MANHATTAN: Supporting Real-Time Visual Team Activity Awareness

Michele Lanza, Marco D'Ambros, Alberto Bacchelli, Lile Hattori, Francesco Rigotti
REVEAL @ Faculty of Informatics - University of Lugano, Switzerland

***Abstract*—Collaboration is essential for the development of complex software systems. An important aspect of collaboration is team awareness: The understanding of the activity of others that provides a context for one's activity. We claim that the current IDE support for awareness is inadequate: The typical setting is to rely on software configuration management systems (SCMs), which are based on an explicit check-out/check-in model. If developers rely only on SCMs information, they become aware of concurrent changes only when they commit their code to the repository. This generates problems such as complex merging and redundant work.**

Most tools to raise awareness notify developers of emerging conflicts in the form of textual notifications. We propose to improve the notification by using real-time visualization integrated in the IDE to notify developers of team activity. Our approach, implemented in a tool called MANHATTAN, eases team activity comprehension by relying on a city metaphor. MANHATTAN depicts a software system as a live city that changes as the underlying system evolves. Within the city, MANHATTAN renders team activity information, updating developers in real-time about changes implemented by the entire development team. Further, MANHATTAN provides programmers with immediate feedback about emerging conflicts in which they are involved.

I. INTRODUCTION

Team collaboration is an important aspect for successful delivery of software systems [4]. A crucial aspect of collaboration is *workspace awareness* [5]. In software development, workspace awareness is the means by which team members can become aware of the work of others that potentially impacts their own work [3]. Awareness is intrinsic to collocated teams and it is obtained through human interactions, such as meetings and informal conversations [10], [11]. However, in distributed teams, the lack of human interaction, caused by geographical distance, lowers awareness. Low awareness brings problems such as communication breakdowns [3], lack of willingness to help others, and production delays [10].

The software configuration management (SCM) community has put a significant effort to increase awareness of distributed teams by supporting coordination across multiple developers working in parallel [1], [2], [7], [9], [14]. These approaches promote *workspace awareness* by detecting in real-time potentially conflicting modifications to software artifacts.

Most of the proposed approaches and tools notify developers of emerging conflicts using textual notifications. We consider this approach to be not optimal, especially in the context of complex systems and cluttered IDEs. We argue that for a developer it can be very cumbersome to understand the conflict that is emerging by reading textual notifications, because of the number of elements involved in a conflict: whom, why, and how it can be addressed. We make an effort to overcome this problem, by proposing a real-time software visualization integrated in the IDE, as a means to notify developers of team activity. We devised and implemented an Eclipse plugin, named MANHATTAN [13], whose goal is twofold:

1) Help developers to reason on a system, by making it easier to understand its architecture and, as a consequence, help them to properly drive the system's evolution.
2) Support collaboration between development team members by increasing each member's awareness of the activity of the team.

MANHATTAN visualizes projects in the Eclipse workspace using the 3D city metaphor, devised by Wettel *et al.* in *CodeCity* [15], which exploits the similarities between software constructs and urban landscapes. The city metaphor gives a shape to the otherwise intangible software, exploiting the human capacity to build a visual mental model of the system, thus easing its comprehension. The true effectiveness of the city metaphor for program comprehension tasks has been demonstrated in a controlled experiment [15].

We reimplemented CodeCity as an Eclipse plugin to amend its major weak point: It is not connected to the development environment. Although this is not a problem when one wants only to understand an existing system, without the need to modify it, in a forward engineering setting this forces users to switch the context between CodeCity and the IDE. Also, if the user changes anything in the code, CodeCity does not automatically reflect it in the visualization, unless the user decides to re-import the code base into CodeCity.

MANHATTAN tackles this issue through its tight integration in an IDE. It visualizes any Java system whose code base is loaded into the Eclipse IDE, keeping the code base and the visualization synchronized. Moreover, MANHATTAN improves team activity awareness by showing to developers a living city where changes from team members and potential conflicting code are animated with different colors and shapes. To exhibit team activity, MANHATTAN relies on APIs provided by the *Syde* plugin for Eclipse [8]. Merge conflicts can be examined in the *Eclipse*'s Compare Editor opened from the visualization.

Structure of the paper: In Section II, we present our real-time visualization of software system as cities within the Eclipse IDE. In Section III, how MANHATTAN gathers and visualizes information to support collaboration and team awareness. In Section IV, we describe a qualitative exploratory study we conducted to collect feedback about the decision taken in our approach. We summarize our work in Section V.

II. Real-Time Visualization of Software Systems as Cities

Figure 1 shows Manhattan in our recommended setup: A dual-monitor setting in which one of the two displays shows the interactive city visualization.

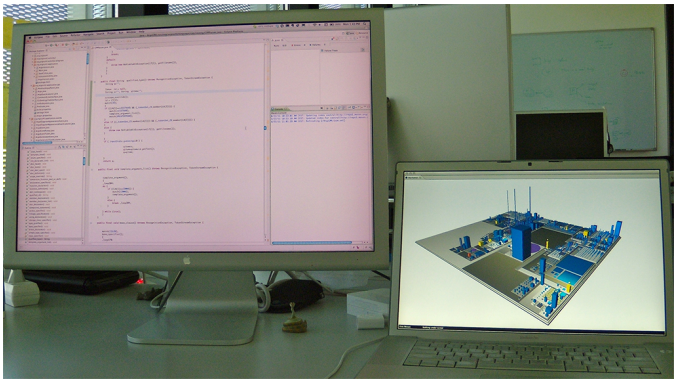

Fig. 1. The Manhattan Eclipse plugin in action in a dual-monitor setting

The city metaphor maps architectural elements to software entities: Projects are represented as cities, packages as districts, and classes as buildings. Further information about the 3D city metaphor can be found in the CodeCity website.[1] In Manhattan, the dimensions of each building is mapped to the number of fields (NOF) and the number of methods (NOM) of any class. To clearly distinguish interfaces, they are represented by circular buildings in a special color.

The user can navigate the city through the *orbital* or *first-person* navigation modes. In the orbital mode the camera is fixed on a dome centered on top of the city and the user moves on the surface of the dome. The first-person navigation mode gives full control over the movement and allows any kind of translation or rotation. By hovering the mouse pointer on any building, a tooltip that describes the metrics (*i.e.,* NOF, NOM, and LOC) of the corresponding class appears; by right-clicking, Manhattan opens the class in the editor. Manhattan also offers a *focused* view, in which only the chosen entities are displayed: The users select any number of entities and press the *v* key to activate this view. Moreover, by clicking on a class and pressing *h*, Manhattan presents a view with only the hierarchically related classes.

To visualize systems and information, Manhattan needs a language-specific parser that extracts the model from the system's code. Currently Manhattan uses the abstract syntax trees provided by the language-integration plugins (JDT for Java). Manhattan handles Java projects by building on top of X-Ray[2], a software visualization plugin that uses the JDT APIs to extract projects' information. By creating different language-specific parsers to extract the model of systems loaded in Eclipse, Manhattan would easily support additional programming languages.

[1]http://codecity.inf.usi.ch
[2]http://xray.inf.usi.ch/

III. Visualizing Team Activity in Manhattan

Figure 2 shows how Manhattan visualizes information to improve collaboration and team awareness.

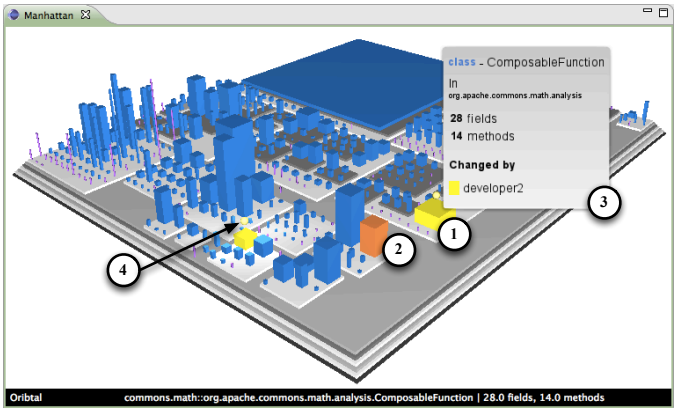

Fig. 2. Manhattan visualizing team activity

Manhattan visualizes (i) the emerging conflicts in which the developer is involved, (ii) the classes that have been changed or deleted by the other developers, and (iii) the developers who modified a class more frequently.

Manhattan is able to visualize the system while it is being changed by several developers in parallel, because Syde[3] monitors source code changes directly at each developer's workspace. Thus, it captures changes and broadcasts its information at every build action, rather than at every commit.

Let us consider a team of three developers working together and analyze their interaction from the perspective of developer 1. When any class is changed by someone other than 1, she sees the corresponding building turning yellow (point 1 in Figure 2). As soon as one of her colleagues removes any class, the building does *not* disappear in her view, but becomes orange (point 2). Supplementary change information is included in the tooltip description for classes (point 3), which lists the names of the developers who recently modified the given class. Developers are grouped by the kind of change they performed, and those that deleted the class are put first.

To visualize a conflict alert on a class, Manhattan places a sphere on top of its building (point 4 in Figure 2). The color depends on the kind of conflict the sphere represents: Emerging conflicts are shown with a yellow sphere, while committed conflicts (those in which one of the involved developers has committed his changes) are shown with a red sphere. We also use *conflict beacons*: a spotlight positioned above of a conflict sphere and pointing towards the ground. These beacons illuminate an area of the city around their associated conflict spheres, so that conflicts are always clearly visible. When the user notices an alert and puts the mouse on the conflict sphere, the beacon is deactivated.

[3]http://syde.inf.usi.ch

IV. Preliminary Evaluation

Given Manhattan's prototypical nature, we opted for a qualitative exploratory study aimed at collecting feedback to verify whether we are on the right track with our approach. We wanted to understand whether: (1) our visual notifications of changes are visible and intuitive, (2) our conflict alerts are usable and provide relevant information, and (3) participants believe that a more mature version of our tool would help them in their everyday development sessions. We ran the preliminary evaluation with four participants.

A. Study Description

Recently, we designed an experiment to verify the conflict detection capabilities of *Syde* [12]: We asked pairs of participants to perform a set of programming tasks on *Checkstyle*[4], to fix a set of broken tests from the project's test suite. Participants were given three tasks each and whenever they completed a task, they committed their changes to a repository. Before moving to the next task, the participants had to successfully incorporate each other's changes so that, at the end of the experiment, the whole test suite was passing for the working copy of both participants. Each pair of programming tasks was designed to lead to conflicts on the repository. Participants could not see each other's screen or talk. We used the same setup to validate Manhattan: , and at the beginning they were asked to fill-in a screening questionnaire. After a brief tutorial on Manhattan, as a "warm up" exercise, the participants did a program comprehension task on the *Checkstyle* system using Manhattan. Afterwards, we asked participants to do a set of programming tasks. After the experiment, participants filled in a debriefing questionnaire. We describe each phase more in detail.

Screening Questionnaire. The first part is focused on assessing participants' experience. They are asked about the size of the systems they usually work with, the size of their development teams, and their experience with SCMs. In the second part we ask participants about their knowledge in software visualization and their reverse engineering experience and habits. We also ask some questions to assess whether the features of Manhattan match participants expectations about tools to support collaborative development.

Warm-up Task. Participants are given ten minutes to explore *Checkstyle* using our tool and to report their findings about the design of this system. The main goal of this task is to give participants some time to get acquainted with the visualization, so that they can use it more efficiently during the subsequent programming tasks. Ten minutes is not a realistic amount of time for a program comprehension task, but we are not specifically interested in the quality of the findings reported by the participants. Still, we ask them a report to verify that they understand how to use the tool.

[4]http://checkstyle.sourceforge.net

Programming Tasks. The assignment is composed of three coding tasks to be done by two participants in collaboration. Each participant has a different, but complementary, set of tasks. For a task to be considered finished, each participant has to implement what the task was requesting, coordinate with his pair, check in his changes, update the code with the changes done by his pair, and verify that all the tests related to the task pass. Each task is accompanied by instructions to prepare for it, which contains the tools and views that participants are allowed (or not) to use. During all programming tasks, participants are allowed to communicate via instant messaging only. We asked the participants to perform the following tasks:

1) *Improve class* `MethodCountCheck`. Participants are asked to make different modifications in method `checkCounters`, which counts the number of methods in a class. One participant has to refactor the method `checkCounters` by creating a utility method to remove the repetition on the code of `checkCounters`. The other has to implement checks missing in `checkCounters`.
2) *Finish class* `PlainTextLogger`. Participants were asked to finish the implementation of class `PlainTextLogger`. One participant had to complete the implementation of method `addError` and implement method `fileStarted`. The second participant had to complete the implementation of method `addError` with different functionalities and implement method `fileFinished`.
3) *Finish class* `JsonLogger`. Participants were asked to complete the implementation of class `JsonLogger`. One participant had to complete the implementation of method `addError` and implement method `fileFinished`. The second participant had to complete the implementation of method `addError` with different functionalities and implement method `fileStarted`. The description is similar to the one of Task 2, but what was asked to be changed in method `addError` was different.

Debriefing Questionnaire. The first part of this questionnaire aims at verifying that the experiment was appropriate, meaning that it was not too complicated for the participants to understand and that we provided enough guidance during the experiment. After that, we ask a few questions to assess the usability of Manhattan during the program comprehension task. Then participants are asked to report their experience during the programming tasks, by answering a set of questions about every task. These questions aim at discovering whether participants managed to avoid merge conflicts thanks to our conflict alerts and to what extent they communicated with each other to resolve emerging or committed conflicts. Afterwards, we ask participants feedback about the way we visualize awareness information and about the tool in general. Finally, participants are asked to write the positive and negative aspects of Manhattan and the future improvements they would be mostly interested in.

B. Results

According to the feedback received by participants, MANHATTAN seems to be on the right track. We have received initial positive feedback from three participants (75%) about the possibility to "see" the activity of the development team.

We find the feedback provided by two participants particularly interesting; we refer to them as *a* and *b*. The former appreciated our unified view, while the latter liked our structural visualization, but was less interested in seeing the team activity. We first discuss the comments from participant *a*, and then focus on the less positive participant *b*.

Participant *a* has almost eight years of experience in the industry, working on large industrial systems. When asked about the positive aspects of MANHATTAN, the participant wrote: "*It is a very intuitive way to get familiar with a code base, and also for informing on conflicts before they become too painful to fix*". This is an argument in favor of our approach, as it comes from an experienced practitioner who is often confronted with merge conflicts (according to *a* conflicts arise "*one time out of three*" when committing changes).

Since the results presented by Grinter show that, due to lack of awareness, developers commit even incomplete changes to avoid conflicts [6], we were not expecting a feedback such as the one given by *b* about his low interest in visualizing team activity. We informally asked the participant the reasons to the answers provided in the debriefing questionnaire. From this further investigation, we understood that *b* is used to work in a small collocated team, where the degree of communication is high and each member of the development team is aware of the activity of the others. In such a context, the participant is used to merge conflicting changes easily. This is an indication that our approach is more geared towards a distributed development team setting, where the level of awareness drops due to low human interactions.

The participants shared some critical remarks about the navigation system, which lacks mouse interactions such as rotating the camera by dragging the mouse, or zooming by scrolling. We agreed with most of these suggestions, and we already implemented many of them after receiving the participants' feedback.

V. CONCLUSION

In this paper, we presented MANHATTAN, an Eclipse plugin to visualize team activity in real-time (available at http://manhattan.inf.usi.ch/). It uses a city metaphor to depict software systems and it enriches the visualization with information about team activity and emerging conflicts, by leveraging the information provided by our Syde Eclipse plugin. We ran a preliminary evaluation with four participants, getting positive reactions, which motivates us to pursue work in this direction.

Our preliminary evaluation lacks the support of a quantitative study of appropriate scale to draw any definitive conclusion. However, the questions in the exploratory study gave us strong indications that our approach is promising for a distributed setting, which nowadays is becoming the rule rather than the exception.

REFERENCES

[1] J. T. Biehl, M. Czerwinski, G. Smith, and G. G. Robertson. FASTDash: a visual dashboard for fostering awareness in software teams. In *Proceedings of CHI 2007 (25th SIGCHI Conference on Human Factors in Computing Systems)*, pages 1313–1322. ACM, 2007.

[2] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Proactive detection of collaboration conflicts. In *Proceedings of ESEC/FSE 2011*. ACM, 2011.

[3] D. Damian, L. Izquierdo, J. Singer, and I. Kwan. Awareness in the wild: Why communication breakdowns occur. In *Proceedings of the ICGSE 2007 (International Conference on Global Software Engineering)*, pages 81–90. IEEE, 2007.

[4] C. R. B. de Souza, D. Redmiles, and P. Dourish. Breaking the code, moving between private and public work in collaborative software development. In *Proceedings of Group 2003 (ACM International Conference on Supporting Group Work)*, pages 105–114. ACM, 2003.

[5] P. Dourish and V. Bellotti. Awareness and coordination in shared workspaces. In *Proceedings of CSCW 1992 (ACM Conference on Computer-supported Cooperative Work)*, pages 107–114. ACM, 1992.

[6] R. Grinter. Supporting articulation work using software configuration management systems. *Computer Supported Cooperative Work*, 5(4):447–465, 1996.

[7] M. L. Guimarães and A. Rito-Silva. Towards real-time integration. In *Proceedings of CHASE 2010 (Workshop on Cooperative and Human Aspects of Software Engineering)*, pages 56–63. ACM, 2010.

[8] L. Hattori and M. Lanza. Syde: A tool for collaborative software development. In *Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering)*, pages 235–238, 2010.

[9] R. Hegde and P. Dewan. Connecting programming environments to support ad-hoc collaboration. In *Proceedings of ASE 2008 (23rd International Conference on Automated Software Engineering)*, pages 178–187. IEEE, 2008.

[10] J. Herbsleb, A. Mockus, T. Finholt, and R. Grinter. Distance, dependencies, and delay in a global collaboration. In *Proceedings of CSCW 2000 (ACM Conference on Computer Supported Cooperative Work)*, pages 319–328. ACM, 2000.

[11] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of ICSE 2006 (28th ACM/IEEE International Conference on Software Engineering)*, pages 492–501. ACM, 2006.

[12] M. L. Lile Hattori and M. D'Ambros. A qualitative analysis of preemptive conflict detection. Technical Report 2011/05, University of Lugano, 2011.

[13] F. Rigotti. Visualizing software systems and team activity. Master's thesis, University of Lugano, Sept. 2011.

[14] A. Sarma, D. Redmiles, and A. van der Hoek. Empirical evidence of the benefits of workspace awareness in software configuration management. In *Proceedings of FSE 2008 (16th ACM SIGSOFT International Symposium on Foundations of Software Engineering)*, pages 113–123. ACM, 2008.

[15] R. Wettel, M. Lanza, and R. Robbes. Software systems as cities: A controlled experiment. In *Proceedings of ICSE (33rd ACM/IEEE International Conference on Software Engineering)*, pages 551 – 560, 2011.