *work items*, descriptions of tasks, to the list of collaborative traces Ariadne is defined to accept.

Along the way, we observed that the default visualizations on top of work item data in Jazz do not support good socio-technical exploration because they 1) rely on developers to manually associate themselves with work items beyond what is included in the initial description of the work item and 2) they only show dependencies between work items themselves, omitting developer-developer relationships that arise from relationships between work items, source-code, and the developers involved. This

observation partially motivates the decision to explore socio-technical visualizations beyond what is offered by the infrastructure.

The following section presents these particular experiences and observations in greater detail.

Work Items in Jazz

A *work item* is a generalized notion of a development task, such as a bug, a feature enhancement, or a refactoring (Figure 2). It provides a high level description of work to be completed, who is responsible, and, sometimes, how it should be done. Work items are added to a project's specification either by project managers or developers themselves.

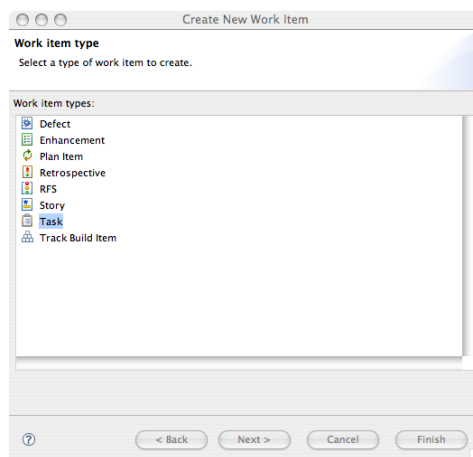


Figure 2. Work item types.

Once a work item has been filled out in the Jazz wizard, it is viewable within the Jazz Work Item Editor. The editor consists of several pages:

- Overview – main attributes, summary of the work to be done
- **Links – file attachments, subscribed users, other artifacts**
- Approvals – reviews, verifications defined for the work item
- Custom – custom attributes defined in process specification
- History – history of work item changes

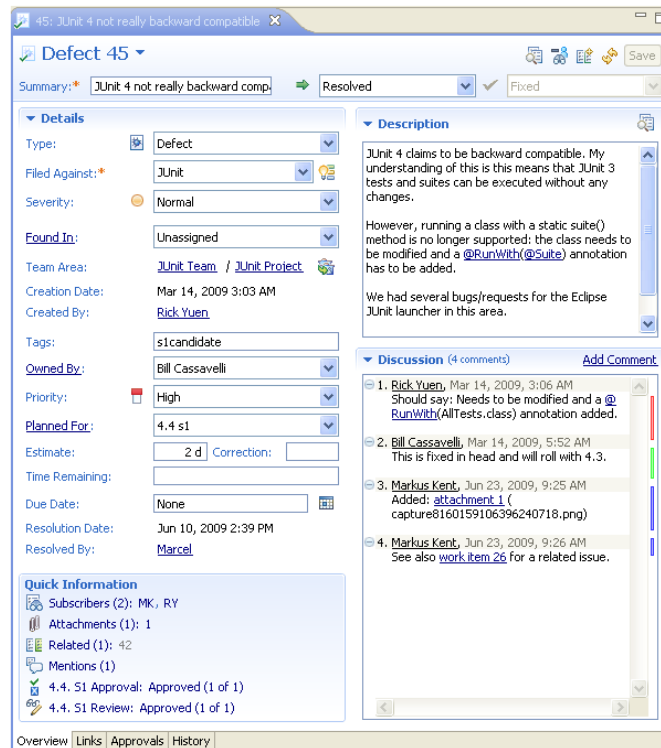


Figure 3. The Work Item Editor.

Links indicate relationships between work items and other artifacts, such as **source-code change-sets** via the *related* link and other work items via the *similar* and *depends on* links, as well as relationships between developers and work items, via the *subscriptions* link. Developers add links using an editor accessible from the links tab (Figure 4). In other words, developers themselves must make the conscious decision to subscribe to work items might involve files with which they work. By extension, they must determine whether one work item is dependent on another without automated tool support. As the number of work items grows, it is not difficult to see that this manual approach does not scale. In response, we reflected on how to automate this process.

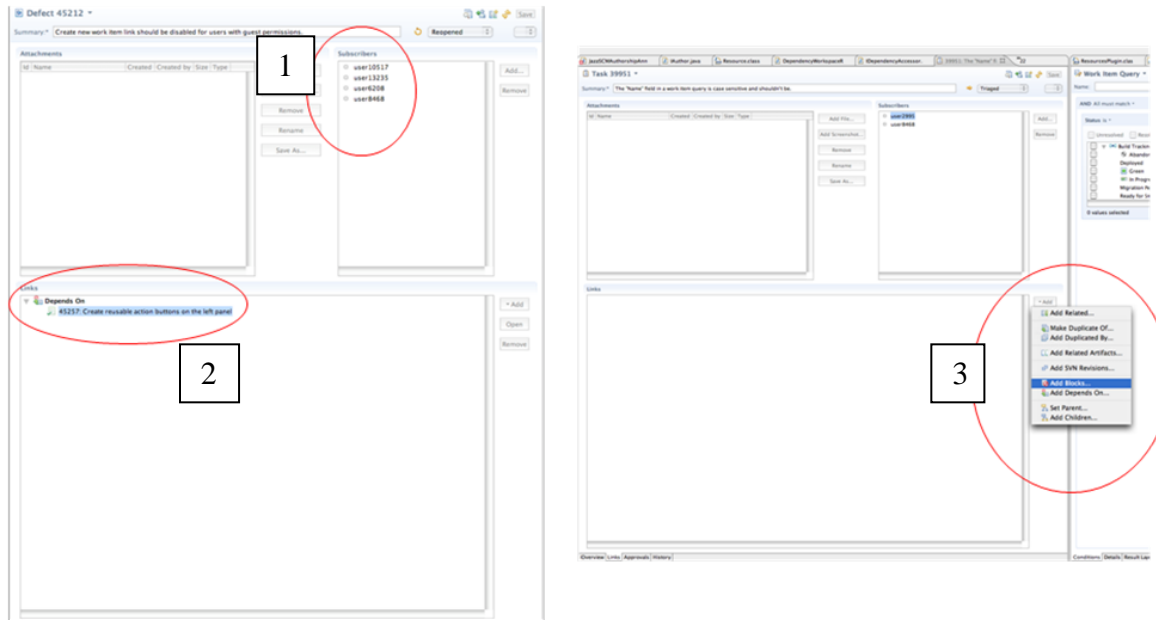


Figure 4. Subscribers (1), depends on (2), and relationship types (3) for a Work Item.

In the same way that source-code authorship distributed among dependent code implicitly creates dependencies between developers, work items that involve dependent source-code modules create dependencies between developers working on that source-code. This is especially important because developers who have not explicitly added themselves to receive updates about certain work items will not be able to account for eventualities that arise from dependent work items that have not been addressed, or are blocking the completion from specific development tasks. This observation led us to conclude that in addition to source-code, work items should be added to the Ariadne infrastructure as another “collaborative trace.”

More precisely, we define work item a to be dependent on work item b if:

- The “depends on” (a, b) relationship has been set in a ’s “links” page to point to b ; or
- A has one or more modules in its “related” link that are also included in b ’s “related” link.

In addition, we say that if a developer is listed in work item a ’s “subscribers” link, that developer has “authorship” associated with a . “Authorship” in this sense is really just a proxy for a *relationship* to the work item, whatever that relationship may be. In turn, developers who subscribe to work items dependent on a will depend on the aforementioned developer.

Figure 5 and 6 below show a socio-technical network (Figure 5) and social dependency network (Figure 6) that illustrate this concept. One can see that the work item “externalize strings” involves user13874 and is dependent on “Work items:

externalize strings,” which involves user1976. The resulting social dependency network illustrated in Figure 6 shows this underlying dependency:

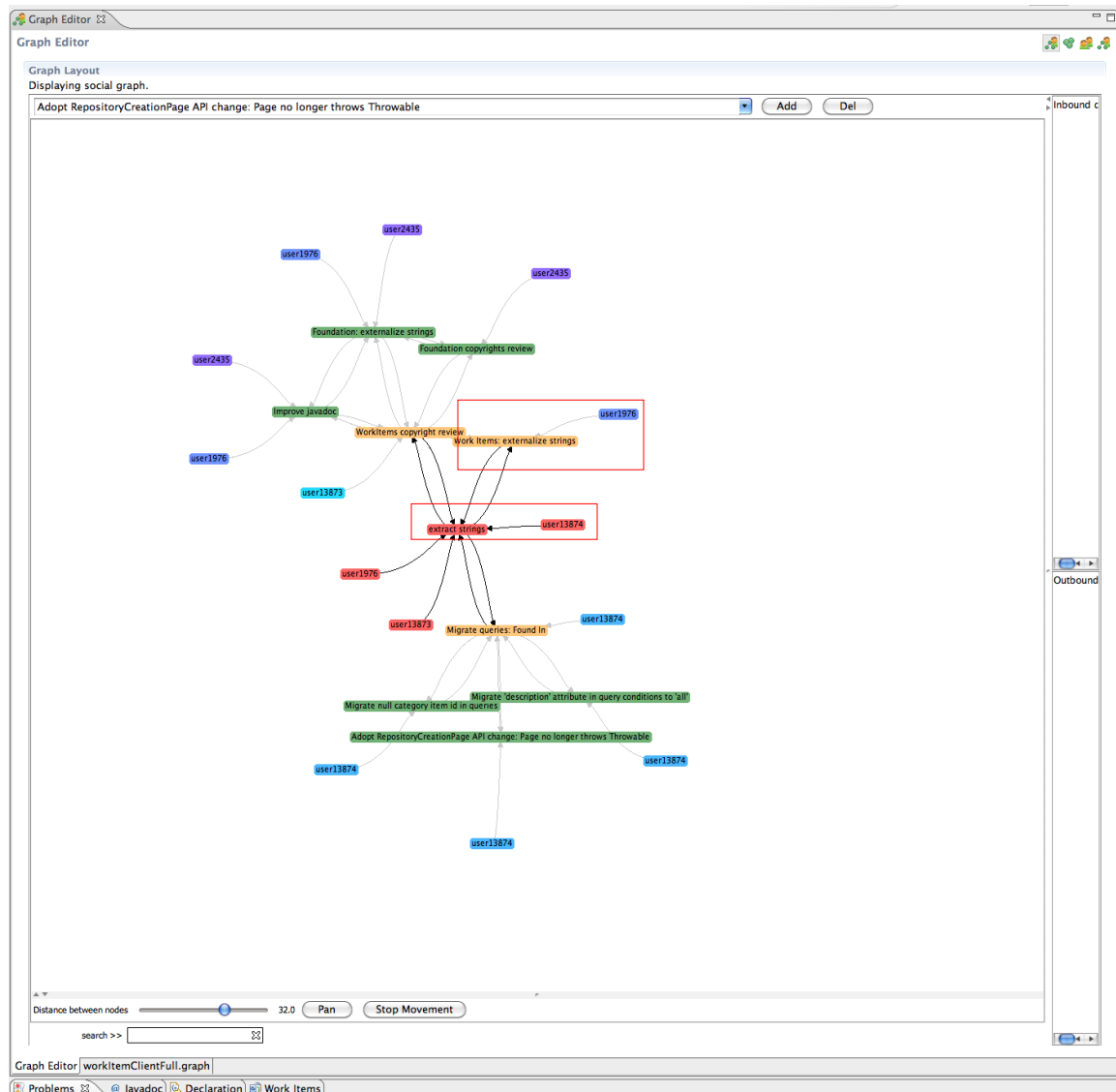


Figure 5. Socio-technical network displaying Work Items annotated with developers.

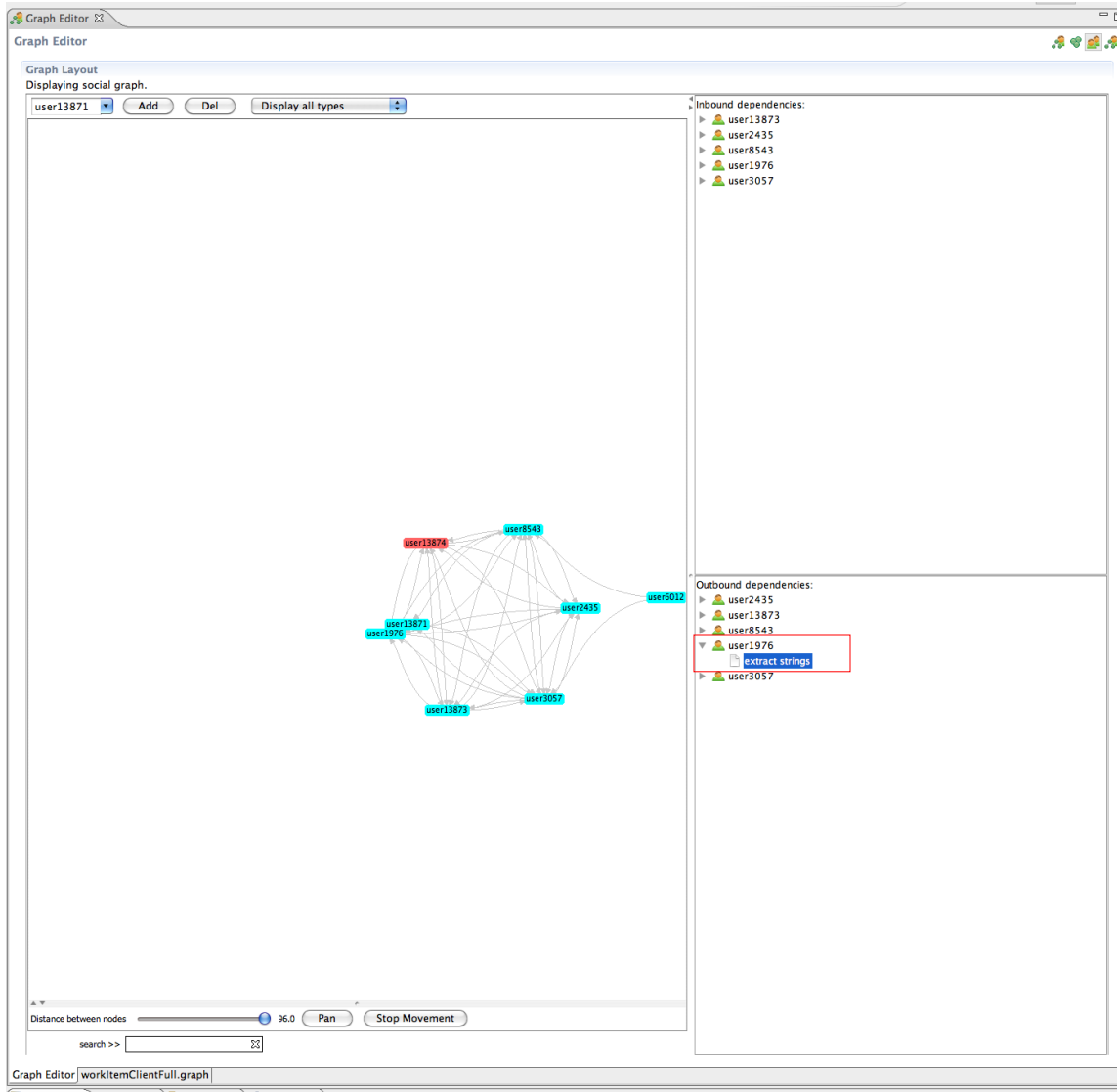


Figure 6. Social dependencies showing dependencies between the developers highlighted in Figure 5, specifically underlying work item “extract strings.”

Visual Representations

After integrating work items into Ariadne’s data storage layer, we looked at ways of visualizing socio-technical relationships using several different types of representations. We subsequently identified two factors that influenced our visualization decisions: *scalability* of the representations and *variability* in both the representations’ vocabulary and layout.

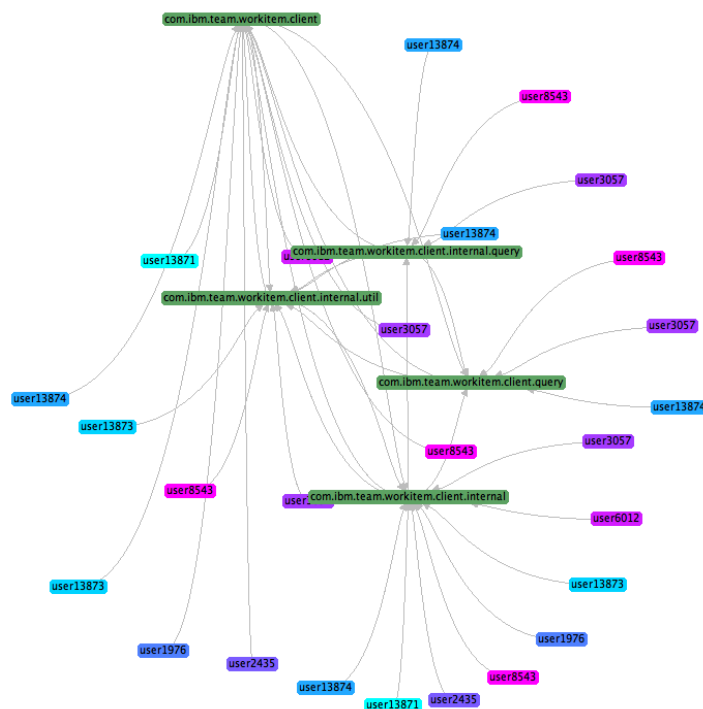
The first visualizations we implemented were fully integrated into the Eclipse editor, by extending the Eclipse UI views with a node-edge visualization. Figure 5 shows such a visualization of call-relationships between modules (and work items) of a system annotated with nodes corresponding to the developers of those modules. This particular

visualization was created with JUNG (<http://jung.sourceforge.net>), a software library that provides interfaces for modeling, analyzing, and visualizing data that can be represented as a graph or network. However, we had problems with respect to the variability of the layout and representations as well as scalability of the visual elements included in the JUNG tool.

Due to the variability of common graph layouts such as the Force Directed Layout, in which vertex placement is determined by a hypothetical physical process usually incorporating attraction between adjacent vertices balanced by a tendency for vertices to repel each other (Fruchterman and Reingold, 1991; Kamada and Kawai, 1989), it was difficult to compare visualizations of the same project (Figure 7).

There also existed little variability in the representations provided by the visualization package. JUNG is inherently limited to node and edge representations. In response, we looked at some of the graphic support in the Eclipse APIs such as the GEF framework, but its limited set of representations such as boxes and arrows were too general for our purposes, and proved to be no more helpful than node and edge diagrams in showing entities and relationships.

Moreover, as the size of the project increases, or if one tries to visualize the whole project in order to put one's activities in the context of others', the visualizations become increasingly cluttered and individual nodes and labels are hard to recognize at all (Figure 8). Therefore, ensuring the scalability of node-edge representations at the project level became problematic.



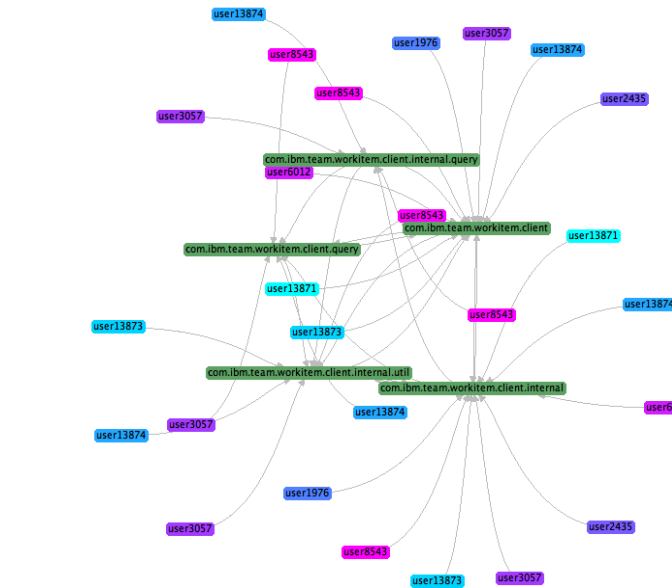


Figure 7. Layout algorithms produce different layouts for the same graphs.

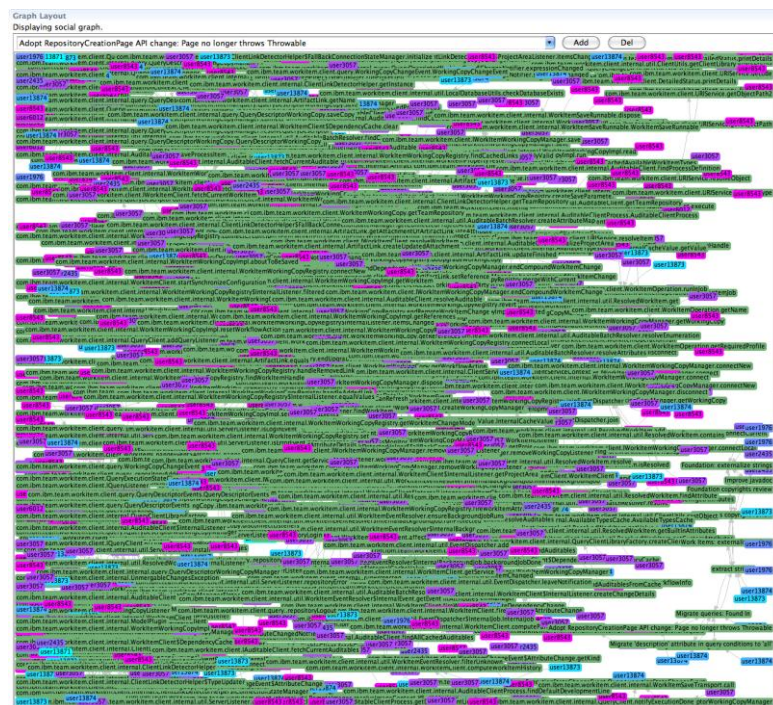


Figure 8. Node and edge visualizations do not scale at the project level (as illustrated by the illegibility of this figure).

Scalability and variability were also issues in visualizations provided by the Jazz infrastructure itself. Querying work items for important attributes is done one-at-a time in an incremental fashion by chaining SQL-like statements together (Figure 9).

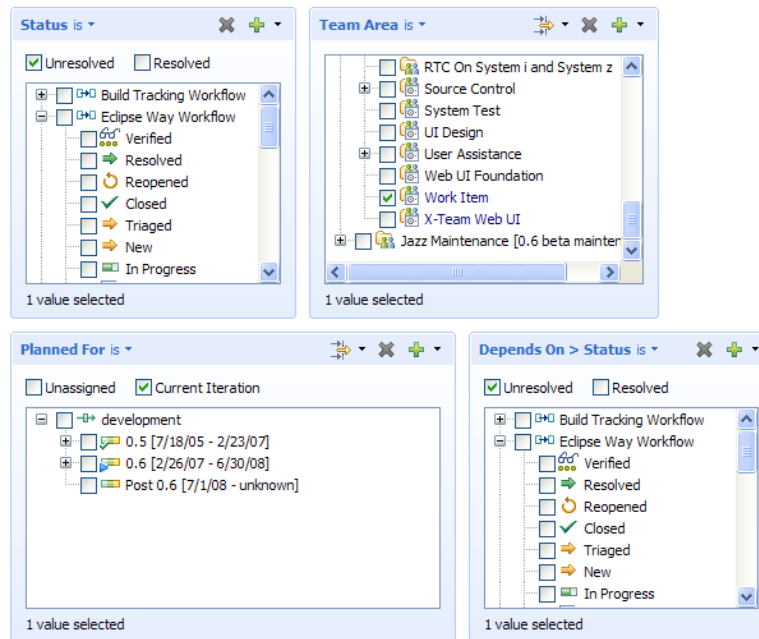


Figure 9. Example Query: “Find all unresolved work items in your team area that are planned for the current iteration and depend on another unresolved work item.”

Work Items							
Found 11 work items - Assertion syntax							
Summary		Status	P	S	Owned By	Created By	
Provide improved Assertion syntax	228	Idea			Bill Cassavelli	Bill Cassavelli	
Children							
Implement new assertThat	226	New			Markus Kent	Bill Cassavelli	
Children							
Write test cases for assertThat	225	New			Markus Kent	Markus Kent	
Specify new assertThat syntax	224	New			Markus Kent	Markus Kent	
Parent							
Based on the assertThat syntax we should provide ass	227	New			Markus Kent	Bill Cassavelli	
Based on the assertThat syntax we should provide assumption	227	New			Markus Kent	Bill Cassavelli	
Implement new assertThat	226	New			Markus Kent	Bill Cassavelli	
Write test cases for assertThat	225	New			Markus Kent	Markus Kent	
Specify new assertThat syntax	224	New			Markus Kent	Markus Kent	
Improve documentation for 4.4	223	Idea			Bill Cassavelli	Bill Cassavelli	
shows green bar while assert false	212	New			Markus Kent	Freddy Freund	
Duplicated By							
shows green bar while assert false	211	Resolved			Markus Kent	Freddy Freund	
shows green bar while assert false	211	Resolved			Markus Kent	Freddy Freund	

Figure 10. Results returned from a Work Item Query.

This search results in a visualization in the style of a JTreeViewer that lists the matching work items and relationships indicated by nesting in the tree (Figure 10). While this simple visualization scales to the number of objects one wants returned (by scrolling down the list), it is limited to displaying the results for one query within a limited space, instead of being able to answer or support the exploration of several queries at once, highlighted by the “overview first, zoom and filter, then details-on-demand” information mantra developed for information visualization (Shneiderman 1996). Therefore, while it scales to results returned, it does not scale to more than one *query* at a time.

Moreover, this visualization is limited in the variety of information it displays: it only shows work item dependencies represented by nesting within the tree. One cannot compare or see relationships between developers through the work items with which they work. The visualization does not display the source-code associated with the work items. It relies on the data explicitly entered by developers, omitting dependencies that can be automated using approaches like Ariadne’s dependency generation method. Last, the results are not returned in any pre-determined order, so the same query may generate a different nested tree layout each time.

Stepping Outside the Infrastructure

In response to problems of scalability and variability of existing representations and layouts, we experimented with a novel visualization using Prefuse, another third-party visualization toolkit. It gave us more freedom in coming up with different visual representations for dependency data, provided fast and powerful mechanisms and data indexing, and required little overhead in terms of learning. The lack of constraints on representations allowed us to create a constant layout and scalable view of the modules and authors at the project level by ordering modules on the horizontal axis while preserving the ease of identifying connections of varying intensities to nodes in a social network graph (Figure 11).



Figure 11. A Table View of dependencies between developers (y-axis) of the Jazz SCM subsystem based on called code (x-axis) dependencies they share.

However, integrating the visualization into Eclipse became problematic, partially because of cross-platform issues with SWT and Swing, but more importantly, because although the visualization scales to the project level, it does so at the cost of available screen real-estate. This resulted in having to display the visualization outside the workspace or on a second monitor, or on a set of monitors. Indeed, the example figure above shows the visualization on three 30” cinema displays. Moreover, navigating

abstract representations such as these come at a learning cost, in contrast to the familiar nested lists and row-column views that have traditionally been part and parcel of development environments.

Discussion

Software development, as practiced in industry, is often comprised of teams of developers and managers who are committed to a development process with rigorous procedures and rules. As such, many dependencies arise between developers based on dependencies in the source-code with which they work. Using Eclipse and Jazz, we were able to reuse critical mechanisms such as the dependency analysis and configuration management resources to develop an infrastructure and visual interface for analyzing and viewing dependencies.

Despite the ability to reuse the back-end storage mechanisms of the infrastructure, our quest for sufficiently scalable and variable representations of socio-technical relationships at the project level required us to go outside the infrastructure. To this end, the traditional views such as lists and trees were not sufficient. Although JUNG provided data structures that adequately modeled our artifacts, developers, and dependencies in a form suitable for visualization, the available renderings were too variable in their layout and did not scale to the project level to be useful for analysis. On the other hand, while Prefuse provided more variety in representations, and thus the ability to create representations that can scale to the whole project, it came at a cost of screen real-estate, leading us to launch the visualization outside the workspace itself.

One possibly more revealing interpretation of our work described so far is one that puts into question our assumption that socio-technical relationships should be viewed at the project level. The simplistic user interface elements in Eclipse and Jazz such as the Work Item Query wizard allow for more incremental and progressive exploration of dependencies in situations where developers prefer a bottom-up approach toward navigating socio-technical space in which they work (Ye et al. 2007).

Our work, however, has always been motivated by the observation that developers are interested in socio-technical relationships of which they are generally, but not always, aware. This is due in large part to the complexity of the entire system. Understanding the system at larger level allows one to put their own work in the context of others' and stay aware of activities occurring "off the radar" that may affect the developer down the line. Visualizing the system in its entirety is a way of accomplishing this goal.

We describe our experiences as indicative, to some extent, of a larger problem: trying to reconcile a platform for software development and project visualization at the same time. Socio-technical visualizations that scale up to a project are not part of typical development processes and workflows. In contrast, the visualizations available in environments such as Jazz are too limited in their expressiveness and their scalability.

We propose a visualization editor in Eclipse that can be used to rapidly prototype visualizations over workspace data that scale to the whole project. One possible way might be to provide a visualization abstraction layer between back-end Eclipse services and third party visualization frameworks. This way, researchers using the visualization

editor would have access to all the same types of services, i.e. the plug-in mechanisms, the interfaces and APIs, and the collaborative traces provided by Eclipse and Jazz, as well as a way to experiment with visualizations they know can be cleanly integrated into the workspace. The abstraction layer would allow developers to address scalability of representations, the variability of representations and layouts, and continue to use the plug-in model to facilitate deployment of the functionality. Ariadne's federated plug-in dependency management mechanism represents a first step in this process.

Conclusion

In our work to date, we have built Ariadne, a dependency infrastructure for the Eclipse and Jazz development environment. The plug-in paradigm used by the infrastructure allowed us to hook into many key services provided by the environment via APIs, but required us to implement some of our own services as well. This is not surprising.

The limitations of the infrastructure turned out to be more revealing than the advantages. Our experiences led us to move beyond the infrastructure, but we ran into limitations with the compatibility of third party tools. Yet these tools provide more variable and potentially scalable visualizations not offered by graphics libraries native to the collaboration platform. In contrast to the visualizations in the infrastructure that rely on manual bookkeeping and incremental querying to make sense of a project's socio-technical network, our visualizations aggregate collaborative traces such as source-code and tasks in conjunction with developer relationships to those traces in order to create more complete representations of the dependencies developers share.

Reflecting on these limitations led us to observe a gap between the scope of graphical support available in the APIs of a development platform's infrastructure and abstract visualization toolkits. We suggested that a bridging mechanism such as an abstraction layer might serve to better integrate an infrastructure's data models with a large variety of visualization representations that either scale or do not scale to the project level.

Our work serves as a stepping stone for researchers wishing to integrate their novel visualizations with the state-of-the art. Moreover, they can improve upon the suggestions and tradeoffs with respect to visualizations for development environments discussed here.

Acknowledgements

This research is supported by the U.S. National Science Foundation under grants 0943262 and 0808783, and by an IBM Jazz Innovation Award. The authors gratefully acknowledge past and present collaboration with Cleidson de Souza and feedback from our research group member, Roberto Silveira Silva Filho.

References

- Booch, G. and Brown, A.W. (2003): 'Collaborative Development Environments', *Advances in Computers*, vol. 59, pp.1-27.
- Cataldo, M., Wagstrom, P. A., Herbsleb, J. D., and Carley, K. M. (2006): 'Identification of coordination requirements: implications for the Design of collaboration and awareness tools', *ACM Conference on Computer Supported Cooperative Work (CSCW)*, November 2006, pp. 353-362.
- de Souza, C. R. B., Redmiles, D. F., Dourish, J. P. (2003): 'Breaking the Code", Moving between Private and Public Work in Collaborative Software Development', *ACM International Conference on Supporting Group Work*, November 2003, pp. 105-114.
- de Souza, C. R., Redmiles, D., Cheng, L., Millen, D., and Patterson, J. (2004): 'How a good software practice thwarts collaboration: the multiple roles of APIs in software development', *ACM SIGSOFT Symposium on Foundations of Software Engineering*, November 2004, pp. 221-230.
- de Souza, C. R., Quirk, S., Trainer, E., and Redmiles, D. F. (2007): 'Supporting collaborative software development through the visualization of socio-technical dependencies', *ACM Conference on Supporting Group Work*, November 2007, pp. 147-156.
- Fruchterman, T.M.J. and Reingold, E.M. (1991): 'Graph Drawing by Force-directed Placement', *Software - Practice and Experience*, vol. 21, no. 11, November 1991, pp. 1129-1164.
- Grinter, R. E. (1995): 'Using a Configuration Management Tool to Coordinate Software Development', *ACM Conference on Organizational Computing Systems (COOCS '95)*, August 1995, pp. 168-177.
- Grinter, R. E. (2003): 'Recomposition: Coordinating a Web of Software Dependencies', *Journal of Computer Supported Cooperative Work*, vol. 12, no. 3, pp. 297-327.
- Kamada, T. and Kawai, S. (1989): 'An algorithm for drawing general undirected graphs', *Information Processing Letters*, vol. 31, no. 1, April 1989, pp. 7-15.
- Mockus, A. and Herbsleb, J. D. (2002): 'Expertise browser: a quantitative approach to identifying expertise', *International Conference on Software Engineering (ICSE)*, May 2002, pp. 503-512.
- Morelli, M. D., S. D. Eppinger, et al. (1995): 'Predicting Technical Communication in Product Development Organizations', *IEEE Transactions on Engineering Management*, vol. 42, no. 3, February 1995, pp. 215-222.
- Shneiderman B. (1996): 'The eyes have it: A task by data type taxonomy for information visualization', *Proceedings of IEEE Workshop on Visual Language*, September 1996, pp. 336-343.
- Trainer, E., Quirk, S., de Souza, C. R. B., Redmiles, D.F (2005): 'Bridging the Gap between Technical and Social Dependencies with Ariadne', *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology Exchange*, pp. 26-30.
- Trainer, E., Quirk, S., de Souza, C.R.B., and Redmiles, D.F. (2008): 'Analyzing a Socio-Technical Visualization Tool Using Usability Inspection Methods', *IEEE Symposium on Visual Languages and Human Centric Computing*, September 2008, pp. 78-81.
- Trainer, E. and Redmiles, D.F. (2009): 'A Survey of Visualization Tools that Promote Awareness of Software Development Activities', *UCI ISR Technical Report #UCI-ISR-09-05*, December 2009.
- Ye, Y., Yamamoto, Y., and Nakakoji, K. (2007): 'A socio-technical framework for supporting programmers', *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, September 2007, pp. 351-360.