Visualizing Developers Interactions with the IDE

Master's Thesis submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Master of Science in Informatics
Software Design

presented by
Lorenzo Baracchi

under the supervision of
Prof. Michele Lanza
co-supervised by
Roberto Minelli

June 2014

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Lorenzo Baracchi
Lugano, 16 June 2014

To whom they loved me,
who I would never love back enough.
To whom they hated me,
who I would never hate back enough.

Much learning does not teach
understanding.

Heraclitus

Abstract

Software developers use Integrated Development Environments (IDEs) to cope with the complexity of editing, navigating and understanding software artifacts. IDEs integrate different tools like editors, debuggers, and version control system.

Analyzing the interactions developers have with IDEs is a first step towards building better IDEs that enhance software development processes from the perspective of the software developer himself.
We built the tool HACKNEYED, that provides a set of visualizations about the behavior of software developers while using IDEs. HACKNEYED leverages development sessions recorded with DFLOW. Using them we can analyze the behavior of software developers. By means of these visualizations we discovered some insights on the habits and actions of software developers. For example we have evidence that developers spend considerably more time in understanding software artifacts than editing them.

Acknowledgements

Among all the pages of this document this is the most challenging to write. How can sole words be enough to this purpose?

I want to start by thanking the whole REVEAL group: It has been a pleasure knowing and working with you. In particular, Roberto for his useful help on this hard work!

Professor Lanza, because of his teachings that go far beyond the ”limited scope” and concepts of this engineering field.
Thank you!

My classmates and Friends, met during these years of studying.
Walking on this path would have been harder without you!

My Whole Family. They never stopped believing in me and they always supported me.
I wish I could be better for you all.

To my girlfriend Ebrisa, who carries the burden of loving and understanding me.
I could not imagine a more demanding commitment in life!

To the many I forgot to mention here.

To those who helped me growing up, for the good or for the bad.
I will not forget!

Contents

Figures

Tables

Chapter 1

Introduction

In *The Mythical Man Month* [Bro95], Frederick Brooks describes the process of Software Development as:

> *The programmer like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination. Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures.*

This flexibility is a double-edge sword for the developer, who can build whatever his imagination allows him to build. Although he has to deal with the complexity of navigating, understanding, and remembering these creations.
Difficulties increase when a developer needs to work not only with the creation of his own imagination, but has to deal with the imagination of other developers. It is the common case of any software product that is built today that developers do not work alone, but cooperate with other developers in teams ([GGD07]).

Integrated Development Environments (IDEs) are software programs that integrate multiple tools used by developers like editors, debuggers and version control systems in a single environment. IDEs aim at lightening the complexity of dealing with the flexibility of software systems. Nevertheless, even using these tools, much of the complexity of software development is left to the developer.
There is the need to provide developers with facilities that help them to understand software system, and enhance their productivity in building or restructuring software artifacts.

We present an approach to analyze and understand the behavior of software developers. We use visualization to better understand the interaction of developers with the IDE. We believe that, by analyzing how developers use the current tools, we can gather information to understand which are the challenges that they face,

and provide some useful insights to build better tools for developers.

Researches have tried to approach this problem and proposed some solutions. LaToza et al. [LVD06] found that the biggest problems in understanding code for developers are: understand the rationale behind the code, understand the code that someone else wrote, and understand the history of a piece of source code. Murphy et al. [MKF06] studied how developers invoke commands in the IDE. They found that a large percentage of commands is invoked by key bindings. LaToza and Myers [LM10] tried to assess how much time developers spend in understanding code, by using surveys, laboratory and field observation. Minelli and Lanza [ML13b] proposed an approach to visualize the navigation flow of developers in the IDE.

In this thesis we present a new visual approach to better understand how software developers interact with IDEs. By means of visualizations we analyze the behaviors of different developers to gather insights of the problems they face while working on software systems.

1.1 Contributions

The main contributions of this thesis can be summarized as follows:

- A set of interactive visualizations to better understand the developers' behavior.
- A categorization of development sessions based on the different interactions of the developers with the IDE.
- An empirical investigation about the time software developers spend during various development session.
- A study of the interactions of developers with the Pharo IDE.

1.2 Structure of the Document

The remainder of this document is structured as follows:

- Chapter 2 explains how we process the raw data obtained from DFLOW to obtain a format more suitable for the visualizations .
- Chapter 3 introduces HACKNEYED, the tool we developed to support the analysis, and describes the visualizations.
- Chapter 4 demonstrates how to use the tool HACKNEYED to analyze development sessions and presents our findings.

- Chapter 5 discusses the related work and highlights the differences with our approach.
- Chapter 6 summarizes our work and discusses future work.

Chapter 2

Interaction Data

In this chapter we describe interaction data recorded by DFLOW [ML13a], a tool that observes and records the interactions of developers with the IDE (section 2.1). Section 2.2 describes how we pre-process the data to build the visualizations.
In section 2.3 we explain how DFLOW estimates development activities from the recorded events. In section 2.4 we present the dataset used for the analysis.

2.1 DFlow and interaction data

DFLOW is a tool that silently observes and records the workflow of developers inside the IDE while performing software engineering tasks [ML13a]. In DFLOW there are two main classes of events: UI events (interactions with the IDE windows) and development events (like editing or inspecting the source code). Every DFLOW event has the timestamp representing the exact time when it has been triggered.
UI events happen when a developer opens, activates, resizes, collapses, expands, or closes a window.
For development events we mean when the developer edits, inspects, or navigates a program entity. Editing events are recorded by DFLOW when the user performs a modification of the code or the image, while navigation events happen when the user jumps from browsing an artifact to browse another artifact. Inspection events happen when the developer inspects an object. Table 2.1 shows the complete list of events that can be recorded by DFLOW.

ID	Description
N_1	Opening a Finder UI
N_2	Selecting a package in the system browser
N_3	Selecting a method in the system browser
N_4	Selecting a class in the system browser
N_5	Opening a system browser on a method
N_6	Opening a system browser on 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	Do-it on a piece of code (*e.g.* workspace)
I_4	Print-it on a piece of code (*e.g.* workspace)
I_5	Stepping into in a debugger
I_6	Run to selection in a debugger
I_7	Exiting from an active debugger
I_8	Proceeding in a debugger
I_9	Browsing full stack in a debugger
I_{10}	Stepping over in a debugger
I_{11}	Entering a full debugger
I_{12}	Browsing the hierarchy of a class
I_{13}	Browsing all implementors of a method
I_{14}	Browsing all senders of a method
I_{15}	Closing the current Smalltalk image
I_{16}	Saving the current Smalltalk image as...
I_{17}	Browsing the version control system
I_{18}	Browsing the stack trace in the debugger
I_{19}	Browse versions of a method
E_1	Creating a new class
E_2	Adding/removing instance variables from a class
E_3	Removing a method from a class
E_4	Adding a method in a class
E_5	Remove a class from the system
E_6	Automatically creating accessors for a class
$W_{1,2}$	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

Table 2.1. List of events that can be recorded in DFLOW.

2.2 Data interpretation

To understand the interaction of developers with the IDE we want to analyze the activities that the developer is performing at a certain time. Since DFLOW only records the timestamp when an event is triggered, we need to estimate the duration of activities from the timestamp of events. We developed different estimation models depending on the type of events generated by DFLOW: If they are events on the windows or events generated by the code editor.

2.2.1 Window Events

We estimate the duration of activities by considering two consecutive timestamps. We show 6 basic examples, for each type of window event, in which we consider two consecutive events respectively at time t_1 and t_2. The examples show:

1. **Open:** when at time t_1 there is an open window event, we assign a small duration to the open event (i.e., 1 second), then we consider that window as active until the next event at time t_2, as in Figure 2.1.

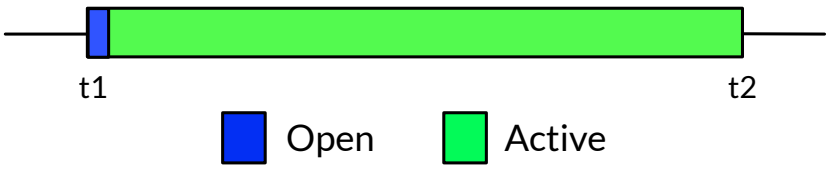


Figure 2.1. Interpretation of an open window event

2. **Activation:** when at time t_1 there is an activation event, we consider the window on which the event happens active until the next event at time t_2, as in Figure 2.2.

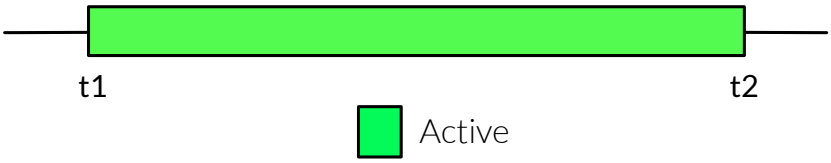


Figure 2.2. Interpretation of an activation window event

3. **Resize/Move:** when at time t_2 there is a resize or move window event, we assign a small duration to that event (e.g. 1 second), then we consider that

window as active until the next event, as in Figure 2.3. This means that the user changed the size or the position of the window, then continued to look at it. It is possible that DFLOW records a series of consecutive move or resize events, in this case we consider the action of resize starting at the first event and ending at the last event.

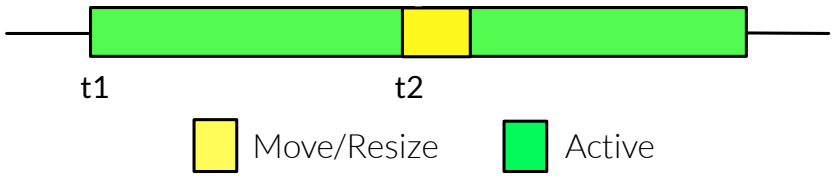


Figure 2.3. Interpretation of a resize/move window evens

4. **Minimize:** when at time t_2 there is a minimize event, we assign a small duration to the minimize event (e.g. 1 second). We know that starting from t_2 the window is not active and not visible on the screen until it is expanded, as in Figure 2.4.

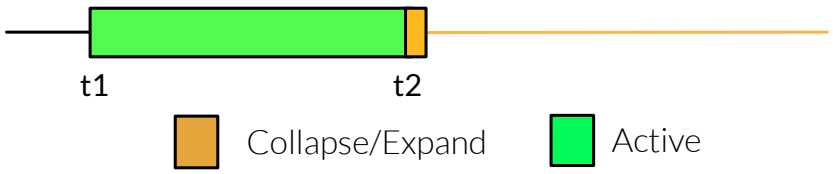


Figure 2.4. Interpretation of a minimize window event

5. **Expand:** when at time t_2 there is an expand window event, we assign a small duration to the expand event (e.g. 1 second), then we consider that window as active until the next event, as in Figure 2.5.

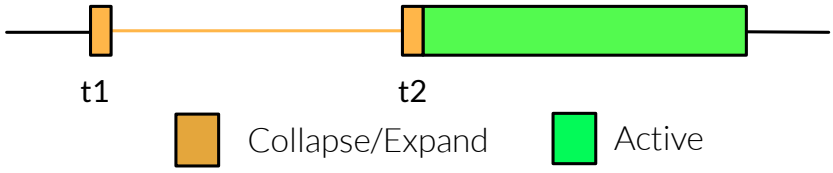


Figure 2.5. Interpretation of an expanding window event

6. **Close:** when at time t_2 there is a close event, we assign a small duration to the close event, e.g. 1 second, as in Figure 2.6 .

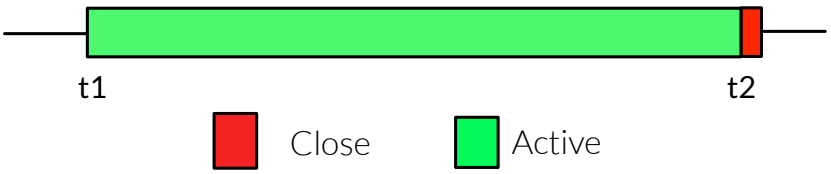


Figure 2.6. Interpretation of a close window event

2.2.2 Development Events

In this section we describe the first naïve approach we devised to estimate development activities from the list of events. This method is rather simple and it is still used in one visualization of HACKNEYED: the Tree Visualization, which was the first attempt of visualization for this work.

There are three important events on the editor that we can use for our analysis: editing, inspection, and navigation. From those events we need to estimate the understanding activities. We consider an understanding activity the time in which the developer is reading the code with the purpose of understanding it. Intuitively, understanding activities can be defined as the time in which the developer is doing neither editing, nor inspection, nor navigation.

We show 3 examples, in which we consider three consecutive events at times t_1, t_2 and t_3.

1. **Editing:** when at time t_2 there is a recording of an editing event, we know that the changes happened between the the time t_2 and the previous event at time t_1, therefore we can consider the duration of the editing activity from time t_1 to time t_2, i.e. $\Delta E = t_2 - t_1$ where ΔE is the duration of the editing activity (Figure 2.7).

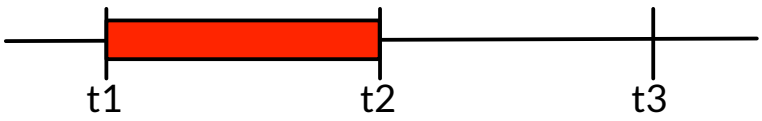


Figure 2.7. Interpretation of edit events

2. **Inspection:** when at time t_2 there is a recording of an inspection event, we know that the developer is looking at the state of the object between the time t_2 and the subsequent time t_3. Therefore we consider the duration of the inspection activity from time t_2 to time t_3, i.e. $\Delta I = t_3 - t_2$ where ΔI is the duration of the editing activity (Figure 2.8).

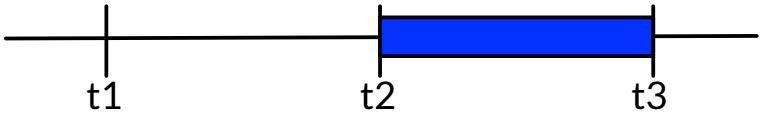


Figure 2.8. Interpretation of inspection events

3. **Navigation:** when at time t_2 there is a recording of a navigation events, we know that the developer navigated to a particular artifact and then looked at it. For this reason we consider a small duration for the navigation (e.g. 1 second) and then from that time until the subsequent event t_3 we consider it as an understanding activity, i.e. $\Delta U = t_3 - t_2 - \Delta N$ where ΔU is the duration of the understanding activity and ΔN is the duration of the navigation activity (Figure 2.9).

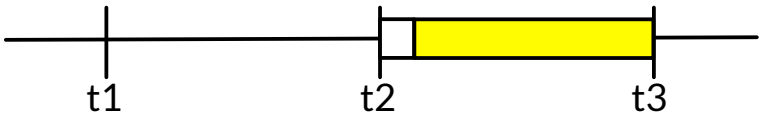


Figure 2.9. Interpretation of navigation events

2.3 Development activities

With DFLOW is possible to estimate the duration of activities in a better way. These estimations were not yet available when we first created the interpretation of events described in section 2.2. In this section we briefly describe how DFLOW estimates the duration of activities.

Figure 2.10 shows an example of interaction history of DFLOW. Where N corresponds to a navigation event, E is an editing, and I is an inspection event.

Navigation: Navigation events correspond to mouse clicks, thus they require a small amount time. DFLOW generates, from navigation events, navigation activities

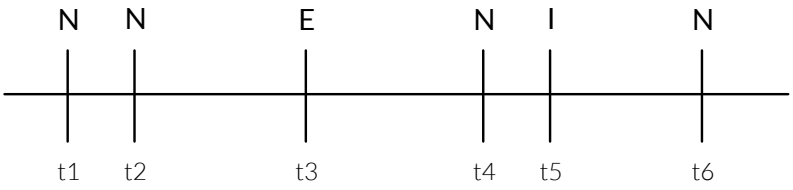


Figure 2.10. Example of DFLOW events

that have a fixed time duration: ΔN. The default value for ΔN is 0.5 seconds.

Editing: DFLOW records editing events at the end of the editing. DFLOW denotes this time as $end(E_i)$. DFLOW assumes that the duration of the edit (ΔE_i) is a fraction (P_E) of the time interval between the end time of the previous activity $end(E_{i-1})$ and the end time of E_i:

$$\Delta E_i = P_E \cdot (end(E_i) - end(E_{i-1}))$$

Inspection: DFLOW records inspection events at the start of the inspection activity: $start(I_i)$. DFLOW assumes that the duration of the inspection (ΔI_i) is a fraction (P_I) of the time interval between $start(I_i)$ and the start time of the following event $start(I_{i+1})$:

$$\Delta I_i = P_I \cdot (start(I_{i+1}) - start(I_i))$$

Understanding: DFLOW then defines as understanding activities, the time gaps that remain after the computation of the navigation, editing and inspecting activities.

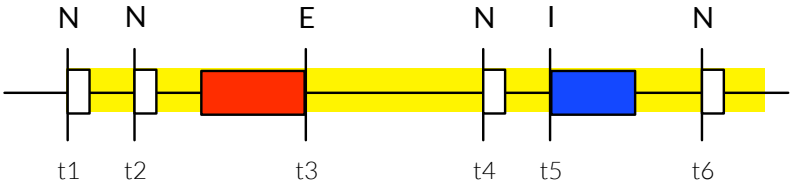


Figure 2.11. Example of DFLOW computed activities

Figure 2.11 shows an example of computed activities starting from the events of Figure 2.10. White boxes represent navigation activities, red boxes represent editing activities, and blue boxes represent inspection activities. In yellow it is displayed the understanding time.

2.4 Dataset

To analyze the interactions of developers with the IDE, we have available about 200 development sessions recorded with the tool DFLOW. These development sessions were recorded by 7 different developers while doing their normal work. Table 2.2 shows simple metrics for the development sessions analyzed.

Developer	Number of Sessions	Average Duration	Events			
			Navigation	Inspect	Edit	Total
Aerys	12	1,906	21,617	183	2,458	24,258
Davos	3	16	393	157	24	574
Luwin	65	102	20,468	2,157	2,091	24,716
Mace	6	1,010	2,183	353	1,196	3,732
Robb	73	272	35,801	2,962	3,316	42,079
Tommen	7	282	6,862	337	472	7,671
Yoren	11	1,888	7,234	486	526	8,246

Table 2.2. Dataset grouped by developers

2.5 Discussion

In this chapter we described the approach used to transform the data available in DFLOW in a format that is easier to use for visualization purpose. The data hereby described are used in HACKNEYED to create the visualizations.

In subsection 2.2.2 and section 2.3, we described two different approaches to extract developer activities from a collection of events recorded by DFLOW for a development session. Section 2.3 tries to estimate the editing time as a fraction of the time elapsed between the two events. The interpretation in subsection 2.2.2 would set an upper bound on the time used for editing.
As as consequence the technique described in subsection 2.2.2 would estimate larger times for editing and inspection activities, but lower times for understandings activities, respect to the times of activities computed with the estimation described in section 2.3.

The estimation of development activities is the first step towards building the visualizations of HACKNEYED. The next chapters describes in details the visualization we developed.

Chapter 3

HacknEyed

The Oxford English Dictionary defines the term *hackneyed* as: *Used so frequently and indiscriminately as to have lost its freshness and interest; made trite and commonplace; stale.* Everyday, developers face the problem of frequently repeat similar activities [Wei85], like navigation through the source code to find a particular function call, or jumping back and forth between the debugger and the code editor to understand the reasons of changes. Hence, we adopted this name for a tool whose goal is to provide visualizations to understand the behavior of a developer interacting with the IDE.

This chapter presents the problems of software developers, and introduces HACKNEYED: the tool that generates the visualizations. Section 3.2 presents the visualizations and section 3.3 describes how to interact with the visualizations. Finally section 3.4 presents the metrics.

3.1 The Problem

Figure 3.1 shows a typical PHARO environment during a development session. PHARO[1] is a pure object-oriented programming language and environment based on the Smalltalk programming language. Differently from most modern IDEs its environment is based on windows instead of tabs. In a typical development session, developers open multiple editor windows, inspector windows, workspaces, and version control windows at the same time to reveal relationships among software artifacts. Doing this they may find themselves opening many new windows or tabs in their IDEs, as in Figure 3.1. Researches call this phenomenon: **Window Plague** [RND09]. The Window Plague can occur both on window based IDEs, like PHARO, and tab based IDEs (see Figure 3.2), like Eclipse.

[1] See http://www.pharo.org/

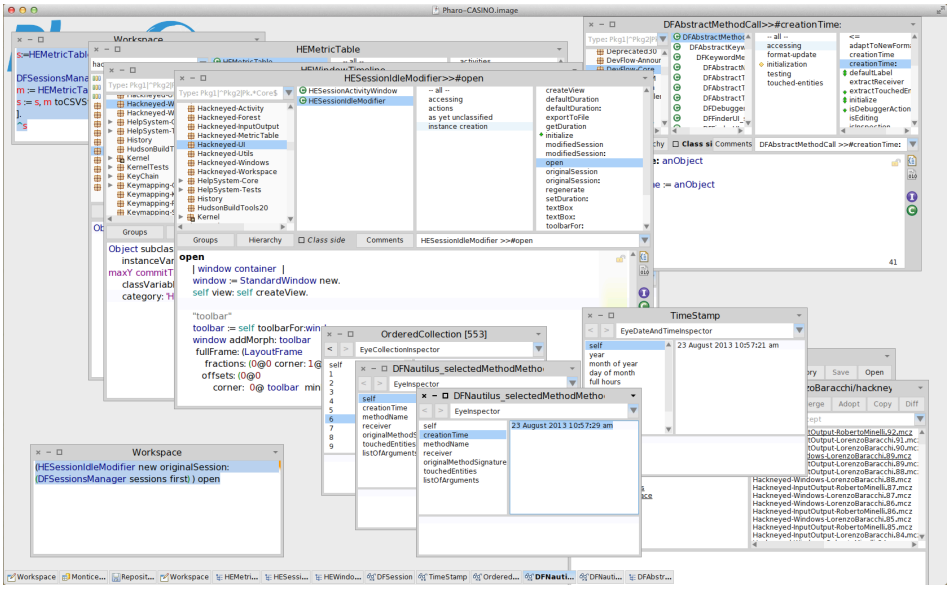


Figure 3.1. A typical PHARO Environment

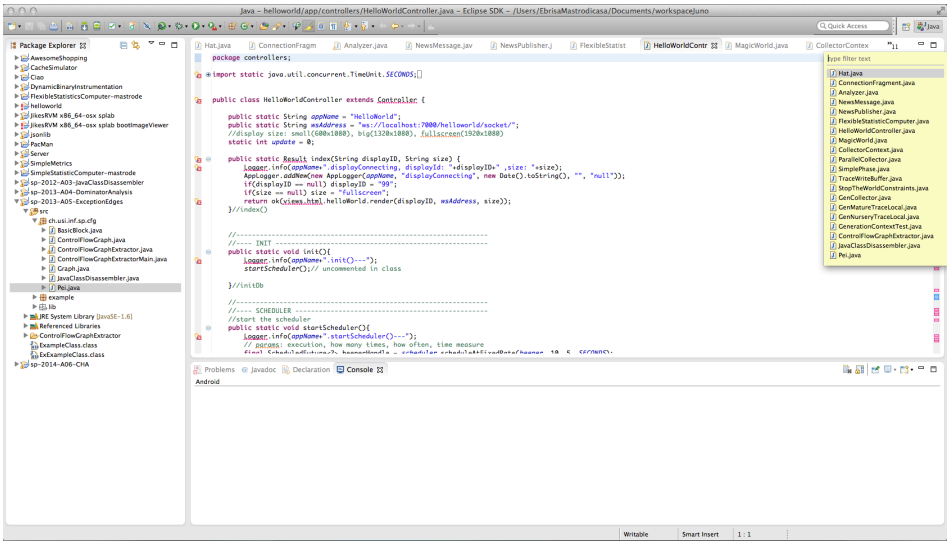


Figure 3.2. A typical Eclipse Environment

To better understand how developers use the User Interface (UI) of the IDE we devised a visual approach that presents interactions data from different perspective. In the remainder of this chapter we present HACKNEYED, the tool that implements our approach, and the visualizations it offers.

By means of visualizations, we devised in HACKNEYED, we can analyze this problem and find some hints on how programmers use the PHARO IDE.

3.2 Visualizations

In this section we present the catalogue of visualizations offered by HACKNEYED. We discuss what they represent and how they are generated.

3.2.1 Tree View

Visualization Principles

The purpose of the *Tree View* is to show the activities of a developer during a development session displayed in form of a tree. The view is built following the structural relationship among the software artifacts on which the activities of the developer happen. Figure 3.3 shows a conceptual representation illustrating the structural relationship used to build the *Tree View*. Class *A1* and *A2* belong to the category A. Methods *m1*, *m2* and *m3* are from class *A1*, while method *m6* is from class *B1*, under category *B*.

We consider the following program entities: methods, classes, and categories (packages). The forest will have as root nodes only categories, on the second level there are the classes, followed by the methods on the third level of the tree. We show only the artifacts that a developer examined during a development session, therefore each category will have as children only those classes that belong to that category and are involved in the session. The same happens for classes and their methods.

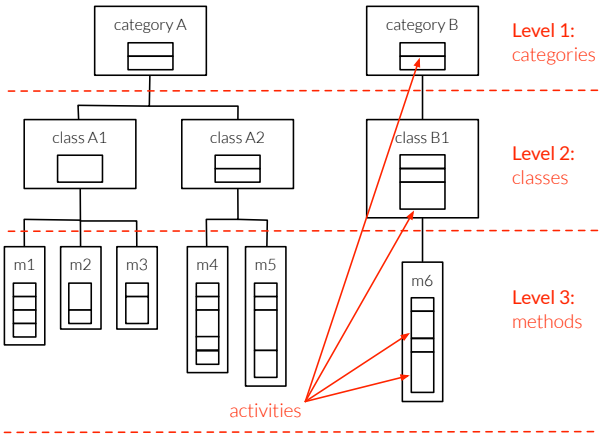


Figure 3.3. Concepts for the Tree View

Every node in the tree contains a list of activities that the developer performed on that artifact. The height of each activity is proportional to the time spent. We distinguish four types of activities: navigation, understanding, inspection and editing. Each activity has a color associated to its type: green for navigation activities, red for understanding activities and yellow for inspection activities. Navigations correspond to clicks of the mouse by the developer, inspection happens when the developer inspects the state of an object at runtime, editing happens when the developer saves the changes on an artifact. Understanding is when the developer is not doing one of the previously mentioned activities, and he is just looking at the code.

Editing activities have a different representation with respect to other activities.

Indeed they have different colors and widths, depending on code size and changes, while the height represents the time spent by the developer on the activity, as for the others. The color of an editing activity is computed on a gray-scale, depending on the size of the code modified in that activity. To implement this we assign the color white to the minimum code size for the considered artifact, and the color black to the maximum code size for that artifact. This is possible since every editing event in DFLOW stores the resulting code after the editing has been performed. The width of the activity is computed based on the impact of a change on the code of the artifact. With impact of a change, we mean the difference on the code size before and after an edit event occurs. For example adding 4 characters to a method will result in an activity with width 4, removing 6 characters from a method will result in an activity with width 6. Figure 3.4 shows an example of different editing events on a Smalltalk method and the resulting activities on the view:

```
sayHello:
  ^'Hello'
```

edit1

```
sayHello:
  ^'Hello World'
```

edit2

```
sayHello:
  "it says hello"
  ^'Hello World'
```

edit3

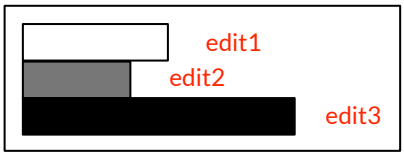


Figure 3.4. Edit activities example

Example

Figure 3.5 shows an example of a *Tree View* generated by HACKNEYED. The figure shows a list of 10 categories represented as trees. Figure 3.5 shows an example of a single tree.

In Figure 3.5 we notice that among all the entities displayed only one method has been edited by the developer. On this method edits happen at different times and the first editing activity is the most important in term of code size changes (the larger gray box).

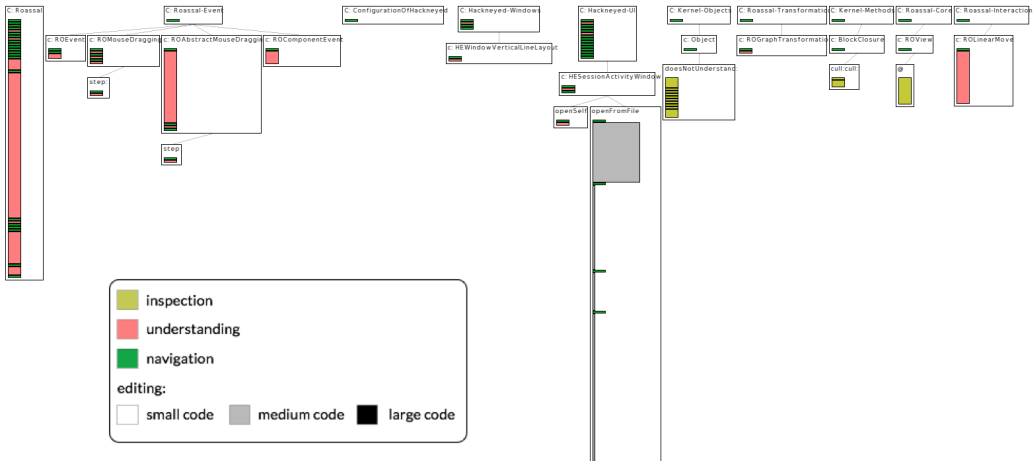


Figure 3.5. An example of a tree view

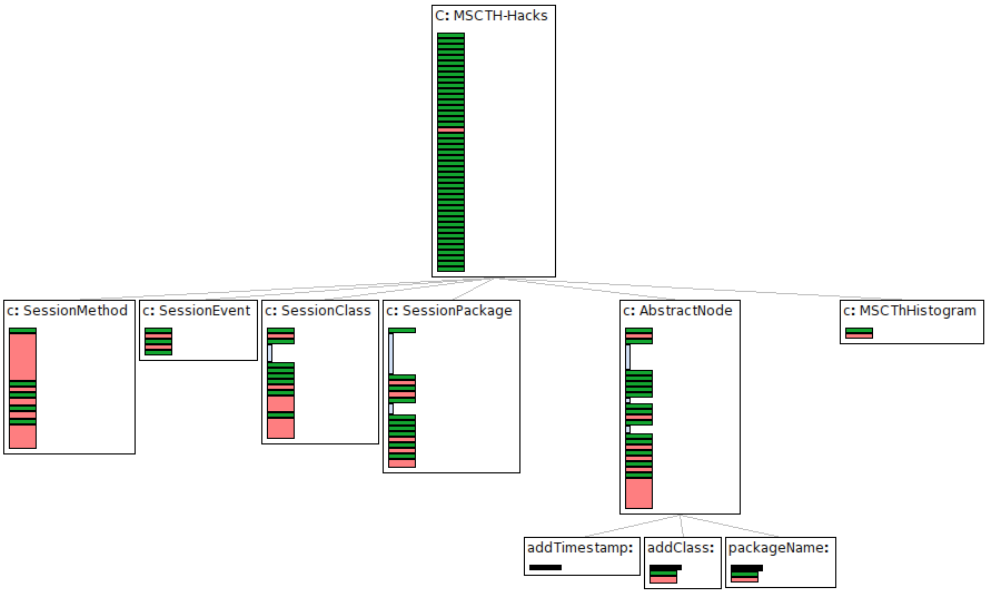


Figure 3.6. An example of a single tree

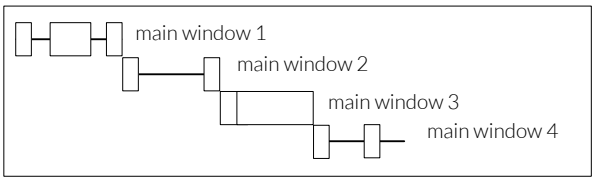
In Figure 3.5 we see that the category at the root of the tree presents a lot of navigation events. For this example, editing activities do not require a large amount of time and do not have large impact regarding code size changes. There are also editing activities on classes, which means that the developer modified the definition of the object and not just the behavior.

3.2.2 Window Activity View

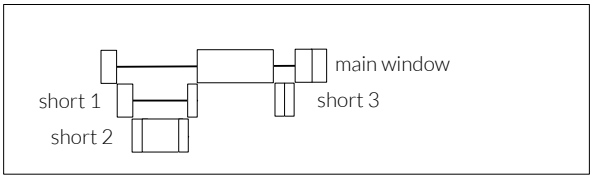
Visualization Principles

The *Window Activity View* shows the activities a developer performed on various windows during a development session. Every window opened during the session is displayed on an horizontal line, which has length equal to the time in which the window was open. We distinguish two types of windows: **main** and **short-windows**. **Main** windows are windows that have a lifetime greater that 1 minute, while **short-windows** are windows that are opened and closed in less than a minute, we can see them as windows containing additional information respect to the main windows from which they are generated.

In Figure 3.7 we show a conceptual representation of the positioning for the *Window Activity View*, where Figure 3.7a shows the positioning of main windows and Figure 3.7b shows the positioning of short windows.



(a) Positioning of main windows



(b) Positioning of short-windows

Figure 3.7. Conceptual representation of the position of windows

Every main window occupies a single vertical coordinate on the visualization. Once a main window takes a position, another window can not use the same vertical position, even if their timespans do not overlap. Short windows are opened and closed in a short period of time, interrupting the workflow of other windows. Therefore we place the short windows under the window on which they have interrupted the workflow. A short window is placed in the first available vertical position below the main window to which it belongs. As a consequence short windows can share the same vertical position under the main window and a short window can be under either a main or a short window.

Figure 3.8 shows an example of a main window together with its short windows that are placed below it.

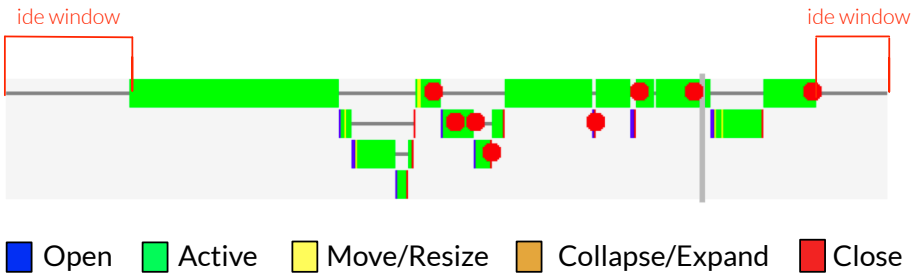


Figure 3.8. An example of main and short-windows

A window, over time, can be opened, closed, activated, resized, moved, collapsed (minimized) and expanded. In our visualization we depict these information with different colors on the timeline of the window. The timeline is a line representing the time in which a window is alive: from the open event to the close event. We draw open events in blue, closed in red, activated events in green, resized and moved events in yellow, collapsed and expanded events in orange. We interpret green bars as time where the developer is focusing his attention on that window. We call windows that are not active or collapsed: idle windows. Idle windows are shown with a thin grey line (see Figure 3.8). This line has an orange color when the window is collapsed, which means that the window still exists, but is not displayed anymore on the developers' screen.

The visualization shows vertical lines in correspondence of subsessions. Subessesions are pauses of the session, when a developer momentarily stops working on the software, and then restarts working, on the same session, after some time. There are two types of subsessions: The explicit subsessions, in which the developer explicitly stops the DFLOW recording by using the interface, and the implicit subsessions, in which the developer does not stop the recording and neither makes any action for long time. We show explicit subsessions as light gray vertical lines, and implicit subsessions as red vertical lines. For every subsession we show the pause time elapsed between the end of a subsession and the start of the next one by means of a label above the vertical line representing the pause. In the visualization we show blue vertical lines in the correspondence of the times at

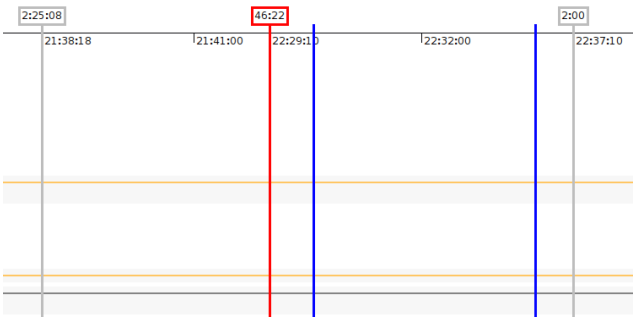


Figure 3.9. Example of the lines for explicit and implicit subsessions, and commits

which a developer committed the changes, in the versioning system, done during the development session. Figure 3.9 shows examples of explicit and implicit subsessions, and the vertical lines for commit events. Moreover in the timeline of windows it is possible to see where the developer did some changes, which are shown as red dots (e.g. Figure 3.8).

Example

Figure 3.10 shows an example of a *Window Activity View* generated by HACKNEYED, and highlights a detail of the top left corner of the example.

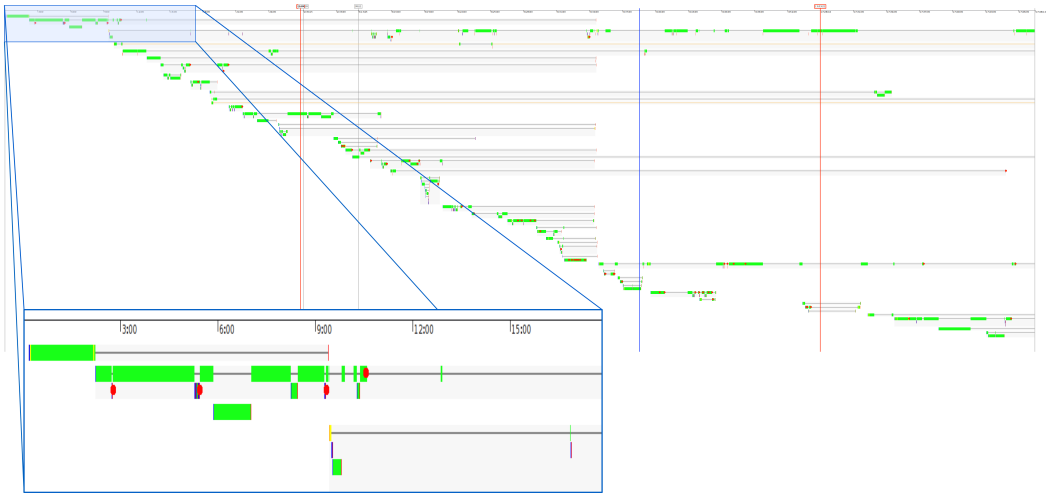


Figure 3.10. An example of a Window Activity View

In this example the developer uses a significant amount of different windows across the session. There are few windows that exist for long time, but for most of the time they are idle, i.e., not active. Two windows get minimized early in the session and then forgotten. Overall, the interaction of the developer with the windows looks fragmented.
The session lasts for about 18 hours but more than 15 hours are pauses between subsessions. There are 4 pauses, two of which implicit. Towards half of the session the developer committed the changes he made.

3.2.3 Activity Views

Visualization Principles

Figure 3.11 shows the conceptual representation for this visualization. We devised two kinds of visualization about activities of a developer over the time of a development session. The first visualization shows the various activities that the developer performs over time, this means that if a developer is editing the code for 5 minutes starting from the 10^{th} minute of development, the view shows an editing activity of 5 minutes starting at that time. Every activity has an associated color: Yellow represents understanding time, red represents editing time, blue represents inspection time, and white represents navigation time. In the visualization we also show a time axis, the title of the session visualized and subsession intervals (the thick vertical lines).

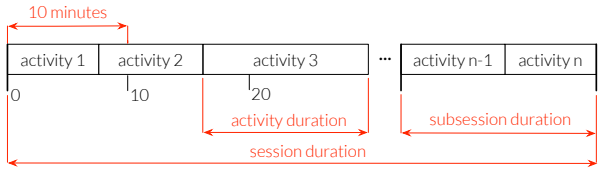


Figure 3.11. Conceptual representation of the Activity view

The second visualization shows the time of all the activities performed by a developer up to a certain time in a cumulative fashion. The first bar represents the activities done in 5 minutes, the second bar represents the activities done in 10 minutes. Therefore, every bar adds 5 minutes of activities to the previous bar, until the end of the session. Figure 3.12 shows the conceptual representation for this visualization. Every bar is composed of 4 elements, which are (from bottom to top): *a*) *N*: navigation time. *b*) *I*: inspection time. *c*) *E*: editing time. *d*) *U*: understanding time. The height of each element represents the overall duration of a type of activity up to that time.

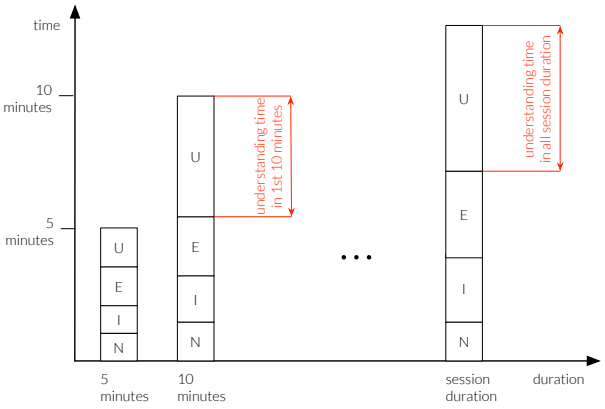


Figure 3.12. Conceptual representation of the Cumulative Activity view

Example

Figure 3.13. Example of the Activity view

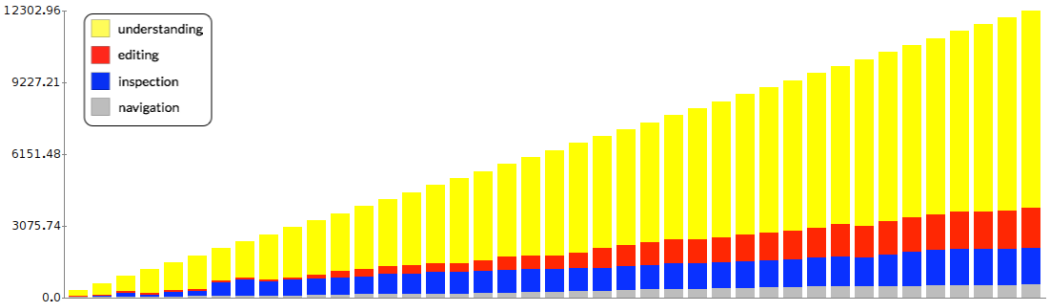


Figure 3.14. Example of the Cumulative Activity view

Figure 3.13 shows an example of an activity view. Figure 3.14 shows an example of a cumulative activity view.

The session represented in Figure 3.13 for the first 10 minutes is mostly dominated by understanding time. The frequency of editing events starts to increase from minute 10. Starting from minute 20 until the end of the session there is the highest concentration of editing events. This means that the developer gathered knowledge at the beginning of the session and edited mostly at the end.

In Figure 3.14 we clearly see that the overall time spent by the developer in editing is around a fourth of the time spent in understanding. The total editing and inspecting time are similar to each other, but the inspection time grows more at the beginning of the session, while the editing time grows more toward the end.

3.2.4 Workspace View

Visualization Principles

The *Workspace View* is an interactive visualization of the workspace of the developer during a development session. As workspace we intend the IDE environment to which the developer interacts.

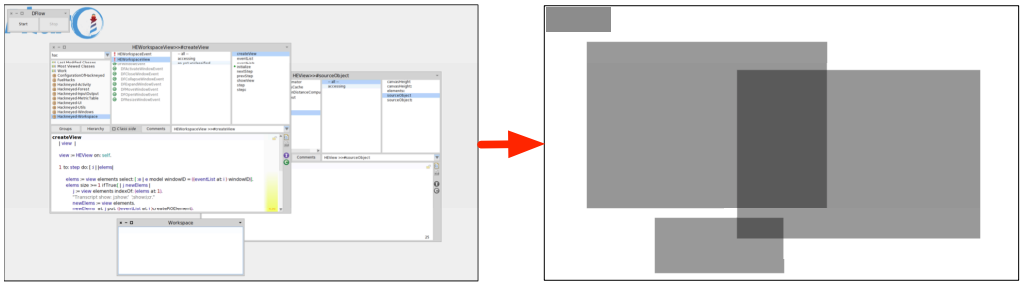
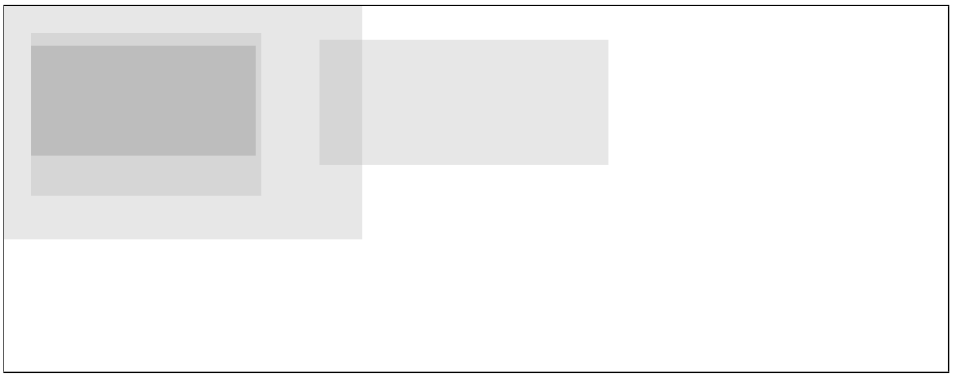


Figure 3.15. Conceptual representation of the Workspace View

Figure 3.15 shows a conceptual representation of the visualization. On the left there is an example of windows inside PHARO IDE, on the right there is the correspondent visualization of the same windows. Both have four windows in the workspace, of which three overlap with themselves. Every window present in the developer workspace is depicted as a gray rectangle with the same size and position of the original window in the IDE. When two or more windows overlap the color of the overlapping area becomes darker. The more a color is darker the more that particular position in the screen contains information for the developer. Hence white areas indicate the absence of windows, light grey areas indicate the presence of few windows, darker areas indicate the presence of a lot of overlapping windows.

In the visualization we show all the windows that are present in the workspace of the developer at a particular time. The user can follow the evolution of the workspace of the developer by changing the time currently displayed. Using this information it is possible to understand how the developer uses the space in the IDE, like: where are placed the most windows or how much windows are moved or resized. This is a useful information to improve window management. For example, a developer that tends to have a messy workspace with many overlapping windows, can benefit of a window manager, while a developer that tends to organize the space following some patterns, could be hampered by a window manager that uses a different approach.

Example



(a) Example of workspace view towards the beginning of a session

(b) Example of workspace view towards the end of a session

Figure 3.16. Examples of workspace view

Figure 3.16a and Figure 3.16b show an example of the workspace view on the same development session at different times.

We see that the developer has an higher number of windows towards the end of the session respect to the number of windows at early times in the session. The developer prefers to maintain windows at the top left corner of the screen, while the bottom right of the screen has fewer windows. It remains to be investigated if in this session there exists main windows, and in which position on the screen they are.

3.2.5 Combined Views

HACKNEYED also offers some combined visualizations to enhance the analysis and understanding of development sessions. Combined visualization are visualization that combine two of the previous discussed visualizations: *Window Activity View* and *Activity view*.

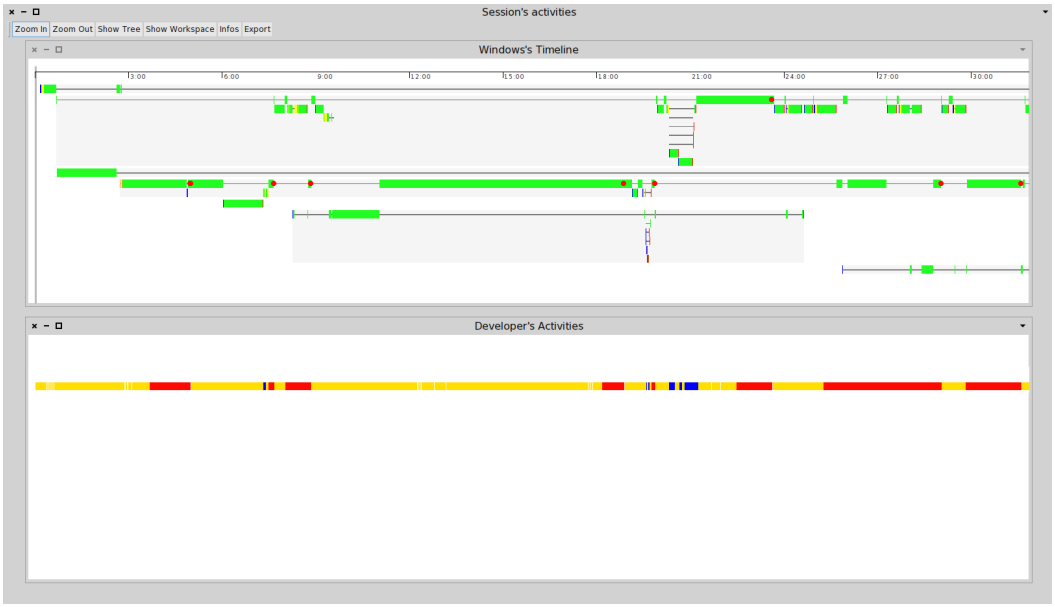


Figure 3.17. First attempt of a combined view

Figure 3.17 shows the first attempt to create a combined view were the window view is placed on top of the activity view, in two different windows.

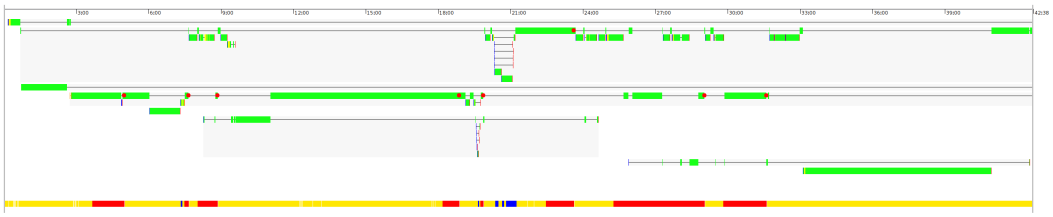


Figure 3.18. Example of a combined view

Figure 3.18 shows an example of a combined visualization where, from top to bottom, there are the window view and the activity view (the combined view also has metrics that are not displayed in the image). Using this visualization is possible to analyze both the activities done by the developer and the actions on the windows, at the same time. It is also possible to look at some metrics about the development session.

3.3 Interaction with the visualizations

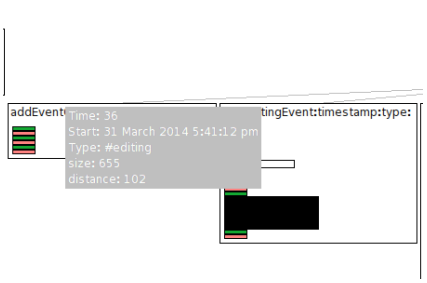
The visualizations described in section 3.2 are interactive, which means that the user can obtain more information by performing actions on them.
In this chapter we describe some interactions that are possible to perform with the visualizations of HACKNEYED.

Basic interaction: All visualizations support basic interactions, such as: pan (changing the view angle), zoom-in and zoom-out.

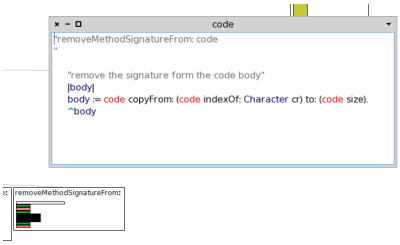
Tree View interactions: On the tree view if the user hovers with the mouse on an event, the view will show a tooltip containing some information:

- Duration of the event and start time of the event.
- Type of the event.
- Code size of the artifact at that time.
- Code difference of the artifact with respect to its previous size.

By clicking on an event the user can see, in an appropriate window, the code of the artifact at that moment.
In Figure 3.19 we show examples of interactions on this view.



(a) The tooltip on a tree event

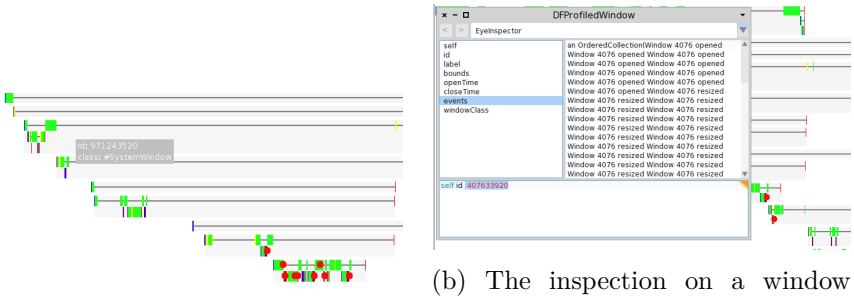


(b) The code of an artifact shown on a tree event

Figure 3.19. Examples of tooltip and code browsing on the tree visualization

Window Activity View interactions: When a user hovers on a window timeline, the view shows a tooltip displaying the window id and the class of window. The window id is an identifier used in PHARO to identify each single window, while the window class is a classification of the type of windows, for example all editor windows belong to the same class and all the inspector windows belong to another class.
When the user clicks on a window timeline it shows a standard PHARO inspector,

in this way the user can interact with the object as any other Smalltalk object; for example he can inspect all the events, or call a method on the window object. In Figure 3.20 we show an example of the interaction on the window activity view.



(a) The tooltip on a window event

(b) The inspection on a window timeline

Figure 3.20. Example of tooltip and inspection on windows

Workspace View interactions: The view shows all the windows open at a certain time in the development session. The user can change the time visualized by the view by clicking on the forward or backward button. It is then possible to see how the workspace of the developer evolves during the session.

Combined View interactions: Combined views inherit the same interactions that are possible in the respective single views. Furthermore they introduce the possibility to open different visualizations starting from them.
They also provide means to tweak some parameter used to create the visualization, such as how much time is considered idle time to create an implicit subsession. We use this parameter to compute the implicit idle time of pauses during the session. For example if the user specifies a 10 minutes of idle time, we process the session to find period of at least 10 minutes where the developer did not interact with the IDE. We then consider that time as a pause time.

3.4 Metrics

This section introduces the metrics that integrate with the various visualizations introduced in section 3.2. We use metrics to enhance the information that can be gathered by inspecting the visualizations.

The metrics we use are reported in the Table 3.1.

Metric	Description
# Subsessions	The total number of subsessions that the considered development session has.
# Explicit Subsessions	The number of explicit subsessions of the development session considered.
# Implicit Subsessions	The number of implicit subsessions of the development session considered.
Duration	The overall duration of the development session considered.
Effective Duration	The duration of the session excluding the time of pause between subsessions.
Explicit Pause Time	The pause time between explicit subsessions.
Implicit Pause Time	The pause time between implicit subsessions.
# Windows	The overall number of windows opened during the development session.
# Navigations	The number of navigations performed by the developer during the considered session.
# Inspections	The number of inspections performed by the developer during the considered session.
# Editings	The number of editings performed by the developer during the considered session.
Time of Navigation	The overall time of navigation in the considered session.
Time of Inspecting	The overall time of inspecting in the considered session.
Time of Editing	The overall time of editing in the considered session.
Time of Understanding	The overall time of understanding in the considered session.

Table 3.1. Description of metrics

3.5 Wrap-Up

In this chapter we briefly presented the architecture of HACKNEYED. We described the various visualizations that HACKNEYED can produce, which are:

- Tree visualization
- Window Activity View
- Activity Views
- Workspace View
- Combined Views

We also described the metrics that integrate information in the various visualizations.

In the next chapter we use the visualization we described to perform analyses of development sessions. We discuss examples of analysis on two development sessions and we present a categorization of development sessions that is based on the interaction of the developer with the IDE.

Chapter 4

Telling development stories with HacknEyed

In this chapter we present two analyses of development sessions performed with HACKNEYED (section 4.1). We present a categorization of development sessions based on the interaction of the developer with the windows in the IDE, and will report findings and statistics about the analyses we performed (section 4.2).

4.1 Analysis

In this section we present some examples of analyses of development sessions performed using HACKNEYED. We collected about 200 development sessions from 7 developers. Here we analyze two sessions presenting different characteristics and from two different developers. For privacy issues, we will refer to developers with fictional names.

4.1.1 An Uninterrupted Session

The first session we analyze is a session from a developer to which we will refer as *Luwin*. The session lasts for about an hour and does not contain any pause, neither explicit nor implicit. Therefore, we believe that this was a full-immersion session where *Luwin* did not get distracted during development and managed to complete his tasks in a rush.

In Figure 4.1 we show the cumulative chart for the activities done during the session (the y-axis shows the time in seconds). This chart reflects the metrics which show that the time of understanding is about 31 minutes while the time of editing is circa half of it (16 minutes). Then, 7 minutes are dedicated to inspection and about 1 and half minute is spent in navigation. This means that in this session the developer actually spent only around a third of his time in writing code.

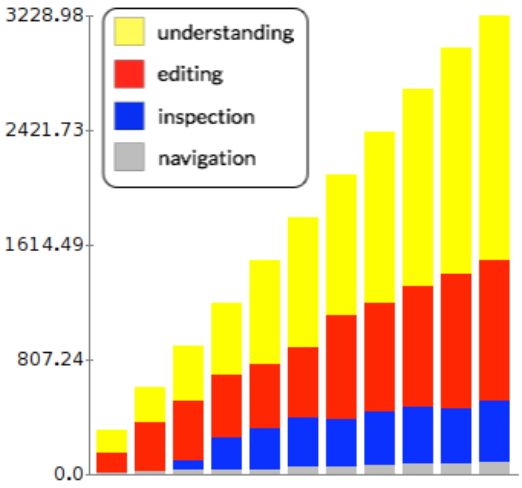


Figure 4.1. Cumulative activities for the Uninterrupted Session

In Figure 4.2 we show the combined visualization of the window and activity view (we show the figure horizontally oriented to improve readability). The session presents two commits towards the end of the session (denoted by the blue vertical lines) which indicate that the programmer considered the changes done during the development session good enough to be versioned.
From the visualization we notice the existence of as single track behavior: That is the presence of a predominant window which is active for most of the time and on which most of the editing activities happen.
This session counts 85 windows in total, of which only a few last for a significant amount of time, but only one that shows the presence of editing events. The latter is the main window of the session.
Considering this information we can conclude that *Luwin* used a single window for editing, while the other windows are mostly used to gather knowledge about the system in order to perform the changes.

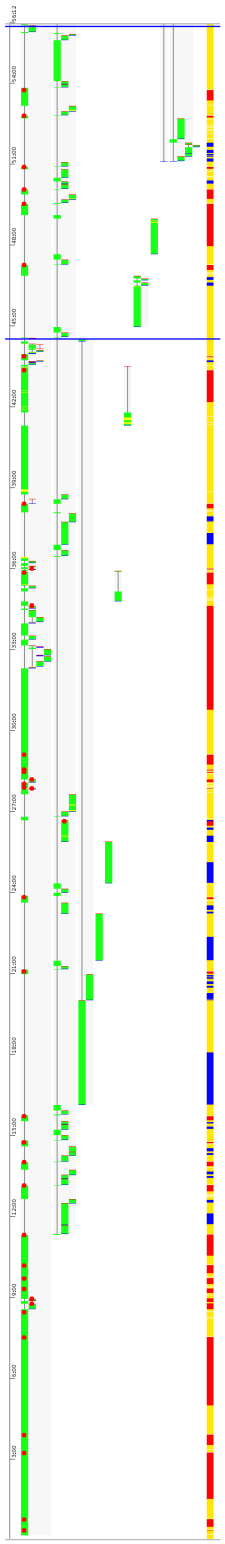


Figure 4.2. Windows and activity visualization for the Uninterrupted Session

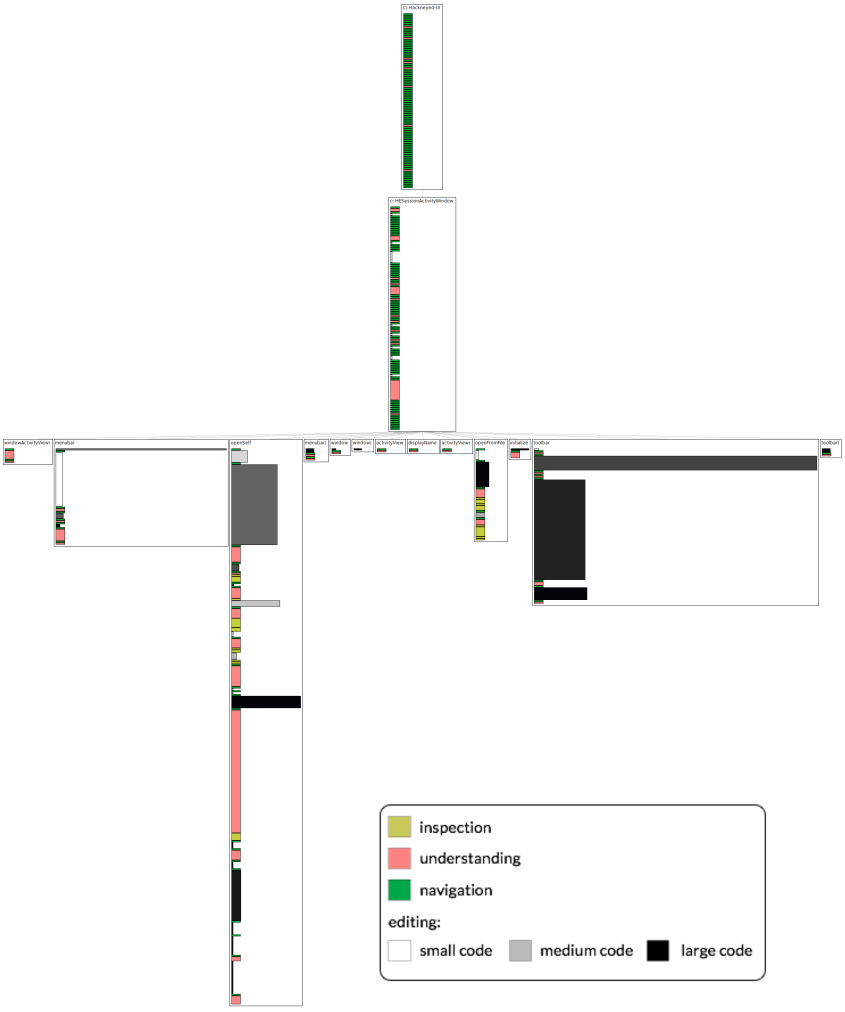


Figure 4.3. Tree view with editings for the Uninterrupted Session

Figure 4.3, Figure 4.4, and Figure 4.5 show parts of the Tree View for the considered development session. The trees shown in the visualizations represent source code artifacts that *Luwin* edited, inspected, or just browsed during this development session.
The first tree (in Figure 4.3) confirms what we already saw with the window view. Indeed this is the only tree that shows editing events. This means that the developer was focused on changing a single class of the system, namely the class `HESessionActivityWindow`.
We see that some methods are edited multiple times over the session, and in the case of the method `openSelf`, editing events are interleaved with inspections and understanding events. This could be due to the fact that *Luwin* performed some changes, then inspected the state of an object at runtime to see if the changes had

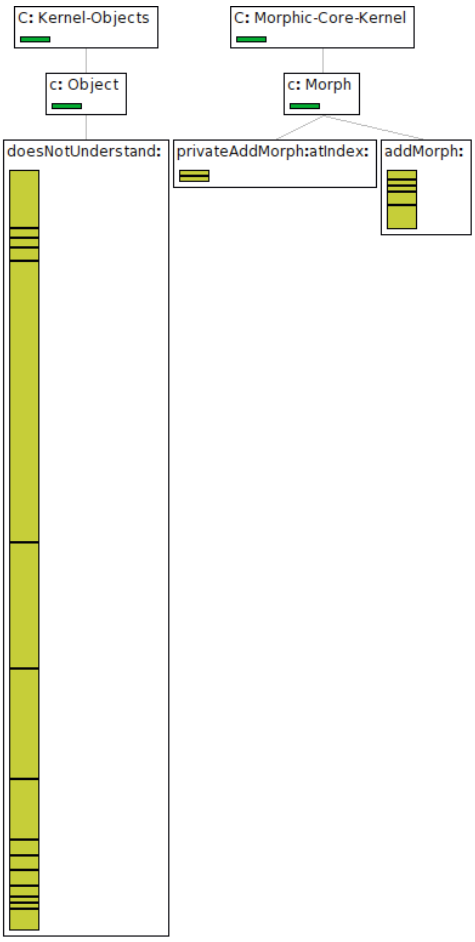
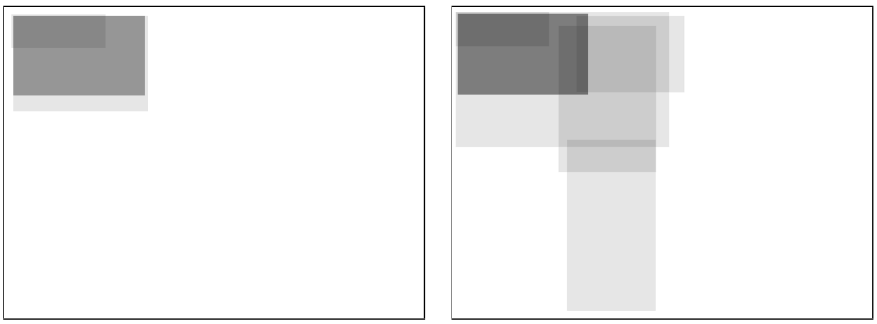


Figure 4.4. Tree view with inspections for the Uninterrupted Session

the effects he wanted.
In Figure 4.4 we notice two trees on which the developer only inspected the code. The first tree is particular since it has a large number of inspection events on the method `doesNotUnderstand` of the class `Object`. These events corresponds to errors appearing at runtime when an object does not have a method that is called on it. This aspect enforces the idea of a developer that makes some changes, then tests them by running the edited code. If he gets an exception, he then needs to investigate on it.
In Figure 4.5 there are some small trees that mostly contain navigation and understanding events. These trees represent classes that could have some structural relationships with the code the developer is editing. An important number of these trees are relative to standard Pharo libraries to create GUIs. This could mean that *Luwin* is seeking for information on the correct way to add certain objects to his

Figure 4.5. Tree view with understandings for the Uninterrupted Session

program interface. Indeed the title of this session: *"Adding Toolbar"* gives a clue on the purpose of the session and on the purpose of understanding and inspecting GUI creation methods.



(a) The workspace for the Uninterrupted Session at an early time

(b) The workspace for the Uninterrupted Session at a later time

Figure 4.6. Workspace of the developer at two different times.

Figure 4.6a and Figure 4.6b show *Luwin* workspace at two subsequent times in the considered session. This visualization tells us the moves of the windows inside the IDE. In this case we notice that the developer prefers to maintain windows in the upper right corner of the screen, and only sometimes he moves them towards other locations.

4.1.2 An Interrupted Session

The second session we analyze is a session from a developer to which we will refer as *Robb*. The session lasts for about 3 hours, of which 1 hour is a pause and the remaining 2 hours is actual development. The pause comes after just 20 minutes of work. This session is identified as a *bug-fixing session*, i.e., a session where the goal of the developer is to fix some unwanted behavior of the software system.
In Figure 4.7 we show the cumulative chart for the development activities done during the session. This chart reflects the metrics which show that the time of understanding is around 1 hour and 20 minutes, the time of editing is less than 25 minutes and time of inspection and navigation are respectively 19 and 2 minutes.

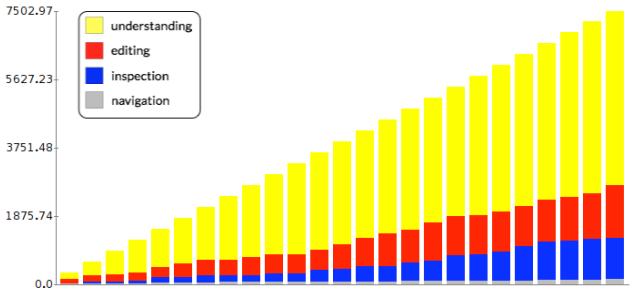


Figure 4.7. Cumulative activities for the Fragmented Session

We notice that the times of editing and inspections are very close to each other, while the time of understanding is approximately 4 times larger than them.
The large discrepancy between the time of editing and the time of understanding could be determined by the nature of the session. Indeed a bug-fixing session can require a deep understanding of the code and its behavior.

In Figure 4.8 we show the combined visualization of the windows and activity view (we show the figure horizontally oriented to improve readability). We can see that the number of windows is significant (226 opened windows during the session). Interestingly *Robb* does not focus for long time on a single window, instead he quickly jumps from a window to another to obtain information. Edit events are also distributed among many different windows. The result of this behavior is a very fragmented flow of actions.
Another particular aspect is that windows either have a short life, or they remains idle for long time. Few windows are minimized, and then never considered again. One window is minimized then used again by the developer.

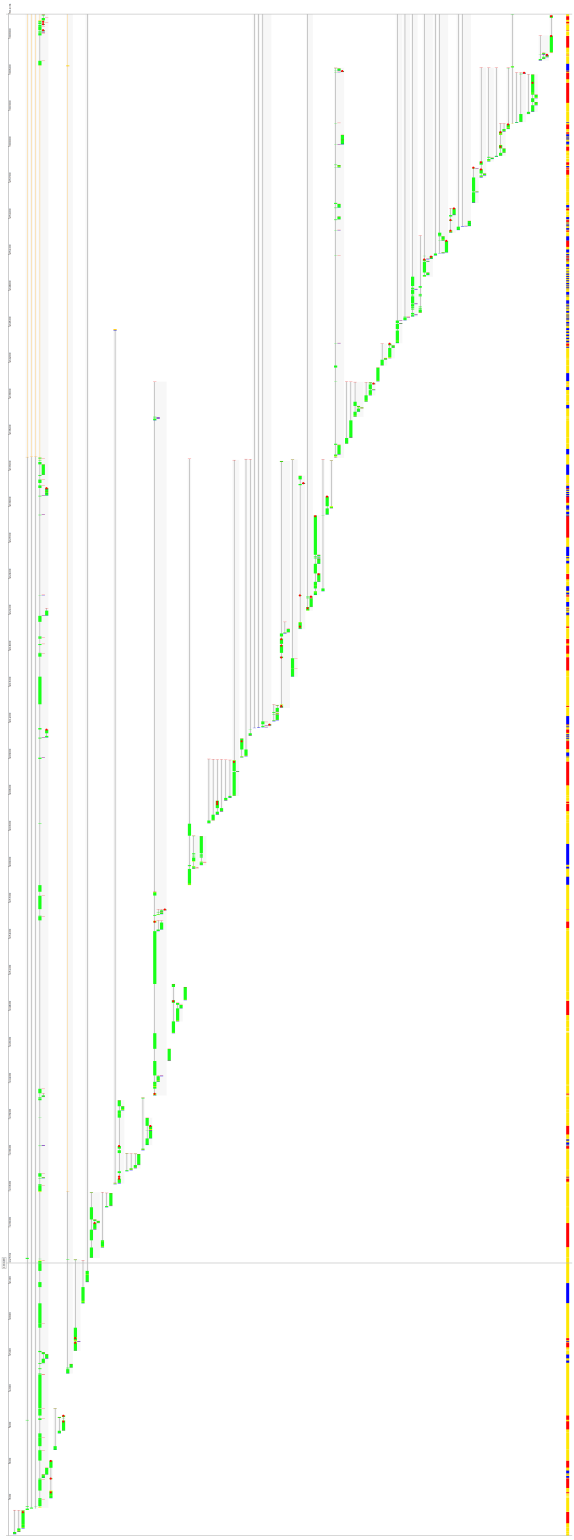


Figure 4.8. Windows and activity visualization for the Interrupted Session

At regular intervals, *Robb* closes a significant amount of windows in a short period of time. This could be seen as an attempt to clean the workspace, or prevent it to become too much crowded with windows. Researches refer to this phenomenon as the window plague [RND09]. In Figure 4.9 we highlight two occurrences of this behavior.

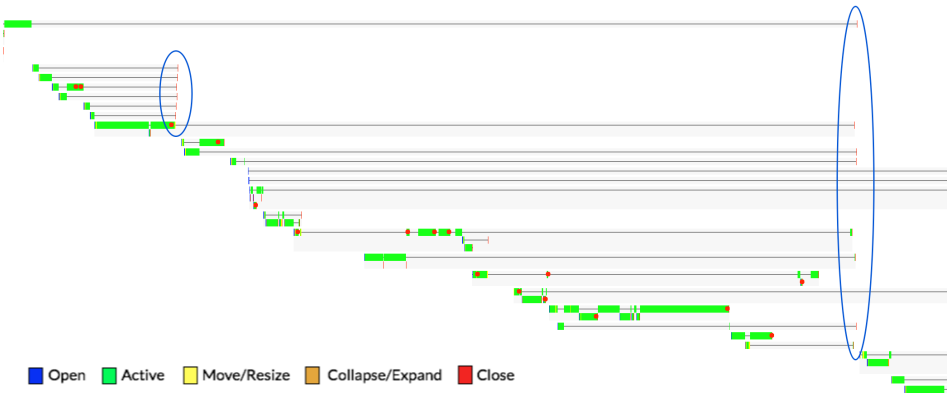


Figure 4.9. Curing the window plague

Interestingly, this session does not present any commit event. This could mean two different things: either the session was completely unproductive, and the user decided not to commit any changes, or the changes where committed after the DFLOW session was ended. It remains to be investigated which of the two hypotheses is correct.

In Figure 4.10, Figure 4.11, and Figure 4.12 we show some parts of the tree visualization for the considered session. Figure 4.10 and Figure 4.11 show two trees where all of the editing events happens during the session. One difference of the two trees is that the second contains a relative bigger amount of inspection respect to the first tree. The first tree contains only one class, while the second tree contains three different classes. It is interesting to notice that editing and inspection events can be interleaved on the same artifact, which could mean that the developer edited the code, then looked at the effects of the changes at runtime, then modified again the code to obtain the desired behavior.

From this visualization we know that the developer modified about a dozen of different methods. If we interrelate this information with the visualization in Figure 4.8 we clearly see that the number of windows on which an edit event happens is bigger than the number of code artifacts modified. This means that *Robb* has a tendency to close the windows which contain the source code that needs to be modified, forcing himself to spawn other windows focused on the same code later on. Otherwise *Robb* does not easily found the window containing the code to be modified, thus he opens a new one to make changes.

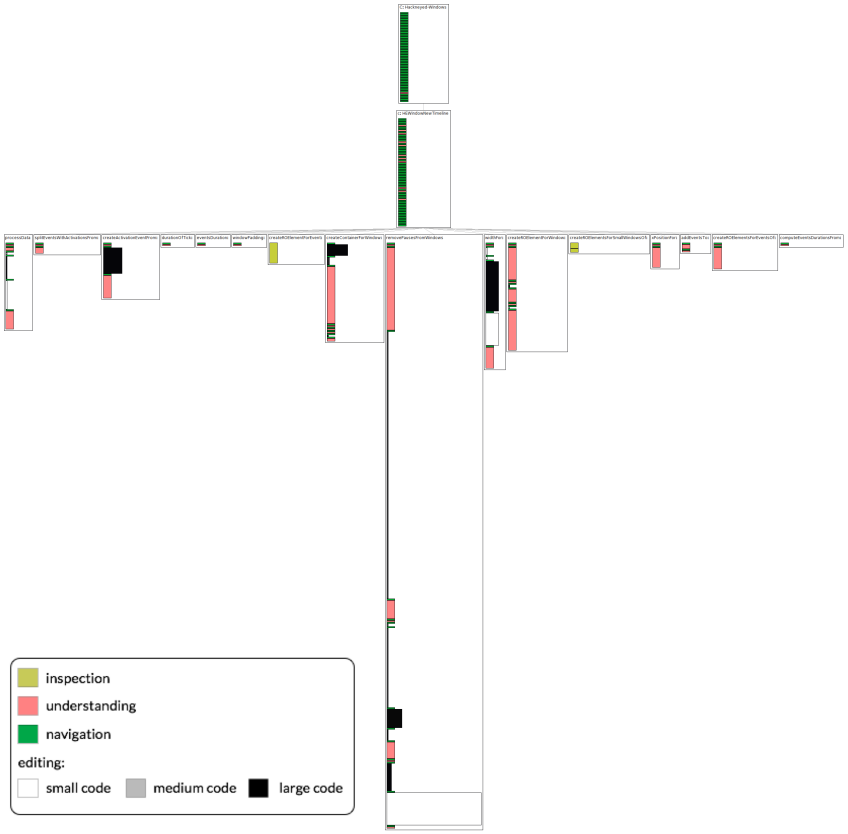


Figure 4.10. Part of tree view for the Interrupted Session

Figure 4.12 shows a group of small trees with a predominance of inspection events. Those trees mostly refer to standard libraries like `Collection` or `TimeStamp`. These types of trees are very common when the developer tries to understand the state of objects on which he is working on. A recurrent inspection event happens on the method `haltIf`, which is used to interrupt the runtime environment at a particular point. These events indicates that the developer is performing a debugging task.

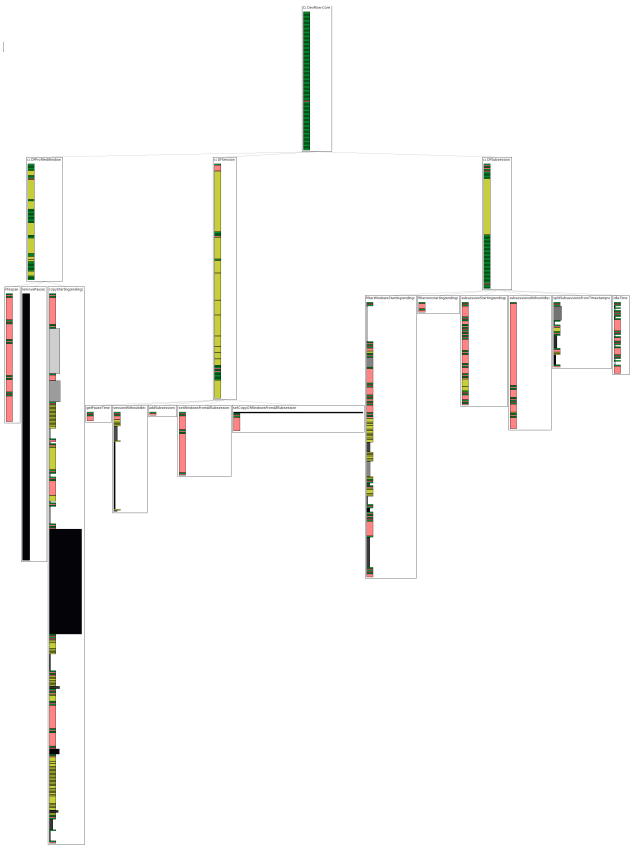


Figure 4.11. Part of tree view for the Interrupted Session

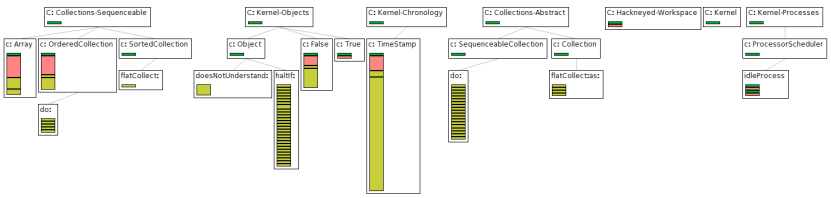
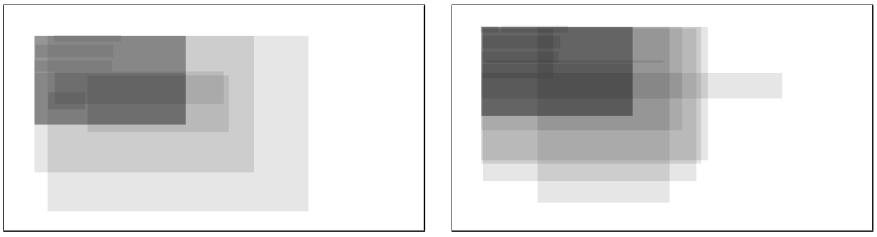


Figure 4.12. Part of tree view for the Interrupted Session

Figure 4.13 depicts the state of the workspace of the developer at two different times. In this session, *Robb* tends to keep all the windows in the same area of the workspace. This could constitute a problem if the number of windows grows too large, because having many windows in the same area makes it difficult to find a particular window that contains what the developer needs.
This could explain the need of closing windows across the session, and the editing events on the same artifact distributed across many windows.



(a) Workspace for the Interrupted Session at an early time

(b) Workspace for the Interrupted Session at a later time

Figure 4.13. Workspace of the developer at two different times.

4.1.3 Wrap-Up

In this section we presented two analyses of two different development sessions from two different developers. We used all the visualizations provided by HACKNEYED to analyze the interactions of the two developers with the IDE.
We saw two different developers' behaviors: The first (subsection 4.1.1) where the number of window is small and the edits are mostly performed on a primary window, while the second (subsection 4.1.2) presents a large number of windows with a fragmented activity on them.

4.2 Categorization

In this chapter we describe a categorization of development sessions based on the interaction of the developer with the user interface of the IDE, and we report findings and statistics about the analyses we performed.
This work has been submitted to VISSOFT 2014 [MMLB14].

4.2.1 Principles of Characterization

We used the Window Activity View of HACKNEYED to classify sessions recorded with DFLOW. We classified sessions based on similar characteristics, in particular, we used the presence of dominant tracks of windows and the type of flow among different tracks to define the types of session. A track o windows is a group of windows dominated by a main window and where the other windows, which have a shorter duration, act as a support of information for the main window.

Dominant Tracks: A dominant track is a window which presents a predominant focus in the session and is often used by the developer to perform edits. In Figure 4.2 we show an example of a dominant track of windows.
We can devise three types of categories by analyzing the presence of dominant tracks in the visualization:

1. **Single-Track:** there exists a single predominant track of windows. See Figure 4.14.

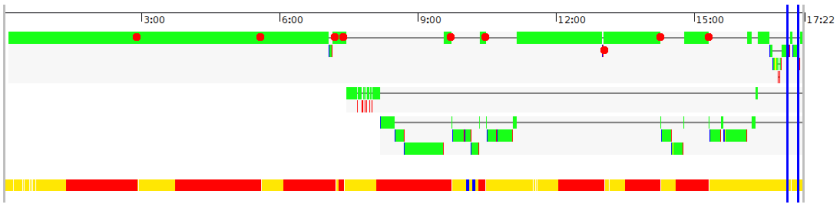


Figure 4.14. Example of single-track window

2. **Multi-Track:** there exists two or more predominant tracks of windows. See Figure 4.15.

Figure 4.15. Example of multi-track window

3. **Fragmented:** there are no predominant tracks of windows, and the developer often changes focus from one window to another. See Figure 4.16.

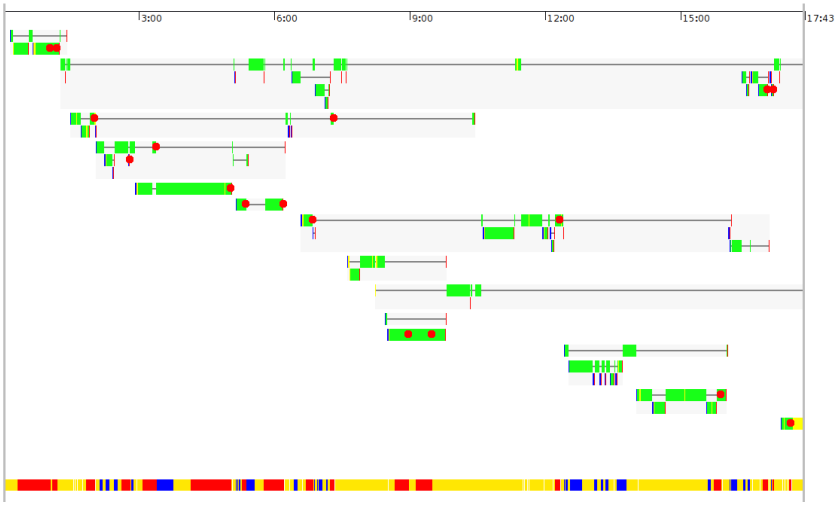


Figure 4.16. Example of fragmented track window

Track Flow: we define the track flow as how the developer alternates his focus among different windows. We can devise two behaviors based on the track flow:

1. **Sequential Flow:** this type of flow is characterized by the developer moving from one track to the next in a sequential way, and rarely come back to the previous tracks. See Figure 4.17

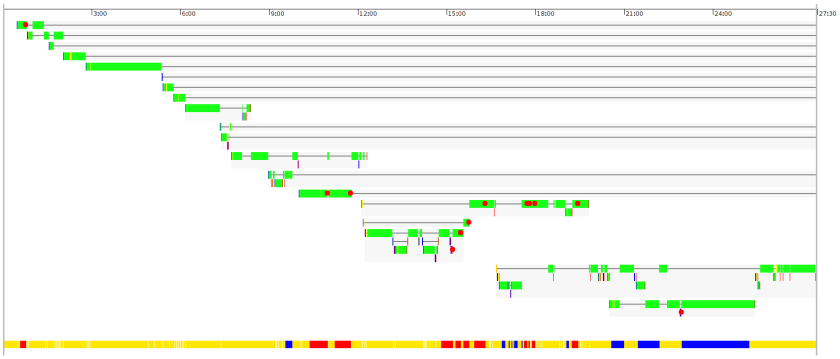


Figure 4.17. Example of sequential flow

2. **Ping-Pong Flow:** the development flow continuously goes from one track to another and back. This behavior could create a fragmented view, especially if no dominant track is present. See Figure 4.18

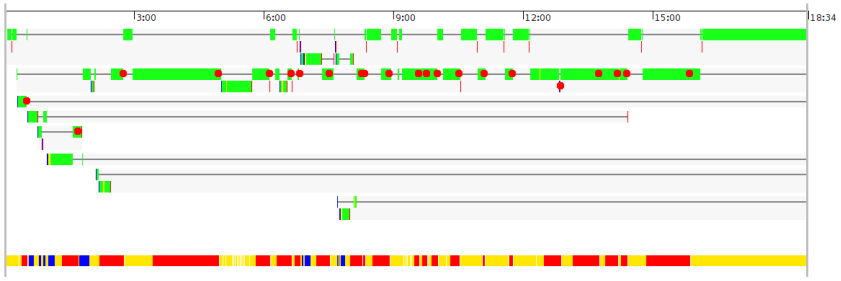


Figure 4.18. Example of ping-pong flow

4.2.2 Characterization of Development Sessions

In this section we present some statistics on the collection of development sessions we have. We analyzed development sessions from 7 different developers.

Sessions Categories

Table 4.1 groups the session using the categorization presented in subsection 4.2.1, we categorized a sample of 164 session, discarding 13 session that were too short or with few events.
Single and Fragmented tracks sessions are more frequent than multi-track session. Sequential flow sessions are more frequent than ping-pong flow sessions, but ping-pong flow is relatively more frequent in single and multi-track session, than in fragmented sessions.

	Sequential Flow	Ping-Pong Flow	Total
Single-Track	50	19	69
Multi-Track	18	7	25
Fragmented	67	3	70
Total	135	29	164

Table 4.1. Categorized sessions

In Table 4.2 we show the categorized development sessions for each developer. The column # reports the number of sessions for developer. The columns *Single*, *Multi*, and *Frag* refer respectively to single-track, multi-track, and fragmented sessions. The columns *Seq* and *PP* refer to sequential-flow session and ping-pong flow session.

Developer	#	Track						Flow				Not classified	
		Single		Multi		Frag		Seq		PP			
Aerys	12	1	8 %	1	8%	8	67%	9	75%	1	8%	2	17%
Davos	3	2	67%	0	0%	1	33%	3	100%	0	0%	0	0%
Luwin	65	45	69%	10	15%	8	12%	45	69%	18	28%	2	3%
Mace	6	0	0 %	0	0%	6	100%	5	83%	1	17%	0	0%
Robb	73	13	18%	10	14%	41	56%	60	82%	4	5%	9	12%
Tommen	7	2	29%	2	29%	3	43%	4	57%	3	43%	0	0%
Yoren	11	6	55%	2	18%	3	27%	9	82%	2	18%	0	0%

Table 4.2. Categorized development sessions for developers

Luwin and Robb are the developers that contributed the highest number of sessions, while for Davos we have little data to characterize his behavior. Single-track sessions are common for Davos, Luwin, and Yoren. Luwin is the developer having most of them for both quantity and average. Fragmented track sessions are common for Aerys, Mace, Robb and Tommen, with Mace having all his 6 sessions categorized as fragmented, and Robb having the highest number of them: 41. With the exception of Tommen, most sessions of all the developers can be categorized as having a sequential flow rather than ping-pong flow. This is particularly interesting since these type of sessions can possibly suffer from the so called **Window Plague** [RND09]. The Window Plague is when a developer opens an high number of windows on different artifacts in order to reveal relationship among code entities. This window plague can lead to a crowded workspace for long development sessions. The Window Plague can occur both on window based IDEs, like Pharo, and tab based IDEs, like Eclipse.

Average times of activities for developers

Developer	Dur	E. Dur	E. Pause	I. Pause	% Edit	% Insp	% Nav	% Und.
Aerys	1,906	181	1	1,723	24.71%	3.78%	3.68%	67.80%
Davos	16	16	0	0	16.37%	12.00%	4.51%	66.63%
Luwin	102	52	19	31	25.22%	8.09%	3.03%	63.54%
Mace	1,010	48	6	957	15.42%	7.94%	2.88%	73.50%
Robb	272	52	204	15	21.42%	7.65%	4.02%	66.82%
Tommen	283	85	132	66	8.30%	6.46%	4.69%	80.49%
Yoren	1,888	141	0	1,747	11.77%	5.34%	1.99%	80.78%
All	782	82	52	648	17.60%	7.32%	3.54%	71.37%

Table 4.3. Average times for developers

In Table 4.3 we report some statistics about the average time each developer spends for the session, or different activities. In the table there are:

- *Dur*: the overall duration in minutes of the session, pauses included.
- *E. Dur*: the effective duration in minutes of the session excluding the pauses.
- *E. Pause*: the explicit pauses duration in minutes. These are the pauses actively triggered by the developer by using the DFLOW interface.
- *I. Pause*: the implicit pauses duration in minutes. These are the pauses we computed after observing a long period of inactivity by the developer (the default is 10 minutes).
- *% Edit* the percentage of time a developer spend on editing respect to the effective duration of the session.
- *% Insp* the percentage of time a developer spend on inspecting respect to the effective duration of the session.
- *% Nav*, the percentage of time a developer spend on navigating respect to the effective duration of the session.
- *% Und* are the percentage of time a developer spend on understanding the code respect to the effective duration of the session.

The table reports the average values across all the session collected for each developer. In the last row, there are the average values from all the sessions.

In average the effective duration for a session is about 1 hour and 20 minutes, but the average duration of sessions varies for each developer. Pause times are different for each developer, especially for the implicit pauses which range from zero or few minutes to more than a day (Aerys and Yoren). We can explain these differences by the interpretation of what a session is. A development session recorded by DFLOW can be identified by a title that represent the purpose of the work that will be performed. This induces the developer to create a relation between session with a specific task, which ultimately depends on how a developer defines a task. There could exists micro tasks, like *"adding a button to an interface"*, or macro tasks, like *"create a new visualization"*, or even no specific tasks like *"working on the system"*.

It is important to analyze the average percentage of editing and understanding time. We clearly see that overall the editing time counts for less than 20%, while the understanding time is more than 70%. Past researches ([ZSG79], [FH83], [Cor89]) stated that programmers spend half of their time editing source code, while the other half in understanding it. The data reported show that actually the editing time can be much lower than 50% of the duration of a development session. It follows that understanding time is much larger: in average more than 70%.

Category	Dur	E. Dur	E. Pause	I. Pause	% Edit	% Insp	% Nav	% Und.
single	191	53	12	125	26.05%	6.39%	3.47%	63.99%
multi	343	69	46	227	21.92%	6.28%	3.34%	68.40%
fragmented	33	18	6	9	14.57%	4.06%	4.01%	77.24%
sequential	619	72	184	363	20.24%	9.89%	3.40%	66.38%
ping-pong	168	71	62	35	24.77%	7.57%	3.45%	64.16%

Table 4.4. Activity times for the categorized sessions

Average times of activities for categorized sessions

In Table 4.4 we report the average time spent by developer in the development session categorized using the categories definitions described in subsection 4.2.1.
We notice that for the track flow categorization the average time spent in various development activities, like editing and understanding, varies for few points in percentage. Nevertheless, we have an higher editing time for ping-pong session than for sequential sessions.
For the categorization based on the dominance track of windows we notice that there is an important difference in the editing time. Indeed the fragmented sessions have less than 15% of editing time, which compared with the 26% of editing time of single-track sessions it makes a big difference. Also the understanding time for fragmented sessions is around 77%, while single track sessions and ping-pong flow sessions have an average time of understanding of about 64%. We can deduce that software developers are more productive if they can focus their attention on few main sources of information (windows) rather than having to gather information from various multiple sources.

Sessions by type

In Table 4.5 we report statistics about the sessions divided by the type of the session that the developer assigns to it when the recording starts.

Type	#	Dur	E. Dur	E. Pause	I. Pause	% Edit	% Insp	% Nav	% Und
Enhancement	91	349	55	83	212	24.23%	7.84%	3.82%	64.02%
Bug-Fixing	25	213	43	39	131	21.07%	11.71%	2.49%	64.57%
Refactoring	7	56	43	12	0	25.41%	0.80%	5.75%	67.89%
General	61	954	103	120	731	17.12%	6.47%	2.93%	73.38%

Table 4.5. Sessions divided by session type

We notice that the percentage of inspecting time is higher in the bug-fixing sessions, while it is lower in the refactoring sessions. We can explain this by the fact that inspecting the state of an object is a fundamental action when the developer is fixing unexpected behaviors (bugs). In a refactoring session the developer is more

focused on improving the quality of the source code, and should not modify the behavior of objects.

General sessions are the session where the developer did not know how to classify them, hence he assigned a general purpose type. These sessions, respect to the other three types, have an average higher duration and pause times, but they also have a lower percentage of editing time. This is a clue that if a developer has a clearly defined task he can be more proficient than when he has not a clear goal.

4.3 Discussion

In this chapter we described how HACKNEYED can be used to analyze the behavior of developers when interacting with the IDE. We presented a categorization of development session based on the interactions of the developer and discussed some statistics. Our data shows that the editing time can be much lower than 50% of the duration of a development session, that was the hypothesis of past researches ([ZSG79], [FH83], [Cor89]). Our approach gave evidence that the average editing time for a development session is less than 20% of the overall time. It follows that the time used by developers to understand the software system is around 70%. This means that on average a developer spend much more time in understanding source code than editing it.

Chapter 5

Related Work

In this chapter we discuss related work divided in categories: Section 5.1 presents studies about the behavior of developers, Section 5.2 presents studies about how developers interact with the IDE.
Our approach can be categorized under two areas of software engineering: Reverse Engineering (Section 5.3) and Software Visualization (Section 5.4).

5.1 Behavior of Developers

There are two main methods used to analyze the behavior of developers: Analyze data gathered from versioning system, and collect data using laboratory studies, observations, and questionnaires.

Versioning systems Greevy et al. and Girba et al., performed studies of how developers collaborate to build a software system [GGD07] [GKSD05]. They focused their studies on defining the division of responsibilities of developers and on establishing the relation between features and developers.
They found that the evolution of a software system presents behavioral patterns regarding the contribution of various developers. For example they defined a *monologue* period, in which a developer makes most of the changes, or a *teamwork* period, where many developers frequently commit changes, or a *silence* period.

In our work we also analyze the behavior of developers while building software systems. We try to establish behavioral patterns that one or more developers present during their work.
Differently to related work, we do not perform an analysis on the data collected from a versioning system, but we use more fine grained data about development sessions. With this approach we know which part of the system a developer has modified, which part of the system was only accessed but not modified, and the

interaction with the development environment (e.g interaction with windows). This information is not available on the data collected from a standard versioning system, which only records the changes that a developer decides to commit.

Laboratory study, observation and questionnaires LaToza and Myers analyzed how developers spend their time by using observations and surveys [LM10]. They found that developers spend a significant time in understanding code, rather than editing it.

LaToza, et al. used surveys to classify the major problems developers have when they comprehend source code [LVD06]. For example, two of the main problems reported are understand the rationale behind a piece of source code and understand code that someone else wrote.

Ko et al. conducted a laboratory experiment to better understand how developers gather information that are necessary to make changes to a software system [KMCA06]. They found, that on average developers edited an unfamiliar source code for a fifth of the time. Also, they found that developers spend about 35% of their time navigating between relevant code fragments.

Singer et al. conducted a study on the time software developers use for various practices [SLVA97]. They used questionnaires, and various observations to fulfill this purpose. Although they did not use a precise measure for the time taken by various developer activities, they noticed that the time spent on writing source code is less than the time spend on other activities, like debugging or searching.

One of the purposes of this work is to establish similar metrics on the work of software developers. Rather than focusing our analysis on the data collected by questionnaires or observations, we use data from recordings of developers interactions with the IDE. Using this data, we are able to estimate the time a developer spends for a particular activity.

5.2 Interaction with IDEs

To understand how developers interact with IDEs, researches recorded and collected data about interactions, like invoked commands or keystrokes. Then, they analyzed this data to better understand the behavior of developers.

Yoon and Myers developed FLUORITE, a tool that can record low-level source code events in the Eclipse IDE [YM11]. By analyzing the recorded data, they found that editing source code is different than editing documents. For example some of the most used keystrokes by programmers are the backspace and arrow keys. Which are used to delete text and move inside the editor. These are evidences that editing code is different than editing text document, where for example the backspace keystroke is less used.

Murphy et al. tried to establish how much relevant is the usage of plugins in the Eclipse IDE [MKF06]. They collected data using the Mylar framework. They found that a large percentage of commands is invoked using key bindings by the developers. This is especially true for the most frequent commands, while the least frequent commands are mostly invoked from the menus.

Robbes and Lanza presented SPYWARE, a tool that records the changes performed by a developer in the editor[RL08].

Minelli and Lanza devised DFLOW, a tool that observes the workflow of developers inside the PHARO IDE while performing software engineering tasks, and records all the actions performed [ML13a]. In this work we rely on the data collected by DFLOW to perform our analysis.

In contrast with the first two works presented, we analyze the data at a higher conceptual level. This means that we estimate what are the actions of a developer from the data collected, rather than analyzing the keystrokes or how a particular command is invoked.

5.3 Reverse Engineering

Reverse Engineering is the process of analyzing a subject system to: Identify the system's components and their interrelationship, and create representation of the system in another form or at higher level of abstraction [CC90].
With this work we can reproduce information about which components of a software system a developer edited or inspected during a development session. We can analyze how a component is modified during development.

Robbes and Lanza introduced the concept of development session, as the time in which a developer actively modifies a software system [RL07]. They state that a development session contains valuable information for program comprehension, which are then lost on the versioning system. Using this 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 assert that a session follows an incremental logic of changes, i.e., a developer starts by introducing basic concepts that are then progressively extended during the session.

In this thesis, we focus on understanding development sessions. In particular we try to assess which actions developers take to modify and understand a software system.

5.4 Software Visualization

"Exploring information collections becomes increasingly difficult as the volume grows"([Shn96]); the goal of visualization is to ease the understanding of large amount of information. Software visualization is a specialization of information visualization focusing on software [Lan03].

Lanza and Ducasse introduced the concept of polymetric view [LD03]. This is a lightweight software visualization technique enriched with software metrics information. Polymetric views help to understand the structure and detect problems of a software system. Girba et al. use visualizations to build "Ownership Maps" for software systems developed by multiple developers [GKSD05]. Ogawa and Ma propose a visualization to show interactions between developers of a project during the development [OM10].

In this work we use visualizations to analyze and understand the behavior of developers when they interact with the IDE. Contrarily to Girba and Ogawa we do not depict interactions among different developers, but we focus on the analysis of a single development session and the interaction between the developer and the IDE.

5.5 Summing up

In this chapter we have presented some approaches related to our work, describing similarities and differences between our work and previous studies. To contextualize our work, we briefly introduced two areas of software engineering: Reverse Engineering and Software Visualization.

Understanding the interactions between developers and their development environment is a fundamental step towards building better tools for developers.
In this work we use software visualization to analyze the behavior and interactions of software developers, by using visualizations.

Chapter 6

Conclusions

In this chapter we summarize our work and we discuss future work.

6.1 Summary

Software development is a time consuming activity. To write source code, a software developer needs to find, read, and understand the relevant parts of the code base of the system. Past researches ([ZSG79], [FH83], [Cor89]) indicated that software developers spend half of their time editing source code, and half understanding it.

Integrated Development Environments (IDEs) are the most common tools used by software developers when working on software systems. The purpose of IDEs is to help developers to handle the complexity of software systems, but it is not clear how much they can ease development, or in which ways software developers use them. There exist various researches that try to understand the interactions between developers and IDEs ([SLVA97], [KMCA06], [LVD06], [MKF06], [LM10], [YM11], [ML13a]).

With this work we want to visually analyze how software developers use the IDE to comprehend and build complex software artifacts. Analyzing the interactions developers have with IDEs is a first step towards building better IDEs that enhance software development processes providing the software developer with supports in understanding software systems. To build better IDEs we should consider the analysis of interactions as an insight about problems developers face with the current tools.
We used data about development sessions from different developers collected with the tool DFLOW. We built the tool HACKNEYED, that provides various visualization abut development sessions (chapter 3).
We use visualizations of HACKNEYED to analyze the collected data about development session. We categorized them by analyzing patterns that they present, like

dominant tracks and *track flows* (subsection 4.2.1).
We extracted metrics from the collected data to improve the estimation on the time spent by developers in editing and understanding source code (subsection 4.2.2).

6.2 Future Works

Growing Sample: We currently have around 200 development sessions from 7 different developers. Having more data would allow to refine our analysis, and possibly find other behaviors that were not covered by the considered sample.

Improve estimation: Future versions of DFLOW will include data about keystrokes and mouse movements performed by the user. These data can be use to ameliorate our visualizations or derive new ones.

Improve analysis: We concentrated our analysis more on the combined visualization of windows and activities (subsection 3.2.5). It would be possible to extend the other visualization and have a more complete analysis.

Support for Versioning System: Adding data collected by versioning systems can improve the analysis. For example it can improve the data about commits, or can add the notion of code ownership to the analysis.

Evaluate Tools: Some tools like Autumn Leaves try to reduce the window plague [RND09]. We could use visualizations from HACKNEYED to evaluate if these types of tools are really useful for the developer.

Ameliorate Window Management: PHARO does not perform any management of windows by default. From the analysis of the workspace visualizations the effect of the window plague is visible. It is possible to ameliorate existing windows management systems by exploiting the availability of run-time information about the interaction of developers with windows.

Development sessions and Evolution of the System: It wold be possible to use the information about development sessions combined to the study of the evolution of a software system. In this way we could understand how each development session affects the evolution. For example we could identify which of the changes performed in a session are actually committed and those who were not.

Understanding the understanding: With this work we have evidence that the time needed for a developer to understand the code has been underestimated. It remains to asses which are the causes of this large times. It is possible to study which source code artifacts require larger understanding time for a developer session, and maybe establish a relation with some common software practice like: absence of comments, missing design patterns, relation with other artifacts, etc.

6.3 Epilogue

We used these visualizations to better understand the behavior of software developers when programming.
We have evidence that the common knowledge of a developer editing source code for half of his time and understanding the source code for the other half of its time, was largely underestimating the problem of understanding the source code. Indeed we found that on average a developer spends less than 20% of the time editing code. As a consequence developers spend around around **70%** of their working time in understanding the source code.
We hope that this evidence will motivate new researches on the problems software developers face each day while working on software systems. By analyzing the causes of the large understanding times we could build better software environments that effectively help developers in the task of gathering knowledge about the software system, hence help developers in their work.

Appendix A

The architecture of HacknEyed

HACKNEYED is a plugin in the PHARO IDE. PHARO[1] is a pure object-oriented programming language and environment based on the Smalltalk programming language. Differently from most modern IDEs its environment is based on windows instead of tabs. Moreover the language is not file based, but image based. It follows that a developer can browse a single method in a window rather than the entire code of a class.
HACKNEYED depends on the tool DFLOW, for gathering the data about developer interaction, and it uses ROASSAL which is a visualization engine for PHARO.
DFLOW is a tool that silently observes and records the workflow of developers inside the IDE while performing software engineering tasks [ML13a]. We use the recorded data form DFLOW to create the visualizations and perform analysis on it.
ROASSAL[2] is a visualization engine for Pharo. With Roassal is possible to produce interactive visualizations for arbitrary data.

The conceptual architecture of HACKNEYED is rather simple. It is composed of:

- **Pre-Processor**, a component responsible to gather the raw data from DFLOW and transform them in something that is easier to visualize.
- **View Generator** which is the component responsible for building the various visualizations.

[1] See http://www.pharo.org/
[2] See http://objectprofile.com/Roassal.html

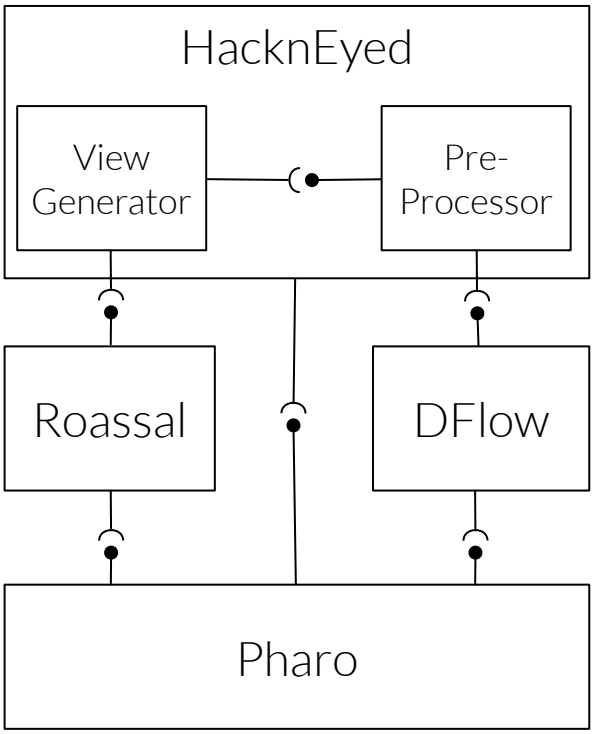


Figure A.1. Architecture of HACKNEYED

Bibliography

[Bro95] F.P. Brooks. *The Mythical Man-month: Essays on Software Engineering*. Essays on software engineering. Addison-Wesley, 1995.

[CC90] E.J. Chikofsky and II Cross, J.H. Reverse engineering and design recovery: a taxonomy. *Software, IEEE*, 7(1):13–17, Jan 1990.

[Cor89] T. A. Corbi. Program understanding: Challenge for the 1990's. *IBM Syst. J.*, 28(2):294–306, June 1989.

[FH83] R. K. Fjeldstad and W. T. Hamlen. Application Program Maintenance Study: Report to Our Respondents. In *Proceedings GUIDE 48*, April 1983.

[GGD07] Orla Greevy, Tudor Gîrba, and Stéphane Ducasse. How developers develop features. In René L. Krikhaar, Chris Verhoef, and Giuseppe A. Di Lucca, editors, *CSMR*, pages 265–274. IEEE Computer Society, 2007.

[GKSD05] Tudor Girba, Adrian Kuhn, Mauricio Seeberger, and Stephane Ducasse. How developers drive software evolution. *Principles of Software Evolution, International Workshop on*, 0:113–122, 2005.

[KMCA06] Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Softw. Eng.*, 32(12):971–987, December 2006.

[Lan03] Michele Lanza. Object-oriented reverse engineering coarse-grained, fine-grained, and evolutionary software visualization, 2003.

[LD03] Michele Lanza and Stéphane Ducasse. Polymetric views – a lightweight visual approach to reverse engineering. 29(9):782–795, September 2003.

[LM10] Thomas D. LaToza and Brad A. Myers. Developers ask reachability questions. In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, editors, *ICSE (1)*, pages 185–194. ACM, 2010.

[LVD06] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 492–501, New York, NY, USA, 2006. ACM.

[MKF06] Gail C. Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the eclipse ide? *IEEE Softw.*, 23(4):76–83, July 2006.

[ML13a] Roberto Minelli and Michele Lanza. Dflow – towards the understanding of the workflow of developers. In *SATToSE 2013 (6th Seminar Series on Advanced Techniques & Tools for Software Evolution)*, 2013.

[ML13b] Roberto Minelli and Michele Lanza. Visualizing the workflow of developers. In *Proceedings of VISSOFT 2013 (1st IEEE Working Conference on Software Visualization)*. IEEE CS Press, 2013.

[MMLB14] Roberto Minelli, Andrea Mocci, Michele Lanza, and Lorenzo Baracchi. Visualizing developer interactions (under revision). In *Proceedings of VISSOFT 2014 (2nd IEEE Working Conference on Software Visualization)*. IEEE CS Press, 2014.

[OM10] Michael Ogawa and Kwan-Liu Ma. Software evolution storylines. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, pages 35–42, New York, NY, USA, 2010. ACM.

[RL07] Romain Robbes and Michele Lanza. Characterizing and understanding development sessions. In *ICPC*, pages 155–166. IEEE Computer Society, 2007.

[RL08] Romain Robbes and Michele Lanza. Spyware: A change-aware development toolset. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 847–850, New York, NY, USA, 2008. ACM.

[RND09] David Röthlisberger, Oscar Nierstrasz, and Stéphane Ducasse. Autumn leaves: Curing the window plague in ides. In Giuliano Antoniol, Martin Pinzger, and Elliot J. Chikofsky, editors, *WCRE*, pages 237–246. IEEE Computer Society, 2009.

[Shn96] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings of the 1996 IEEE Symposium on Visual Languages*, VL '96, pages 336–, Washington, DC, USA, 1996. IEEE Computer Society.

[SLVA97] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. An examination of software engineering work practices. In *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '97, pages 21–. IBM Press, 1997.

[Wei85] Gerald M. Weinberg. *The Psychology of Computer Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1985.

[YM11] YoungSeok Yoon and Brad A. Myers. Capturing and analyzing low-level events from the code editor. In Craig Anslow, Shane Markstrum, and Emerson R. Murphy-Hill, editors, *PLATEAU*, pages 25–30. ACM, 2011.

[ZSG79] Marvin V. Zelkowitz, Alan C. Shaw, and John D. Gannon. *Principles of Software Engineering and Design*. Prentice Hall Professional Technical Reference, 1979.