Visualizing Developer Interactions

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

***Abstract*—Integrated Development Environments (IDEs) have become the *de facto* standard vehicle to develop software systems. The user interface (UI) of an IDE offers a staggering amount of facilities to manipulate source code, such as inspectors, debuggers, recommenders, alternative viewers, etc.**

It is unclear how developers use the UI of an IDE and whether such UIs actually give appropriate support to the developers.

We present a visual approach to understand and characterize development sessions from the UI perspective. The tool supporting our approach mines and processes the finest-grained UI-level events making up development sessions and presents them visually. We have collected, visualized, and analyzed hundreds of development sessions and report on our findings.

I. Introduction

Integrated Development Environments (IDEs) are the most widely used tools to develop software systems [1], [2]. Eclipse[1], IntelliJIDEA[2], and the Pharo IDE[3] are just a few examples. In essence, IDEs are collections of tools and facilities to ease the manipulation of source code [3], [4]. Eclipse, for example, offers a number of perspectives (*i.e.,* visual containers for a set of views and editor) customized for different tasks such as *developing*, *debugging*, or *running test suites*. Murphy *et al.* studied how developers use these perspectives: They found that programmers use most perspectives offered by the IDE to varying degrees and that they often use keyboard shortcuts to perform activities [5]. Besides this study, it remains unclear how developers use the UI of IDEs and whether these UIs actually give appropriate support to the developers.

In this paper we present a visual approach to answer the question: *"How do developers use an IDE with respect to the user interface it offers?"*. Our approach leverages IDE interaction data recorded by our tool DFlow to produce an interactive visualization that i) describes how developers structure their work by exploiting the IDE UI and ii) synthesizes how UI elements and UI events relate to development activities.

DFlow silently intercepts and records developer interactions while the developer is programming. DFlow considers two kinds of interactions: development events and UI interactions. Development events are actions that involve source code to different extents: We classified such interactions as i) *navigation events*, used to browse code entities, ii) *inspection events*, used to inspect the state of objects at runtime, and iii) *edit events*, that constitute actual modifications of source code. In addition, we extended DFlow to observe how developers use UI elements. In our target environment, the Pharo Smalltalk IDE, windows are the main UI building blocks. For each window, our tool records the opening and closing timestamp, as well as events such as move, resize, and minimize.

We recorded more than 170 development sessions, lasting from a few minutes to several hours, coming from 7 developers. Each development session has a type explicitly assigned by the developer, *e.g.,* bug-fixing and enhancement. In total DFlow recorded more than 110,000 development activities and about 80,000 interactions related to the usage of windows. To make sense of this data, DFlow first pre-processes it and then uses interactive visualizations to present it. Pre-processing includes for example the automatic detection and removal of idle times.

The visualization we propose has two main purposes. On the one hand it shows how developers interact with UIs, and how the work is essentially organized in development tracks, with each track led by a main window. The visualization also depicts how developers alternate between such different tracks during activities, and how the environment grows and shrinks from a UI point of view, giving a visual representation of the environment's "entropy". At the same time, the view synthesizes how development activities relate to UI usage, enabling the understanding of the UI structure at important development events like source code edits and commits. This combined view helps to gather a better understanding of the data being visualized. For example, it is interesting to see what happens to the source code when the developer spawns multiple windows and how the number of active windows influences the navigation between source code elements.

Our visual analysis led to the development of a pattern language to characterize both developers and session types. For example, we identified *"conservatives"* developers that use to use a limited number of windows and, on the other side, *"frenetic"* developers that continuously spawn windows, most likely due to a complete immersion in the development, *i.e.,* a state of mental focus so intense that awareness of the real world is lost [6], also known as "flow" [7].

This paper makes the following contributions:

- A novel visualization of development sessions that depicts how developers use the UI of the IDE;
- A catalogue of types of development sessions (and developers) based on the visualization;
- A presentation of DFlow, the tool used to collect interaction data silently while the developer is programming.

Structure of the Paper. Section II illustrates our approach and presents the principles of our visualizations. Section III presents our findings. Section IV presents and discusses the related work and Section V concludes our work.

[1]See `http://www.eclipse.org`
[2]See `http://www.jetbrains.com/idea/`
[3]See `http://pharo.org`

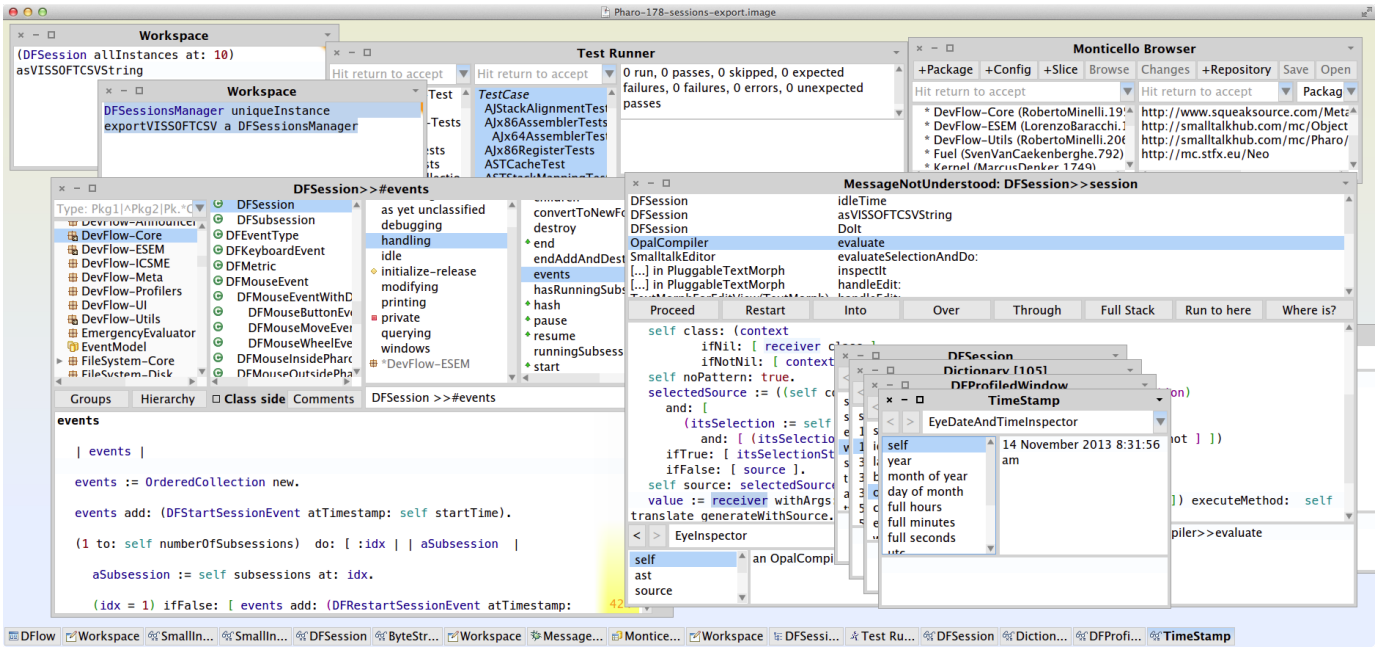

Fig. 1: A Typical Pharo Environment.

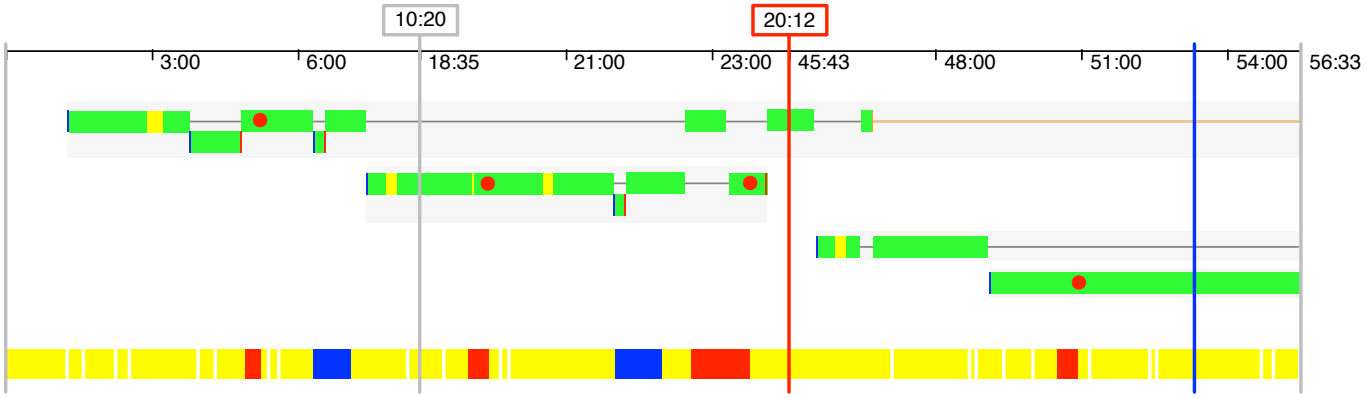

Fig. 2: A visualization of a development session.

II. Visualizing Developer Interactions

This section introduces DFlow, the tool we use to collect interaction data, and highlights the principles behind our visualizations of development sessions.

Figure 1 shows a screenshot of the Pharo IDE during a typical development session. It is composed of multiple windows of different types: Each type of window supports specialized development tasks, like code browsing and editing, object inspection, and access to the version control system. The development paradigm induced by many IDEs, including Pharo, supports the presence of a growing number of windows, and for this reason, development activities can lead to the so-called *window plague* [8]. This *plague* is also present in tab-based IDE, *i.e.,* Eclipse. The lifetime of windows is also very different: some windows are central to a particular development task and support it throughout a session, while windows supporting debugging and inspection can be very short-lived.

Figure 2 provides a sneak peek of the visualization we propose: It is composed of an upper part, the *UI View*, that depicts of the developer with the UI of the IDE, and a lower part, the *Activity Timeline*, that portrays development activities. Essentially, the UI View visualizes how the development session is organized and how the developer distributes her work in different windows. At the bottom, the Activity Timeline outlines the distribution of development activities over time, like navigation (white), inspection (blue), edit (red), and understanding (yellow). Finally, vertical lines depict idle times in the session (red), explicit termination of a sub-session by the developer (gray) or commits to the version control system (blue). Both parts of the view are aligned on the same timescale.

We briefly describe the interaction data collected by DFlow (Section II-A), we illustrate how data is pre-processed (Section II-B), and then we discuss our proposed visualization and its properties (Section II-C).

A. DFlow: *A Tool to Record Developer Interactions*

Table I lists the development and window events recorded by DFlow with an identifier and a short description. The initial character of each identifier represents the event type: **N**avigation, **I**nspection, **E**diting, and **W**indow. At the current state, DFlow records a timestamp for each event without a proper duration. Event information also contains the entity that a particular event concerns: program entities (*e.g.,* classes) for development events and window identifiers for window events.

TABLE I: Events recorded by DFlow.

Development Events	
ID	**Description**
N_1	Opening a Finder UI
$N_{2,3,4}$	Selecting a package, method, or class in the system browser
$N_{5,6}$	Opening a system browser on a method or a class
N_7	Selecting a method in the Finder UI
N_8	Starting a search in the Finder UI
I_1	Inspecting an object
I_2	Browsing a compiled method
$I_{3,4}$	Do-it/Print-it on a piece of code (*e.g.,* workspace)
$I_{5,6,7}$	Stepping into/Stepping Over/Proceeding in a debugger
I_8	Run to selection in a debugger
$I_{9,10}$	Entering/exiting from an active debugger
$I_{11,12}$	Browsing full stack/stack trace in a debugger
$I_{13,14,15}$	Browsing hierarchy, implementors or senders of a class
I_{16}	Browsing the version control system
I_{17}	Browse versions of a method
$E_{1,2}$	Creating/removing a class
$E_{3,4}$	Adding/removing instance variables from a class
$E_{5,6}$	Adding/removing a method from a class
E_7	Automatically creating accessors for a class
Window events	
ID	**Description**
$W_{1,2}$	Opening/closing a window
W_3	Activating a window, *i.e.,* window in focus
$W_{4,5}$	Resizing/moving a window
$W_{6,7}$	Collapsing/expanding a window, (*i.e.,* minimize/maximize)

Each session is composed of one or more sub-sessions. A sub-session can be triggered explicitly by the developer using the minimalistic UI of DFlow, depicted as a state machine in Figure 3. When a session is not running (3.a) the user can start a new session; DFlow asks her to insert a *brief description of her intentions* and a *type* describing the intended purpose, *e.g.,* refactoring or debugging. When a session is running (3.b) she can *Pause* or *Stop* the session. The former action ends the current sub-session that can be further resumed, while the latter also finalizes the session. By iterating between the two states in Figure 3.b and 3.c the user can split her session in different sub-sessions. DFlow session meta-data includes author and timing information (*e.g.,* sub-session start and end times).

At present, we collected around 170 development sessions, totaling more than 110,000 development events and 80,000 interactions on windows. Table II illustrates basic statistics for the recorded sessions, aggregated by developer. The table includes average pause time (*i.e.,* the average interval between two explicit sub-sessions) and idle time per session. Table III outlines statistics about development events, and Table IV outlines statistics about window events. In particular, window

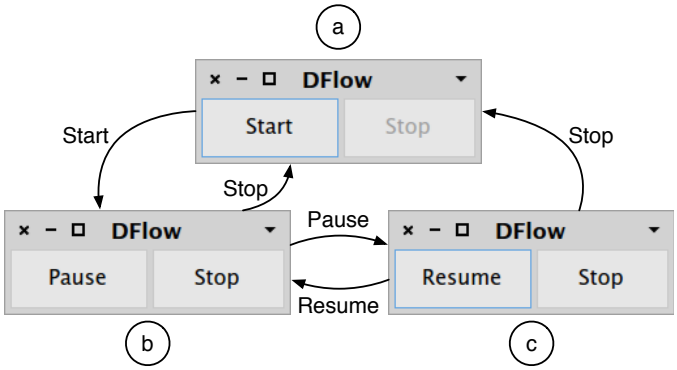

Fig. 3: The UI of DFlow and its possible states.

events suggest that developers may exploit UI in different ways: D4, D6, and D7 use on average a reduced number of windows. Our visualization, described in Section II-C, aims to highlight further insights on how development sessions are structured.

TABLE II: Sessions Statistics Grouped by Developer.

Dev.	Sessions	Sub-sessions (Avg.)	Avg. Session Duration [hh:mm:ss]	Avg. Pause / Idle Time [hh:mm:ss]
D1	12	73 (6.08)	3:01:24	0:01:26 / 28:42:55
D2	3	3 (1.00)	0:16:27	0:00:00 / 00:00:00
D3	65	97 (1.49)	0:52:32	0:18:12 / 00:29:20
D4	6	11 (1.83)	0:48:13	0:18:26 / 00:58:43
D5	72	202 (2.81)	0:54:56	3:42:45 / 00:15:52
D6	7	30 (4.29)	1:25:18	2:11:36 / 01:05:37
D7	12	80 (6.67)	1:34:25	1:41:18 / 17:29:52
ALL	**177**	**496 (3.45)**	**1:16:11**	**1:10:32 / 07:00:20**

TABLE III: Development Events Grouped by Developer.

Dev.	Navigation	Inspect	Edit	Total
D1	21,617	183	2,458	24,258
D2	393	157	24	574
D3	20,468	2,157	2,091	24,716
D4	2,183	353	1,196	3,732
D5	35,801	2,962	3,316	42,079
D6	6,862	337	472	7,671
D7	7,234	486	526	8,246
ALL	**94,558**	**6,635**	**10,083**	**111,276**

TABLE IV: Window Information Grouped by Developer.

Dev.	Windows	Open	Activation	Resize Move	Collapse Expand	Close	Tot.
D1	3,144	2,255	2,488	4,114	143	3,711	12,711
D2	71	63	59	275	0	51	448
D3	3,183	2,518	2,355	4,841	52	2,807	12,573
D4	608	609	134	978	5	710	2,436
D5	7,365	6,088	4,792	28,805	175	7,211	47,071
D6	555	525	549	468	0	580	2,122
D7	769	773	392	3,512	3	691	5,371
ALL	**15,695**	**12,831**	**10,769**	**42,993**	**378**	**15,761**	**82,732**

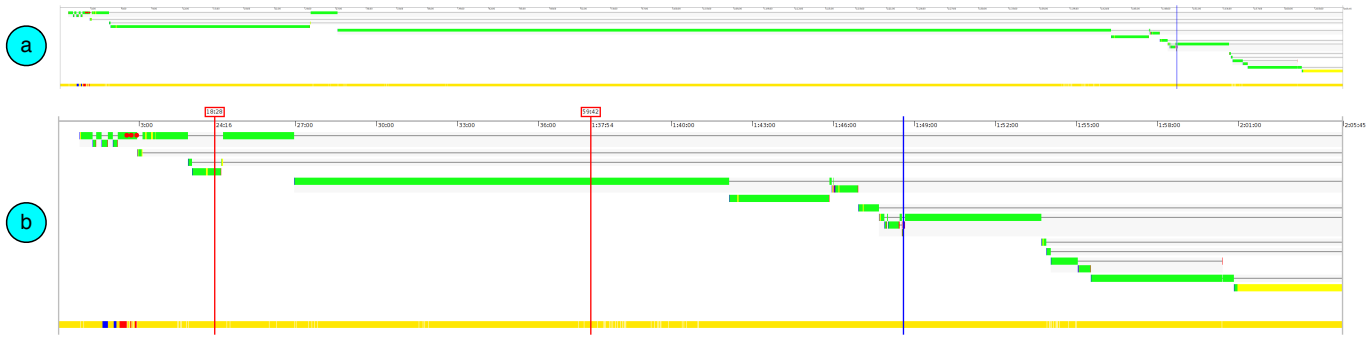

Fig. 4: The same development session visualized (a) including idle time and (b) after idle time removal.

B. Data Pre-Processing

Due to significant amounts of noise, the raw data collected by DFLow needs to be cleaned before visualization.

We apply three pre-processing phases: i) cleanup of windows events; ii) automatic detection and removal of idle times; and iii) estimation of the duration of development activities from interaction histories.

An example of *information noise* in windows events is generated when the user resizes or moves a window. During a resize, the IDE triggers a large number of small resize events, *i.e.,* one event each time the size of the window changes by one pixel. However, in essence the user resized the window just once: For this reason we compress all *resize chains* into single resize events with a duration that spans from the first to the last resize event triggered by the IDE.

A non-trivial pre-processing concerns the *automatic detection and removal of idle times*. It might happen that when a user is programming, with DFLow recording her development session, she leaves her desk for a pause. The best case is when the user explicitly pauses DFLow, using the UI depicted in Figure 3. If this is not the case, DFLow remains *"idle"*, *i.e.,* it does not record anything but it is not aware of the pause. We devised a mechanism to automatically detect idle times *a posteriori*. When DFLow mines interaction data it searches for pairs of events between which the time elapsed is more than a defined "minumum idle time", that by default is set to 10 minutes. When it finds a pair of such events, say (ev_n, ev_{n+1}), it ends the current sub-session at the timestamp of event ev_n plus some "awake time" (default: 15 seconds). DFLow then creates a new sub-session starting from the timestamp of ev_{n+1} minus the "awake time", that models the moment when the developer comes back to the IDE. In this way we introduce implicit sub-sessions and we compress data to produce better, and more useful, visualizations.

Figure 4 shows a view of a session before (a) and after (b) the removal of idle time, where it becomes evident that idle times indeed can have a deforming impact on the recorded information: The session in Figure 4 seemingly lasted a bit more than 2 hours, but in reality the actual time spent by the developer is less than half, *i.e.,* about 40 minutes (due to two idle times lasting, in total, for more than one hour).

The last pre-processing of the data assigns a duration to both windows and development interactions. Window events have a timestamp. We assign a fixed conventional duration of 1 second to open, close, minimize, and expand window events. Instead, the duration of resize/move and activation events is computed. In raw interaction data, resize and move events often appear in chains, one event for each movement or resize of 1 pixel. The duration of a resize/move activity is the time difference between the last and the first resize/move event in such chains. In the PHARO IDE, only one window at the time is active, *i.e.,* in focus. An activation event represents when the user activates (*i.e.,* clicks) on a window. The duration of an activation event is the time elapsed between the activation itself and the next event. Similarly, development events are sequences of events with their timestamp. To compute their duration we interpolate the time from one event to the next.

C. Visualization Principles & Proportions

Figure 5 explains the proposed visualization. The view is composed of two parts: an *UI View* and an *Activity Timeline*.

UI View: The *UI View* (Figure 5.a) depicts the interactions of the developer with the UI of the IDE, that is composed of multiple windows. This part of the visualization identifies and summarizes the different *tracks of windows* that the developer follows while working. A track is a composition of a main window with a set of associated short-lived windows.

Tracks of windows, essentially, represent where and how the developer used the UI elements of the IDE. There are developers that concentrate their work on a single track and developers that are more proficient when spreading their work on multiple parallel tracks. Each track of windows is "dominated" by one window, i.e., the main window of the track (*e.g.,* Figure 5.c for track 2). In turn the main window might have a number of short-lived windows associated to it, the *short-windows* (*e.g.,* Figure 5.d for track 2). These short-windows are windows with a lifespan (*i.e.,* the time that spans from their open to their close time) shorter than a given threshold (default: 1 minute). To determine which is the main window originating a small-window, say W_S, we search, among main windows, which one was last active before the birth of W_S and is still open during its lifetime.

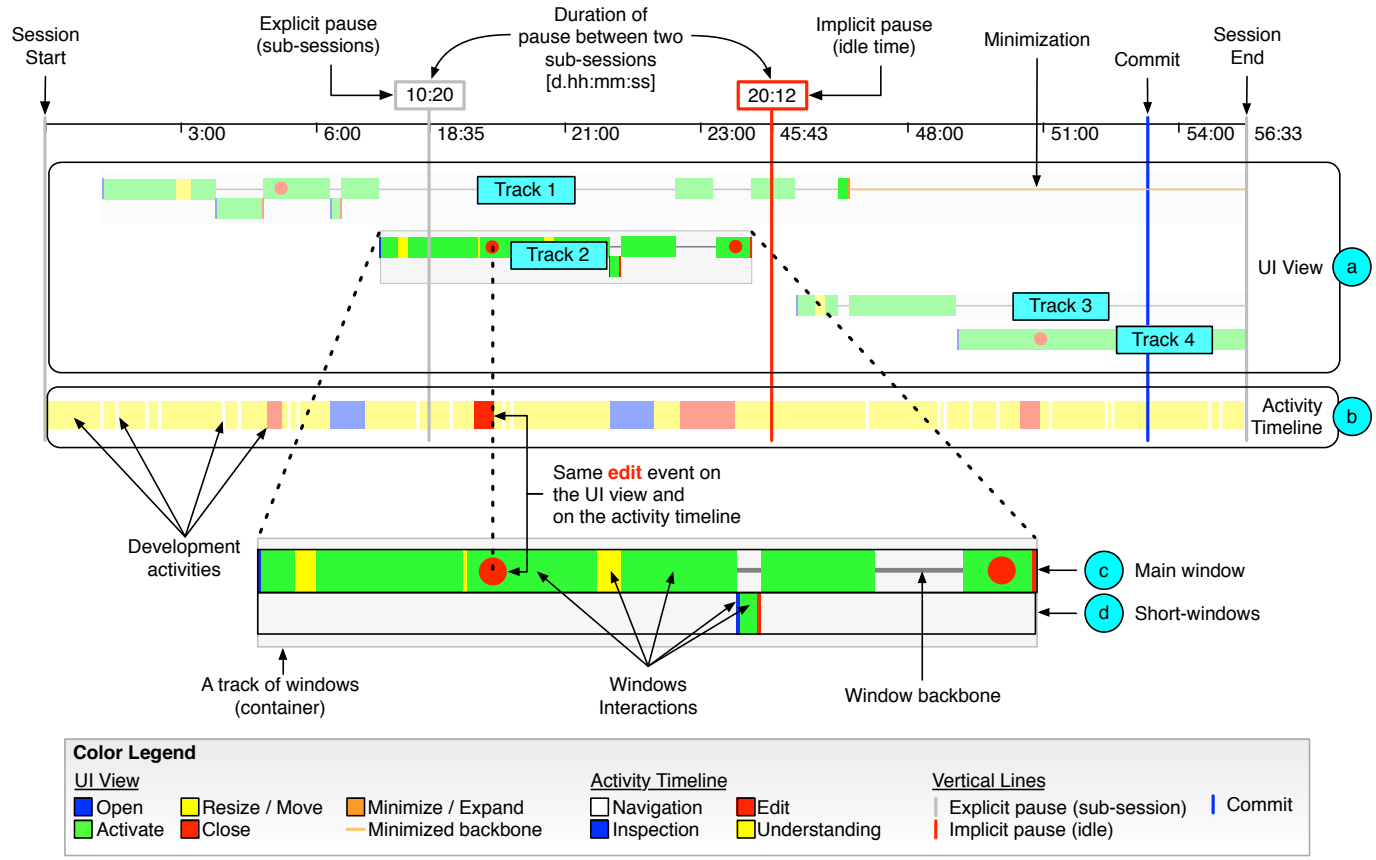

Fig. 5: Our visualization explained.

Once we created all the associations between main and small-windows, we apply the time-based horizontal layout: The horizontal coordinate of each container, or track, represent the open time of the main window that dominates it. The width of the container is proportional to the lifespan of the main window. The height of the container varies according to the number, and position, of short-windows. In the general case all short-windows of a track are positioned at the same y-coordinate. However, in case of two overlapping small-windows (*i.e.,* their lifespans overlaps), the layout pushes the second small-window down. The layout considers the windows in order of appearance, *i.e.,* sorted by open time. The y-coordinate of containers is used only to avoid overlapping between them.

A window is represented by a line, the *window backbone*. The length of this line is proportional to the lifespan of the window that represents. Window interactions, *i.e.,* events, are positioned on this line. Each event is a box with fixed height. Its length is proportional to the duration of the event. The x-position of the event on the window backbone represents time, *i.e.,* time difference between the timestamp of the event and the open time of the window. The color identifies the type of the event: open (blue), activate (green), resize/move (yellow), collapse/expand (orange), and close (red). When the window backbone is visible, *i.e.,* not covered by boxes representing events, it means that the window is currently open but not in focus. In addition to window events, on each window, the visualization shows a red dot when a source code edit event happens. This visual clue helps to identify which windows were used to perform source code changes.

The UI View in Practice: The visualization in Figure 5.a depicts 4 tracks of windows. The main window of tracks 1, 3, and 4 remain open after the session end, *i.e.,* there is no close event and the window backbone continues until the session end. On track 1 there are 2 small-windows, on the second track there is only one. The remaining tracks only have main windows. Towards the end of track 1 we can see that the main window gets minimized and remains collapsed for the rest of the session, *i.e.,* light orange window backbone. On track 2, magnified in the figure, the main window is opened, then active for some time and resized (or moved). Afterwards, it remains active for another time interval and is then quickly resized or moved again. Then an edit event happens on this window, *i.e.,* red dot. After some time the window is resized again and then a small-window is opened that remains open for a little time before giving control back to the main window. After this time, the main window remains active until the focus is given to another window (*i.e.,* the main window of track 1). When the focus is back to this track another edit event happens and after some time the window, and the track, terminate.

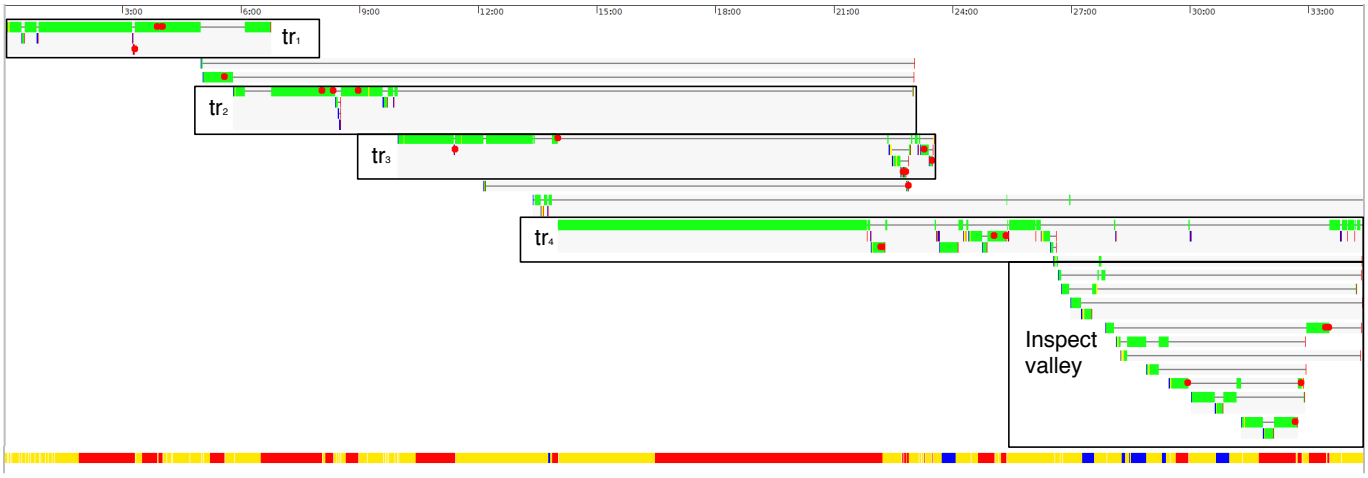

Fig. 6: Developer Story for D5: The Inspection Valley.

Activity Timeline: The *Activity Timeline* (Figure 5.b) portrays development activities such as navigation, inspection, edit, and understanding. Using this timeline one can have a clue of when and for how long the developer performed different kinds of activities. Each activity is a box with fixed height. Its length is proportional to the duration of the activity. Navigation activities are white ticks, since the navigation per se lasts for a short amount of time, *e.g.,* 1 second. Other types of events, instead, have a duration that we estimated from the raw interaction histories of DFlow. The color identifies the type of activity: navigation (white), inspection (blue), editing (red), and understanding (yellow).

Pause Times and Commits: The last visual elements on the visualization are vertical lines that span both visualizations. There are four types of such lines. Two gray lines without labels indicate the start and the end time of the session. Blue lines without label indicate the timestamp of commits in the version control system. Gray and red lines with label depict pause times. The label of these lines represents the pause time. Gray lines depict the *"explicit"* pause time, *i.e.,* when the developer paused DFlow. Red lines identifies *"implicit"* pause time, or *"idle"* time.

III. Discussion

We used the approach described in Section II to visualize the corpus of development sessions collected with DFlow. We discuss example development stories extracted from our visualizations (Section III-A) and delineate a preliminary characterization of both sessions and developers (Section III-B).

A. Telling Development Stories

The aim of our visualization is to understand how developers use the Pharo IDE while performing their daily programming tasks. In this section we examine and discuss 4 development stories about specific sessions in our corpus.

The Inspection Valley*:* Figure 6 shows part of a session recorded by developer D5. The session, excluding pauses, lasted for 49 minutes and 18 seconds.

The Figure shows the first sub-session lasting 34 minutes. There are 4 subsequent main tracks of windows, denoted as tr_1-tr_4 in the Figure. The session started with tr_1 where the user mainly performed navigation, understanding, and some edits (see the *Activity Timeline*). Then she moved to a new track and performed additional edit operations while triggering a number of small-windows for navigation purposes. The interesting part starts when she moved to tr_4. At the beginning waters are calm: The developer remained for about 8 minutes on the main window of the track. Then she performed a short but convoluted sequence of activities on tr_3 spawning more than 5 small-windows and then she happened to get lost in the *"Inspection Valley"*. Starting from minute 27, in fact, she abandoned all the main tracks to drill down in a series of inspection and understanding activities that led to a series of edit events on different windows. Finally, 6 minutes later, she cleaned up the IDE (*i.e.,* closing most of the windows) and terminates her trip into the inspection valley. These drill downs in "valleys" are a recurrent pattern in a number of sessions. We believe that this practice is encouraged by the multi-window nature of the Pharo IDE. It remains to be investigated if this pattern can lead to confusion and whether developers prefer alternative means to perform, for example, chains of inspections on object instances.

Implement First, Verify Later*:* Developers are often in a rush so they first jump here and there to understand where and what to modify and then start performing changes. Then they run their code, encounter some problems, and spend considerable amounts of time with debugging activities.

Figure 7 shows a session of developer D3 (*the image is divided in two to fit the page*). This session lasts for more than one hour and a half, removing pause times. The session counts three explicit sub-sessions, *i.e.,* vertical gray bars in the visualization, and involves 57 windows. The Activity Timeline reveals two distinct development phases. In the first two sub-sessions the developer mainly acquires knowledge of the system, through navigations (white) and understanding phases (yellow),

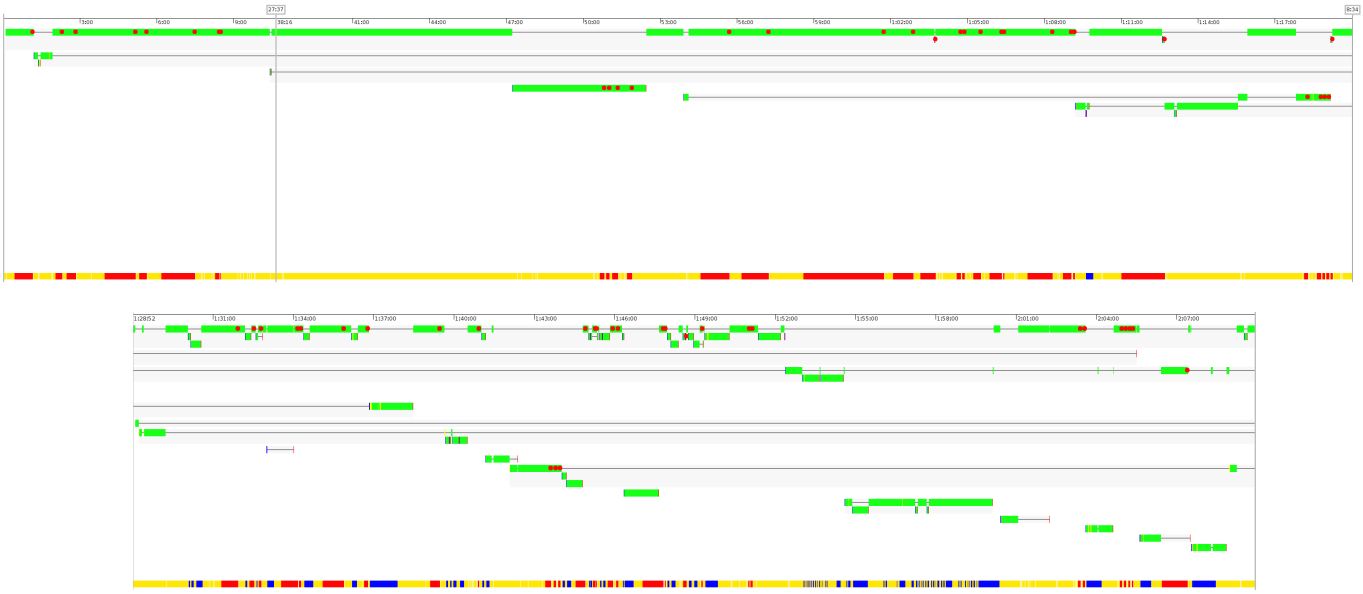

Fig. 7: Developer Story for D3: Implement First. Verify Later.

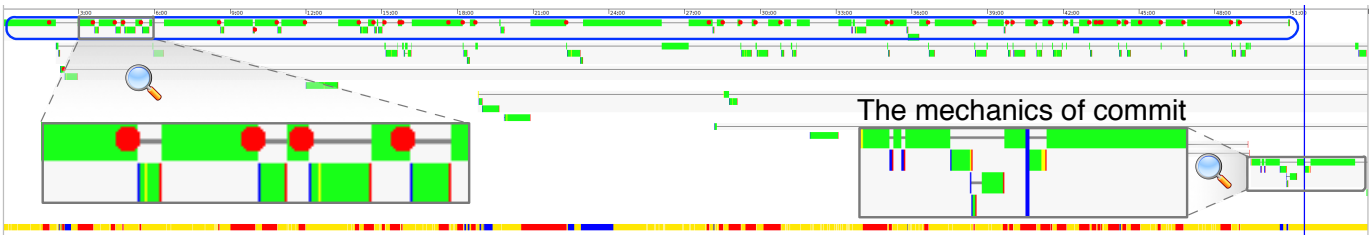

Fig. 8: Developer Story for D3: Home Sweet Home.

and performs a number of source code modifications (red). After this, the developer took a break of 8 minutes (*i.e.,* the idle time between the second and third sub-session) and then started to exercise her code. From the Activity Timeline we can infer that she was not really satisfied with her changes. The third sub-session, in fact, is full of inspections (blue) that are often related to debugging activities. In Smalltalk, developers use inspections to observe instances of objects at runtime mostly to verify the values of their fields. In this sub-session inspections are interleaved with a high number of edit activities (red), symptom of the fact that the changes performed in the first two sub-sessions were not really successful.

Home Sweet Home*:* The story is about a session of developer D3, depicted in Figure 8. The session lasts for about an hour and features 45 edit events. The visualization exposes a single main track of windows, *i.e.,* the first one. This track starts at the beginning of the session, lasts for almost its entire duration and it is intensively occupied by the main window. When this main window is not active (*i.e.,* the window backbone is visible) the developer transfers the focus for a small amount of time to other windows (*i.e.,* maximum 1.5 minutes). She wiggles around, navigates code, reads code, and finally she gets back to the main track. The interesting pattern is that she only performs changes on the first track, more precisely, on its main window (*i.e.,* all but one red dots are in the main track). It is clear that this window is a "pillar" for this session since the flow of development always returns to it.

Another peculiarity of this session is that, almost after every edit event, the developer opens a small-window, closes it and get back to the main track. It seems that the developer uses small windows as verification means for her changes. An example of this fact is magnified on the left part of Figure 8.

The last interesting thing, common to several sessions of different developers, is the visual cluster appearing towards the end of the session, near the blue commit line. This last track of windows, in fact, represents the *"mechanics of commit"*, *i.e.,* the sequence of user interface actions needed to commit source code to the versioning system. The main window of that track is the browser for *Monticello*, *i.e.,* the version control system used by the Pharo IDE. The small-windows originated from it are: i) the *change browser*, that lets the user browse for the changes before commit; ii) the *commit message box*, that lets the user enter a commit message for the current version; and iii) the *confirmation dialogs* that acknowledge every commit.

Curing the Window Plague*:* Figure 9 shows a distinctive feature of developer D5. Researchers called *Window Plague* the fact that to reveal relationships between code entities developers are forced to open a high number of windows on different

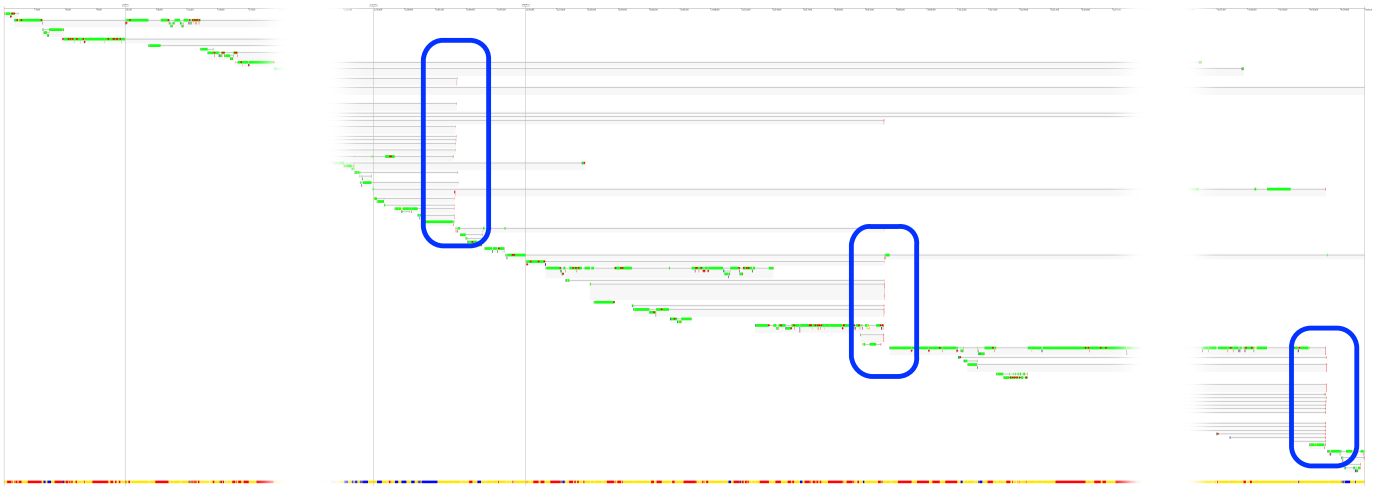

Fig. 9: Developer Story for D5: Curing the Window Plague.

artifacts [8]. The window plague leads to a crowded workspace with many opened windows (or tabs in case of tab-based IDEs). Naïvely the IDE does not come to the rescue of the developer in case of a crowded workspace. Some developers ignore this issue but others like to cleanup their environment from time to time. Developer D5, for example, is a developer that systematically cleans up her environment. Figure 9 depicts one of her sessions lasting for more than 4 hours (*note that the view has been compressed to fit the page by eliding parts of the session from the visualization*). At regular intervals (about every hour and a half) she triggers "cleaning stages" where she closes almost all windows to refresh her environment. The UI View shows that the number of windows continuously grow until the entropy level of the environment becomes unbearable for her and she decides to decrease it. There are two possible interpretations for this: Either the developer has accomplished her task and starts a new fresh task, or the environment has become so convoluted and disordered that she decides to start over even though the task is not finished. It remains to be investigated how frequent this phenomenon is and if and how the IDE can automatically come to the aid of developers in such cases.

B. Categorizing Developers and Development Sessions

We use the visualization presented in Section II-C to derive a classification of the development sessions. The ground element for our classification is the number and behavior of tracks of windows. We discarded 13 sessions that were either too short or lacked a significant number of events.

We use two dimensions for the classification: 1) the presence of dominant tracks of windows, and 2) the flow between the different tracks.

Dominant Tracks: We define a dominant track as a track with a privileged role in the development session. In other words, dominant tracks are the tracks with a predominant focus time and concentration of edit events. We devise three categories based on *Dominant Tracks*:

- **Single-Track:** There is a only a single track of windows that is predominant over all the others, if any.
- **Multi-Track:** There are two or more tracks of windows that are predominant over all the others, if any.
- **Fragmented:** There are no real dominant tracks of windows. The development flow and the focus of development are strongly fragmented, *i.e.,* the developer continuously shifts her focus from one window to another and perform edits with no apparent strategy.

Track Flow: Track flow describes the way the developer alternates from different window tracks. We devise two additional categories based on *Track Flow*:

- **Sequential Flow (S):** The development flow follows a sequential trend, *i.e.,* from one track the focus moves to the next and so on. On this type of sessions the focus rarely goes back to a previous track. These are the sessions that might suffer from the *window plague* [8], and in such cases often there are no dominant windows.
- **Ping-Pong Flow (PP):** The development flow and the focus of development continue to *zig-zag* between two or more tracks. If the fragmentation is heavy, there can be no dominant track and the visualized session appears frenetic and chaotic.

Consider the developer stories we described in Section III-A. Figure 6 is mostly fragmented since there are no dominant windows that characterize the whole development session. It also has a dominant sequential flow, because the developer starts a development track, and when another one is created the developer has either closed the previous one or leaves it out of focus. There is also minimal ping-pong behavior.

The session in Figure 7 is single-track in the implementation phase, while in the verification phase it becomes fragmented with a mixed sequential and ping-pong flow.

Consider instead Figure 8, corresponding to the "Home Sweet Home" developer story. The session has a main track where most of the edits happen, and that the rest of development activities happen in other window tracks. The session has

TABLE V: Track and Flow Characterization of Developer Session.

	Single-Track				Multi-Track				Fragmented				All						
Dev	*S*	*%*	*PP*	*%*	*S*	*%*	*PP*	*%*	*S*	*%*	*PP*	*%*	*S*	*%*	*PP*	*%*	**NC**	**%**	**Total**
D1	0	0%	1	8.3%	1	8.3%	0	0%	8	66.7%	0	0%	9	75.0%	1	8.3%	2	16.7%	12
D2	2	66.7%	0	0%	0	0%	0	0%	1	33.3%	0	0%	3	100.0%	0	0%	0	0%	3
D3	32	49.2%	13	20.0%	6	9.2%	4	6.2%	7	10.8%	1	1.5%	45	69.2%	18	27.7%	2	3.1%	65
D4	0	0%	0	0%	0	0%	0	0%	5	83.3%	1	16.7%	5	83.3%	1	16.7%	0	0%	6
D5	10	13.7%	3	4.1%	9	12.3%	1	1.4%	41	56.2%	0	0%	60	82.2%	4	5.5%	9	12.3%	71
D6	1	14.3%	1	14.3%	1	14.3%	1	14.3%	2	28.6%	1	14.3%	4	57.1%	3	42.9%	0	0%	7
D7	5	45.5%	1	9.1%	1	9.1%	1	9.1%	3	27.3%	0	0%	9	81.8%	2	18.2%	0	0%	11
ALL	**50**	28.2%	**19**	10.7%	**18**	10.2%	**7**	4.0%	**67**	37.9%	**3**	1.7%	**135**	76.3%	**29**	16.4%	**13**	7.3%	**177**

a typical ping-pong flow. In fact, the developer frequently alternates between the main track and other tracks, with a minimal privilege towards the second track.

Finally, the session in Figure 9 shows instead a fragmented session with sequential flow. Edits and focus are spread to many tracks, and no track dominates in the whole session. A lot of old tracks are simply out of focus and never closed, laying in the background (*window plague* [8]). Tracks grow almost monotonically.

Table V illustrates the characterization of our corpus of development sessions, aggregated by developer. In general, we observe that Multiple-Tracks are less common (14.2% of the total), and that the remaining sessions are uniformly distributed among the other two categories: Single-Track and Fragmented (around 40% of each type). We also observe that Sequential Flow (S) is relatively more frequent than Ping-Pong (PP), 76.3% versus 16.4%. These numbers again support the fact that developers may frequently experience the window plague. Another interesting general observation is that Ping-Pong behavior is mostly correlated with Single-Track and Multi-Track sessions: This means that Ping-Pong Flow happens between a single dominant track and minor tracks, or between multiple dominant tracks.

The results of our classification also suggest differences between the development style of different developers. Three out of seven developers (D2, D3 and D7) exhibit a strongly dominant preference for Single-Track sessions: For them such sessions account for more than 50% of the total. Developers D1 and D4, instead, tend to work in a more Fragmented fashion (66.7% and 83.3% respectively). All developers strongly prefer Sequential Flows; this result can be debated for developer D6, for which we did not collect enough sessions to observe a significant difference. Developer D6 is the subject that more frequently exhibits Ping-Pong Flow in her sessions (42.9%). Developer D3 is the subject with the higher number of sessions with Ping-Pong Flow but, in percentage, has a smaller number than D6 (27.7%).

Considering developer styles in isolation, we observe that sessions of developer D5 are almost only Sequential (82.2% of the sessions) and frequently Fragmented (56.2%). Instead, sessions of developer D3 are still mostly Sequential (69.2% of the times) but mostly exhibit a Single-Track of development (69.2% of the sessions).

IV. Related Work

Related work can be classified in approaches that study recording of IDE interactions and software visualization.

Recording IDE Interactions. One approach to understand how developers interact with IDEs is to record IDE events, like invoked IDE API methods and keystrokes.

Yoon and Myers developed Fluorite, a tool that records low-level development events in the Eclipse IDE [9]. By analyzing the recorded data, they concluded that editing source code is different than editing documentation artifacts. For example, one of the most used keystrokes by programmers is the backspace.

Murphy *et al.* tried to establish the relevance of plugin usage in the Eclipse IDE [5]. To this aim, they collected data using the Mylar framework. Analyzing the collected data, authors found that a large percentage of frequently used commands is invoked by developers using key bindings.

Robbes and Lanza proposed an approach to record fine-grained source code changes in the IDE [10]. With this data, they defined a "development session" as the time in which a developer actively modifies a software system. Using the information about development sessions, it is possible to depict a software system as the results of multiple changes operation rather than a sequence of versions. They asserted that a session follows an incremental logic of changes, *i.e.,* a developer starts by introducing basic concepts that are then progressively extended during a session.

Our tool, DFlow, integrates the collection of fine-grained development and window events generated by the IDE. At the current state, editing events are coarse grained and do not include fine-grained source code changes.

Singer *et al.* performed different studies on the time and practice of software developers, mainly using questionnaires (*e.g.,* [11]). Although they did not use a precise measure for the time taken by developer activities, they noticed that the time spent on writing source code is less than the time spent on other activities, like debugging or searching.

Software Visualization. Researchers visualized interaction histories. For example, Yoon *et al.* developed Azurite, an Eclipse plug-in that visualizes fine-grained code change histories [12]. The tool provides a "timeline" that lets developers navigate through the history of changes and quickly reach the needed information. Azurite offers also a "code history diff" to inspect the changes of particular code fragments.

Servant and Jones developed CHRONOS, an Eclipse plug-in that lets developers query, explore, and discover historical source code change events [13]. The authors provide a motivating example to show how developers can benefits from the visualization to understand how two methods co-evolved.

Researchers also mined versioning systems to study how developers collaborate to build software. Gîrba *et al.* visualize code ownership [1]. They first define a measure of code ownership and then build the "Ownership Map", *i.e.,* a view to understand when and how different developers interacted with a system. They found that the evolution of a software system can presents behavioral patterns on how developers contribute to it. They identified i) *monologue* periods, in which a developer makes most of the changes; ii) *teamwork* periods, where many developers frequently commit changes, and iii) *silence* periods.

Similarly, Greevy *et al.* visualized code ownership in a structural view [2]. The view depicts packages and classes of a system and colors each node according to its owner.

Telea & Auber developed Code Flows, a tool that uses a tubes visualization to show changes between revisions of files and highlights important events such as drift and merges [14].

Ogawa & Ma propose two visualizations of source code and developers: code_swarm, a tool that produces animated software histories from data coming from version control systems [15] and a historical visualization to show the interactions between developers in the evolution of software projects [16].

Our visualization borrows visual elements from related works, but it is inherently different. Related work focuses on fine-grained source code changes, code ownership, and other data coming from version control systems. Instead, our visualization describes single-developer sessions, and depicts fine-grained interaction data collected with our DFLOW tool.

V. CONCLUSION

We presented a novel approach to visualize how developers use the UI of their IDE and when and how they perform different development activities such as navigating, writing, and understanding source code. Our visualization enables the identification of main development tracks in terms of usage of IDE UI components like windows, and relates such usages with development activities like edit events, inspection events, and commits. To gather data we used DFLOW, our tool that silently records all fine-grained interactions with the IDE.

We discussed several developer stories from the visualized sessions, that identified both peculiar developer behaviors emerging from the usage of the IDE and their activities, and well known phenomena like the window plague. We proposed a simple classification of the visual features of development sessions in terms of dominant window tracks (Single-Track, Multi-Track, and Fragmented sessions) and Flow between tracks (Sequential or Ping-Pong). By using such a classification, we found that different developers exhibit different behaviors and usages of the UI of the IDE. For example, most developers either work with a Single or no dominant window track, with a strong prevalence of Sequential Flows that may lead to cluttered environments. We want to use the insights gained from our visualizations to reflect on ways to support developers in becoming more proficient in terms of IDE user interface usage. We envision the construction of a recommender system that on-the-fly suggests alternative ways to use the IDE.

Acknowledgements. We gratefully acknowledge the financial support of the Swiss National Science foundation for the project "HI-SEA" (SNF Project No. 146734). We thank all the developers that helped us gathering their data.

REFERENCES

[1] T. Gîrba, A. Kuhn, M. Seeberger, and S. Ducasse, "How developers drive software evolution," in *Proceedings of IWPSE 2005 (8th 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 (11th 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] W. Lidwell, K. Holden, and J. Butler, *Universal Principles of Design*. Rockport, 2003.

[7] M. Csikszentmihalyi, *Flow - The Psychology of Optimal Experience*. Harper Perennial, 1990.

[8] D. Roethlisberger, O. Nierstrasz, and S. Ducasse, "Autumn leaves: Curing the window plague in IDEs," in *Proceedings of WCRE 2009 (16th 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 (3rd 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 (15th International Conference on Program Comprehension)*. IEEE, 2007, pp. 155–166.

[11] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil, "An examination of software engineering work practices," in *Proceedings of CASCON 1997 (8th Conference of the Centre for Advanced Studies on Collaborative Research)*. IBM Press, 1997, pp. 21–36.

[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 (1st 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 (10th 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 (5th International Symposium on Software Visualization)*. ACM, 2010, pp. 35–42.