Visual Storytelling of Development Sessions

Roberto Minelli, Lorenzo Baracchi, Andrea Mocci and Michele Lanza
REVEAL @ Faculty of Informatics — University of Lugano, Switzerland

***Abstract*—Most development activities, like program understanding, source code navigation and editing, are supported by Integrated Development Environments (IDEs). They provide different tools and user interfaces (UI) to interact with the source code, such as browsers, debuggers, and inspectors. It is uncertain how and when programmers use different UI elements of an IDE and to what extent they appropriately support development.**

Previously we developed DFLOW, a tool that seamlessly records and processes interaction data. Our long-term goal is to assess to what extent the UIs of IDEs support the workflow of developers and whether they can be improved. As a first step we present our approach to analyze development sessions in the form of visual storytelling. We illustrate our initial catalogue of visualizations through two development stories.

I. INTRODUCTION

Integrated Development Environments (IDEs) are the fundamental applications used to develop software systems [1], [2]. They are composed of different tools and facilities to support the work of developers that involve source code [3], [4]. Besides few studies (*e.g.,* [5]) it is unclear how developers interact with the UI of IDEs, and whether the UI gives enough support to the development process.

We are investigating an approach to record the interactions between the developer and the IDE [6]. Our recording tool, DFLOW, silently records IDE interactions while the developer is programming, including development events and UI events. The former are actions, such as navigating or editing source code. The latter are mainly *window* interactions, like window movement or resizing since our target IDE—the PHARO SMALLTALK IDE—is a window-based development environment (see http://pharo.org).

Our hypothesis is that interaction data recorded by DFLOW may reveal important insights about developer behavior inside the IDE. In this work, we present preliminary results on leveraging such data to enable visual storytelling of development sessions. We recorded more than 200 development sessions from different developers performing their own tasks, totaling over 100,000 development activities and about 80,000 interactions with the UI of the IDE. We illustrate a preliminary catalogue of visualizations and we show how they can be leveraged to obtain interesting development stories.

This paper makes the following contributions:

- A collection of development stories driven by novel visualizations of development sessions;
- A set of visualizations to depict how developers use the IDE from different perspectives.

Structure of the Paper. Section II describes a catalogue of visualizations to depict the developer behavior. Section III leverages such visualizations as the medium for visual development storytelling. Section IV presents related work. Finally, Section V concludes our work.

II. VISUALIZING INTERACTION DATA

DFLOW collects interaction data and processes it to characterize development sessions. A development session is composed of two classes of interaction data: development events and UI events. The former are the events that involve source code (*i.e.,* from browsing to editing source code). UI events are triggered when the user interacts with the windows of the Pharo Smalltalk IDE, the target IDE for our study. Previously we devised a visual approach to understand and characterize development sessions from the UI perspective [6]. With our *"UI View"* we characterized more than 200 development sessions according to how the development flows across the different windows of the IDE. In this paper we recall the *"UI View"* and introduce four additional visualizations.

Activity Forest: The view depicts the program entities involved in a development session enriched with structural source code information. Figure 1 shows an example.

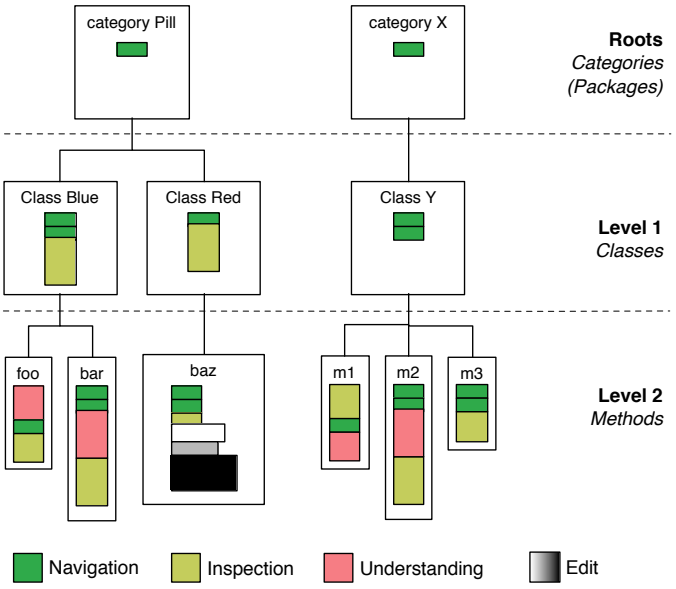

Fig. 1. An Example of the Activity Forest.

The visualization is composed of a forest of trees (two in the example), where each tree represents a sequence of development actions in a context, *i.e.,* subsequent actions happening on program entities contained in the same package. Thus, the root of each tree is a category (or package). Each category has classes as children. Only the classes subject to development actions are displayed (*i.e.,* not necessarily all the classes in the category). In the same way, classes have methods as children. Inside each node the view portrays development activities as colored boxes. Figure 2 shows a magnification of the activities on the method `baz` from Figure 1.

Each color represent a type of activity (see the color legend in Figure 1). The height of each activity box is proportional to the time spent, the width of boxes is fixed. The only exception are edit activities: Their width is proportional to the *size of the change*, *i.e.,* the difference between the size of the method before and after the edit. Edit activities are colored with a greyscale to represent the size of the method, *i.e.,* white for smaller methods and black for the biggest method edited in the session.

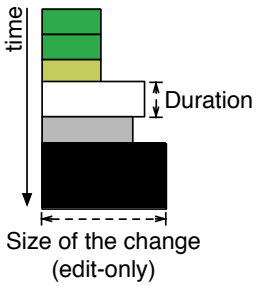

Fig. 2. A detail of Figure 1

Activity Timeline: The view portrays the development activities of a development session as a timeline. The view emphasizes the sequential nature of the activities and their duration. Figure 3 shows an example.

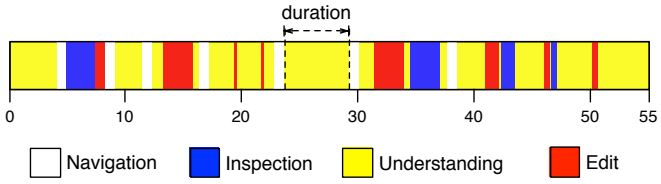

Fig. 3. An Example of the Activity Timeline.

In the example, each activity is represented by a colored box: white for navigation, blue for inspection, yellow for understanding, and red for editing. While the height of the timeline (*i.e.,* and of the activities) is fixed, the width of each activity is proportional to its duration. The timeline is enriched by regular time ticks at 10 minutes intervals.

Cumulative Activity View: The view shows development activities in a cumulative bar chart. This layout stresses the partitioning of different types of activities (*e.g.,* navigation, editing, understanding). Figure 4 shows an example.

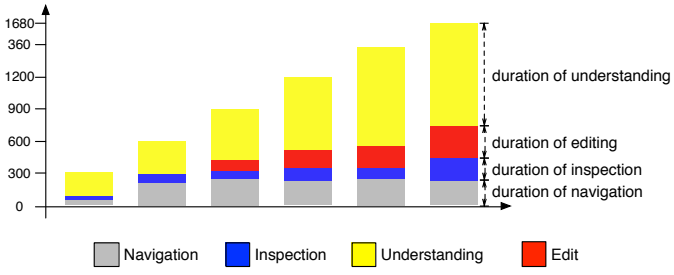

Fig. 4. An Example of the Cumulative Activity View.

This visualization presents the same information depicted in Figure 3 but in a different form. The vertical axis of the graph represents time (in seconds). The first bar represents the activities in the first 5 minutes of the session, the second bar portrays the first 10 minutes, and so on.

UI View: The visualization depicts the interactions of the developer with the UI of the IDE, *i.e.,* windows. It emphasizes how the development flow advances through multiple windows. Figure 5 shows an example.

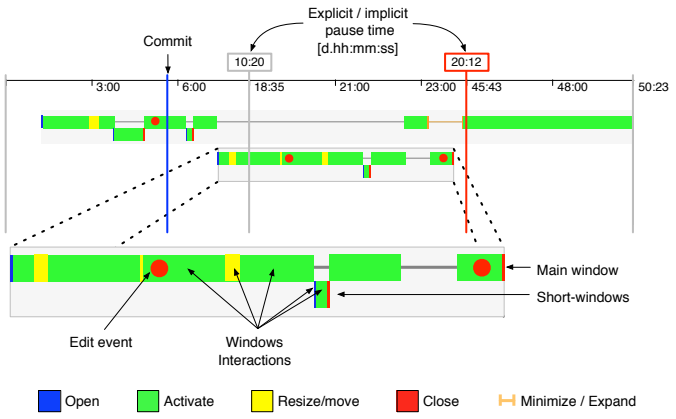

Fig. 5. An Example of the UI View.

The visualization depicts *"tracks of windows"* [6]: Each track is composed of a *main window* and several *short-lived windows* (or small-windows). Main windows were open for more than 1 minute during the session, small-windows have a lifetime shorter than 1 minute and are "attached" to a main-window. Different tracks of windows occupy different vertical coordinates. Each window is represented by a gray line with events on it. When the gray line is visible it means that, during this period, the window is "idle", *i.e.,* the developer is focusing on another window. The view depicts the following window interactions: open (blue), resize/move (yellow), activate/focus (green), close (red), and minimize/expand (orange). The view features red dots to depict when and on which window edit events happened. Vertical lines across the view depict the start and end times of the session and pause times during development. Pauses can be explicitly triggered (gray) by the developer and implicitly detected (red) if the developer is inactive for a given amount of time (say 10 minutes).

Workspace View: The view mirrors the Pharo IDE and depicts position and size of opened windows over time. It highlights which areas of the IDE are the most crowded.

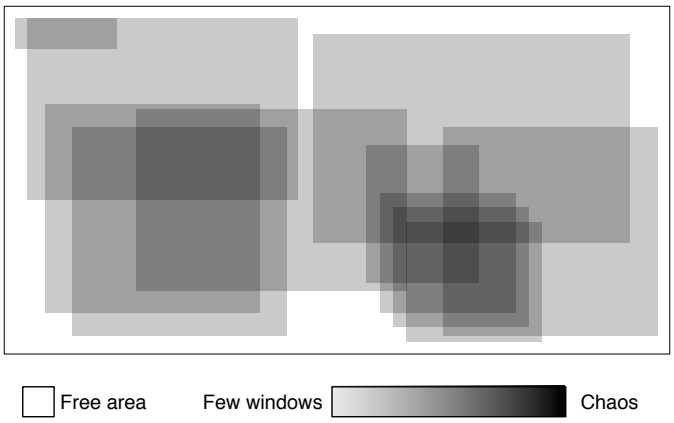

Fig. 6. An Example of the Workspace View.

Figure 6 shows an example. The outermost container is the Pharo IDE. Inside there are translucent grey boxes representing the windows the developer interacted with during a development session. The view shows the evolution of the entire session, step by step. Figure 7 shows three subsequent snapshots of a session through the Workspace View.

Fig. 7. Subsequent Moments Visualized through the Workspace View.

At the beginning of the session (see Figure 7) the developer concentrates her focus on the leftmost part of the IDE. As the session continues the developer fills the IDE with windows. The highest *"concentration of windows"* happens near the bottom right corner of the IDE (*i.e.,* darkest area of the view).

Summing Up

We presented a catalogue of 5 visualizations to characterize the behavior of developers during a development session. The Activity Forest, Activity Timeline, and Cumulative Activity View depict development activities such as navigating, inspecting, editing, and understanding source code. In a previous work we provided an estimate of the duration of different activities from the recorded interaction data [7]. The Activity Forest highlights the program entities involved in the development session and their source code structure. The Activity Timeline and the Cumulative Activity View, instead emphasize how time is spent while interacting with the IDE. The other two views, the UI View and the Workspace View, focus on pure UI interactions. In the next section we put our visualization in practice to perform visual storytelling of development sessions.

III. Visual Storytelling

A. Killing Bugs and Windows

The first story is about a session of a developer that we will call Alice. Upon starting a session DFlow asks the user for a *title* and a *session type*. The developer categorized the session as *bug-fixing*. The session lasts for about three hours, including 1 hour of pause. Figure 8 shows how the developer managed her time in terms of development activities.

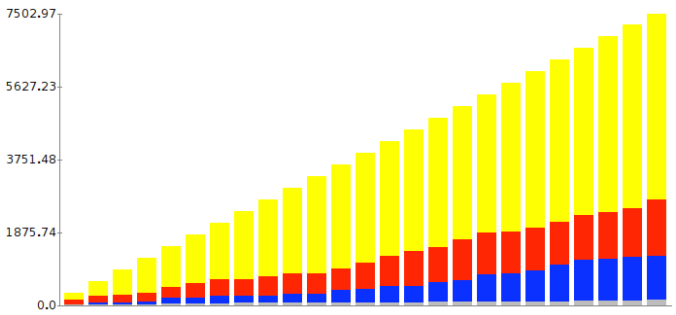

Fig. 8. Cumulative Activity View for a Bug-Fixing Session of Alice.

In each of the 5 minutes slices the developer mainly performed understanding activities. At the end of the session understanding accounts for 1 hour and 20 minutes, editing activities lasted for less than 25 minutes, and duration of inspections and navigations are respectively 19 and 2 minutes. The large predominance of understanding could be intrinsically connected with the nature of the session: Bug-fixing requires a deep knowledge of the code base.

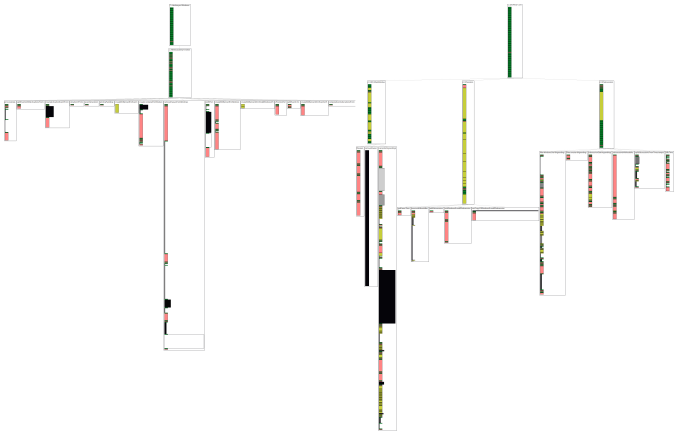

Fig. 9. Part of the Activity Forest for a Bug-Fixing Session of Alice.

The half hour the developer spent in editing source code can be summarized with the part of the Activity Forest (see Figure 9). All the edits are condensed in two contexts (*i.e.,* packages) and involve only a dozen methods. Most of the times edit events are interleaved with inspections (depicted in pale yellow), which are the means to understand the effects of the changes.

Until now we only focused on development activities. DFlow also captures interactions with the UI of the IDE. Figure 11 shows a combined visualization of the UI View (top) and the Activity Timeline (bottom) for the same Bug-Fixing Session of Alice. In this session Alice used 228 windows and she focused for very little time on each of them (*i.e.,* about 30 seconds per window). The highly interrupted development's flow of Alice in this session may be due to the way the IDE supports debugging activities. In Smalltalk, while debugging, developers perform inspections on instances of objects. Most of the times when the user inspects an object the Pharo IDE triggers an *Inspector*, *i.e.,* a small window that shows details about the inspected instance. This assumption is supported by a high number of inspection events (222). This session features 60 edit events on a dozen of methods. The red dots in the UI view represent when and where edit events happened. From Figure 11 we can observe that there are more than a dozen windows with edit events. This means that Alice tends to open multiple source code browsers on the same artifact, and close browsers immediately after an edit, thus being forced to reopen it for the next edit.

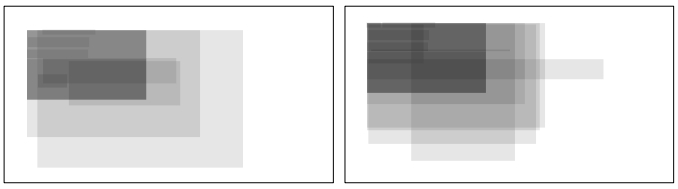

Fig. 10. Two Workspace Views of the Bug-Fixing Session of Alice.

Developers are often forced by IDEs to spawn a number of windows (or tabs) to reveal hidden relationships among source code entities. Röthlisberger *et al.* called this phenomenon the *"Window Plague"* [8]. From the highlights in Figure 11 we can observe how the environment of Alice is affected by this

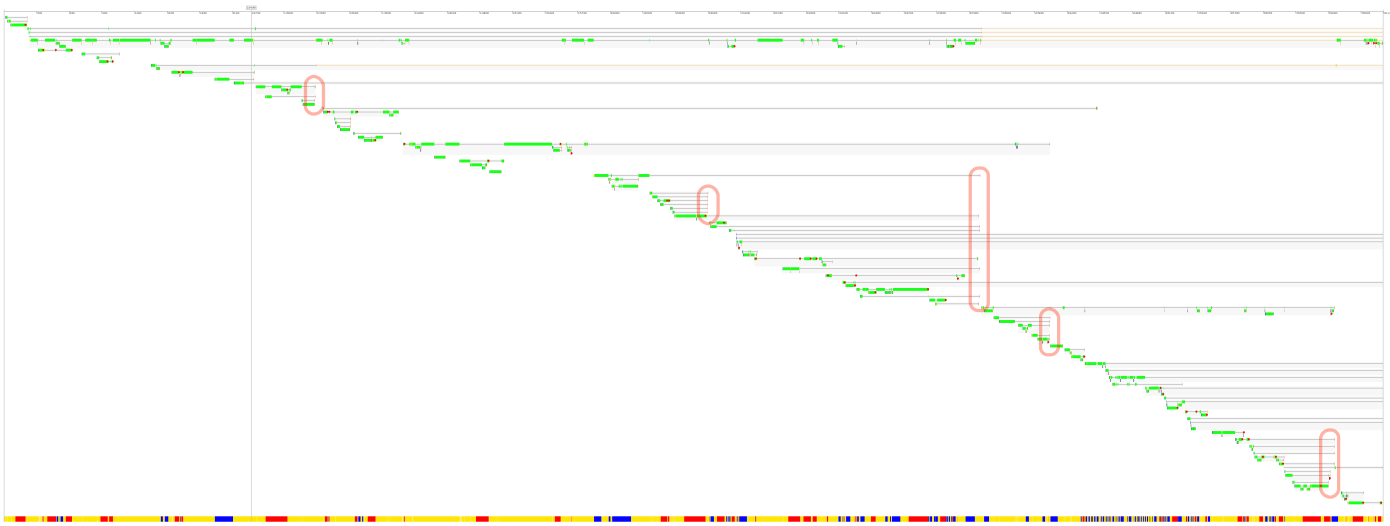

Fig. 11. A combined visualization of the UI View and the Activity Timeline for a Bug-Fixing Session of Alice.

plague. When her IDE reaches a certain "level of chaos" she simultaneously closes a number of window to lower it. Figure 10 shows two snapshots of the session of Alice using the Workspace Views. Alice has a tendency to use only the leftmost part of the IDE. This could possibly motivate the need to cure the window plague so often.

B. One Window Takes It All

This story is about an *enhancement session* of Bob. Enhancement means that the developer's intention is to add new or enhance existing functionality.

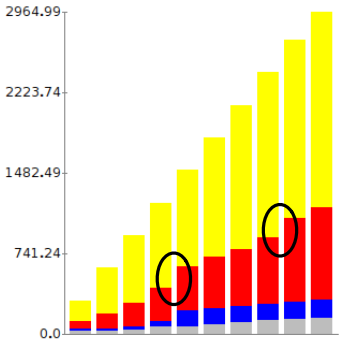

Fig. 12. Cumulative Activity View.

The session lasts for about an hour and for its entire duration the developer invests much more time in understanding than in other activities (see Figure 12). Editing increases constantly throughout the session with two major jumps, highlighted in the view.

All the edits happen in a single category (or package) as shown by the part of Activity Forest depicted in Figure 13. It highlights the activities on two methods, both part of the same class, *i.e.,* the most edited methods. From the visualization we see that the developer tends to shorten these methods while editing them, *i.e.,* their color goes from black to white. The complete Activity Forest (not shown for lack of space) includes a number of small trees depicting classes the developer browsed while building his knowledge to perform the changes.

Figure 14 shows how the IDE looks like at the beginning, towards the middle, and at the end of the session. There is a big window (*i.e.,* a code browser) that occupies almost the entire IDE space. This window remains active for the entire duration of the session. The developer tends to open (or move) all the windows towards the top let corner of the screen, hiding

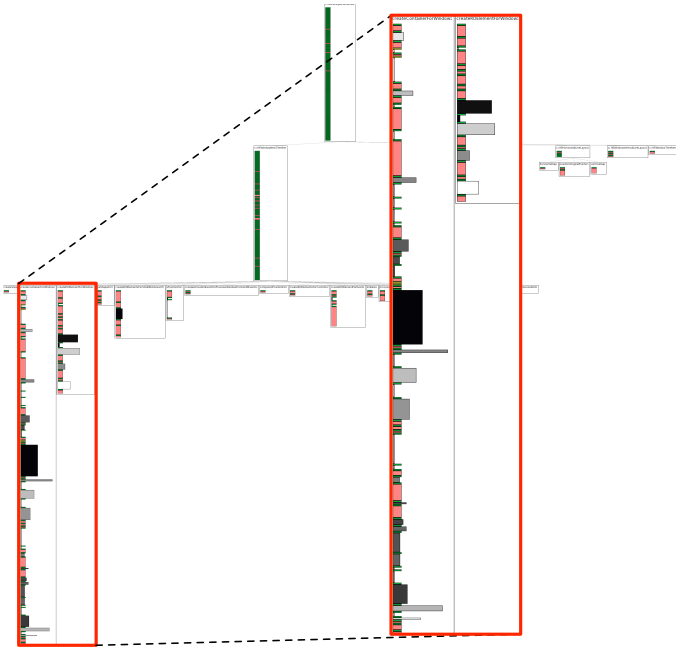

Fig. 13. Part of the Activity Forest for an Enhancement Session of Bob.

the top half of the big window. Pharo code browsers display source code in the lower part of the window. Bob moves all the windows so that he can always see the lower part of the big-window, most likely because he wants to keep an eye on the source code displayed in it. The UI View, depicted in Figure 15, shows this long-lived window, *i.e.,* the first window track. All the edits are performed using this window, *i.e.,* this session revolves around this key window.

Fig. 14. Three Workspace Views of the Enhancement Session of Bob.

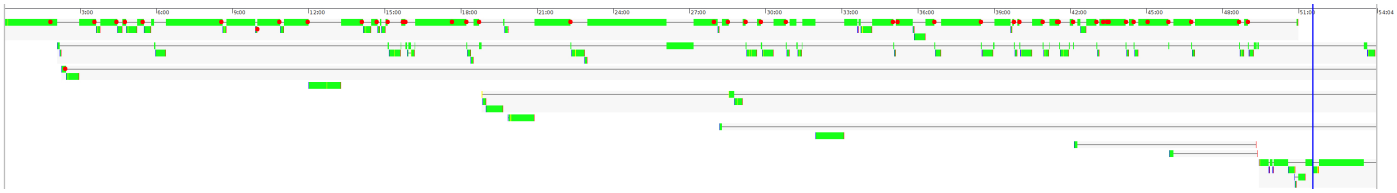

Fig. 15. The UI View of the Enhancement Session of Bob.

IV. Related Work

To understand how developers interact with IDEs researchers recorded IDE events, such as invoked API methods and keystrokes. Yoon and Myers proposed FLUORITE, a tool that records low-level development events in the Eclipse IDE [9]. Murphy *et al.* developed the MYLAR framework and observed how developers use plugins in the Eclipse IDE [5]. Robbes and Lanza proposed SPYWARE, a tool to record fine-grained source code changes in the IDE [10]. Kobayashi *et al.* developed PLOG, a MYLAR extension that records more fine-grained code changes [11]. They built a prediction model for change propagation based on the recorded interaction histories.

To make sense of development sessions researchers often used visualization techniques. AZURITE is an Eclipse plug-in that visualizes fine-grained code change histories [12]. The tool provides two views that let developers navigate through the history of changes. Servant and Jones developed CHRONOS, an Eclipse plug-in that lets developers visually query and explore historical source code change events [13]. They also provide a motivating example to show how developers can benefits from the visualization offered by CHRONOS. Gîrba *et al.* [1] and Greevy *et al.* [2] visualized code ownership with the "Ownership Map". Telea & Auber developed CODE FLOWS, a tool that shows changes between revisions of files and highlights important events such as drift and merges [14]. Ogawa & Ma propose different visualizations of source code and developers (*e.g.,* [15], [16]).

In our work we collect data with our DFLOW tool [6] and we use a catalogue of visualizations to understand the developer behavior. While most related work focuses on version control systems data, our views depict fine-grained interaction data collected with our DFLOW tool.

V. Conclusions

IDEs offer an significant amount of tools and UIs to support development activities such as editing, navigation and program understanding. However, it is unclear how developers exploit such diverse facilities to perform development activities.

We devised five views to support preliminary analytics of developer interactions with the IDE. The views leverage the data recorded by DFLOW, our supporting tool that silently records IDE events. As a proof of concept we used the visualizations to support visual storytelling of interesting developer behaviors. The two stories illustrate that it is possible to infer insights about how developers use the IDE, pointing out veritable development styles in terms of UI usage.

As future work we plan to extend our catalogue of visualizations and enlarge our dataset of development sessions to discover new interesting development stories.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science foundation for the project "HI-SEA" (SNF Project No. 146734).

References

[1] T. Gîrba, A. Kuhn, M. Seeberger, and S. Ducasse, "How developers drive software evolution," in *Proceedings of IWPSE 2005 (8^{th} International Workshop on Principles on Software Evolution)*. IEEE, 2005, pp. 113–122.

[2] O. Greevy, T. Gîrba, and S. Ducasse, "How developers develop features," in *Proceedings of CSMR 2007 (11^{th} European Conference on Software Maintenance and Reengineering)*. IEEE, 2007, pp. 265–274.

[3] A. Ko, B. Myers, M. Coblenz, and H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE TSE 2006 (Transactions on Software Engineering)*, vol. 32, no. 12, pp. 971–987, 2006.

[4] J. Sillito, G. C. Murphy, and K. De Volder, "Asking and answering questions during a programming change task," *IEEE TSE 2008 (Transactions on Software Engineering)*, vol. 34, no. 4, pp. 434–451, 2008.

[5] G. C. Murphy, M. Kersten, and L. Findlater, "How are java software developers using the eclipse IDE?" *IEEE Software*, vol. 23, no. 4, pp. 76–83, 2006.

[6] R. Minelli, A. Mocci, M. Lanza, and L. Baracchi, "Visualizing developer interactions," in *Proceedings of VISSOFT 2014 (2^{nd} IEEE Working Conference on Software Visualization)*, 2014, p. to appear.

[7] R. Minelli, A. Mocci, M. Lanza, and T. Kobayashi, "Quantifying program comprehension with interaction data," in *Proceedings of QSIC 2014 (14^{th} International Conference on Quality Software)*, 2014, p. to appear.

[8] D. Roethlisberger, O. Nierstrasz, and S. Ducasse, "Autumn leaves: Curing the window plague in IDEs," in *Proceedings of WCRE 2009 (16^{th} Working Conference on Reverse Engineering)*. IEEE, 2009, pp. 237–246.

[9] Y. Yoon and B. A. Myers, "Capturing and analyzing low-level events from the code editor," in *Proceedings of PLATEAU 2011 (3^{rd} Workshop on Evaluation and Usability of Programming Languages and Tools)*. ACM, 2011, pp. 25–30.

[10] R. Robbes and M. Lanza, "Characterizing and understanding development sessions," in *Proceedings of ICPC 2007 (15^{th} International Conference on Program Comprehension)*. IEEE, 2007, pp. 155–166.

[11] T. Kobayashi, N. Kato, and K. Agusa, "Interaction histories mining for software change guide," in *Proceedings of RSSE 2012 (3^{rd} International Workshop on Recommendation Systems for Software Engineering)*, 2012, pp. 73–77.

[12] Y. Yoon, B. Myers, and S. Koo, "Visualization of fine-grained code change history," in *Proceedings of VL/HCC 2013 (IEEE Symposium on Visual Languages and Human-Centric Computing)*, 2013, pp. 119–126.

[13] F. Servant and J. Jones, "Chronos: Visualizing slices of source-code history," in *Proceedings of VISSOFT 2013 (1^{st} IEEE Working Conference on Software Visualization)*, 2013, pp. 1–4.

[14] A. Telea and D. Auber, "Code flows: Visualizing structural evolution of source code," in *Proceedings of EuroVis 2008 (10^{th} Joint Eurographics / IEEE - VGTC Conference on Visualization)*. Eurographics Association, 2008, pp. 831–838.

[15] M. Ogawa and K.-L. Ma, "code_swarm: A design study in organic software visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 1097–1104, Nov 2009.

[16] ——, "Software evolution storylines," in *Proceedings of SOFTVIS 2010 (5^{th} International Symposium on Software Visualization)*. ACM, 2010, pp. 35–42.