Measuring Navigation Efficiency in the IDE

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

***Abstract*—While coding, developers construct and maintain mental models of software systems to support the task at hand. Although source code is the main product of software development, the process involves navigating and inspecting entities beyond the ones that are edited by the end of a task. Developers use various user interfaces (UI) offered by the Integrated Development Environment (IDE) to navigate the complex, and often hidden, relationships between program entities. These UIs impose fixed navigation costs, in terms of the number of interactions that a developer is required to perform to reach an entity of interest. It is unclear to what extent the actual navigation effort differs from an ideal setting, and if there is any room for actual improvement.**

We present a preliminary empirical study, where we analyzed a corpus of IDE interaction data coming from 6 developers totaling more than 20 days of development activity. To measure the navigation efficiency, we compute a combination of different ideal settings and compare them against the observed navigation events. Our findings reveal that, on average, developers perform 1.5 to 19 times more navigation events than the ideal case. While different factors make the ideal setting unfeasible, we believe that this calls for novel approaches to support the navigation in integrated development environments.

I. Introduction

Developers build and need to maintain complex software systems, composed of several parts which often they did not write. To support their development tasks, and in particular to support program understanding, they implicitly construct mental models of the subject software system, *e.g.,* [1], [2].

According to Storey *et al.*, a mental model *"includes a mapping from the specified program goals to the relevant parts of the implementation"* [2]. While it is common ground that *source code* is the main product of software development, the development process involves a larger set of program entities than the ones that are edited and delivered, *i.e.,* committed to the version control system, by the end of the task, *e.g.,* [3], [4]. Developers often use various user interfaces (UIs) offered by the Integrated Development Environment (IDE) to navigate the complex and often implicit and hidden relationships between program entities [5]. This process imposes some fixed costs, for example, in terms of the number of clicks that a developer is required to perform to navigate the system structure and reach the code of a particular program entity. The actual navigation effort, *i.e.,* the real amount of navigation events performed by developers, differs from the ideal settings. Researchers, in fact, proposed different approaches to support and improve the efficiency of the navigation through software entities [6], [5]. However, it is unclear *to what extent* the actual navigation effort differs from an ideal setting, *i.e.,* how they can be empirically compared to understand how much improvement is possible.

During the last years, we developed DFLOW, a profiler for the PHARO IDE[1] [7]. PHARO is a window-based environment inspired by Smalltalk supported by an active open-source community. DFLOW records many kinds of interaction events at different levels of abstraction. For example, DFLOW records development meta-events like source code modifications and navigation, UI events like interactions with the windows of the IDE (*e.g.,* minimization, resizing), and interactions with input devices like mouse movements, clicks, and keystroke events. In this work, we focus on a particular subset of interaction events recorded by DFLOW to empirically investigate how efficiently developers perform navigation through software entities using the basic UI components of PHARO. For this reason, our main focus is on *source code navigation* events, for example when the user selects a program entity in the code browser (see Figure 2), the mostly used UI in the PHARO IDE to support code navigation and editing. The dataset supporting our empirical study counts more than 200,000 interaction events coming from 6 developers, and totaling more than 504 hours (*i.e.,* 21 days) of actual development activity. The recorded development sessions last on average around 40 minutes, and contain, on average, 276 development meta-events, 214 of which are navigation events.

To measure the efficiency of developers in navigating source code, we compare their *real* navigation effort with a set of different *ideal* settings. The ratio between real and ideal settings provides preliminary estimates for the navigation efficiency. The ideal scenarios leverage how the developer could, theoretically, make the most from the user interface and the components of the IDE while navigating source code. We assume two different ideal cost models for single navigation events, and two different models to estimate the ideal working set required by developers to implement the task at hand in a given development session.

We discovered that, on average, developers perform 1.5 to 19 times more navigation events than the ideal scenarios. In half of the cases, however, the efficiency is even worse, with redundant navigations being from 3 to 30 times more than the ones needed in ideal settings.

Structure of the Paper: Section II details DFLOW, our supporting tool, describes the conceptual model of DFLOW interaction data events, and describes the dataset. Section III defines the concepts of navigation cost and effort. Section IV defines the navigation efficiency and reports our results. In Section V we summarize the related work and Section VI concludes our work.

[1]See http://pharo.org

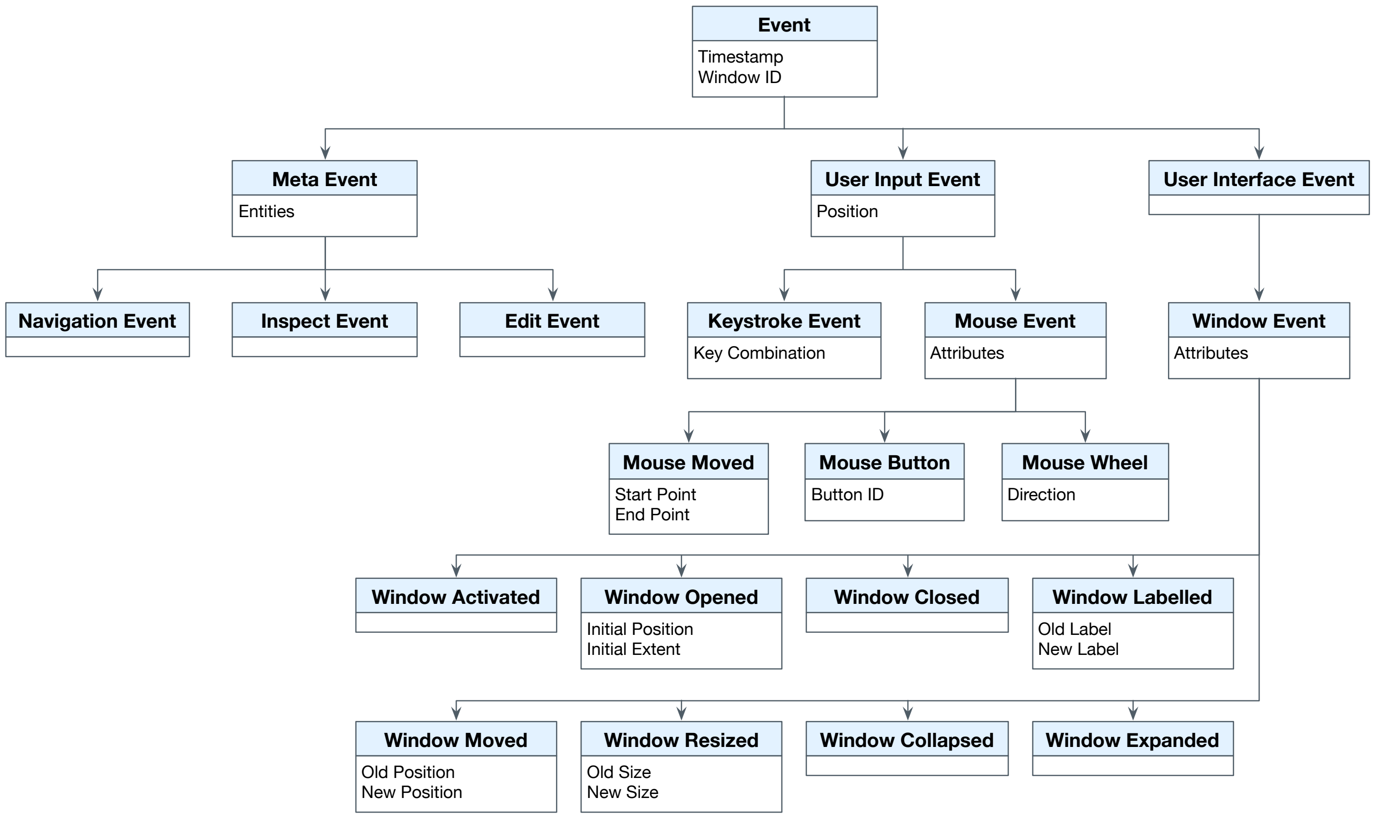


Fig. 1. The Conceptual Model of DFLOW Interaction Data Events

II. DFLOW, INTERACTION DATA, AND THE DATASET

This section introduces *interaction data* (Section II-A) and DFLOW, the supporting tool we developed to record this data (Section II-B). Finally, Section II-C details the dataset.

A. What is Interaction Data?

Figure 1 shows the conceptual model of the interaction data events we consider. We distinguish three main categories of events by their level of abstraction: meta events, user input events, and user interface events.

Meta events represent all the interactions of the developer with program entities, *e.g.,* classes and methods. According to the *impact* of the interaction on the source code, we identify three kinds of meta events: navigation, inspect, and edit events. Navigation events correspond to the events the developer performs while exploring source code entities, *e.g.,* selecting an entity in the code browser or performing a search with the appropriate UI. Inspection events represent actions that a developer performs to understand the execution of a program. Examples include when the developer debugs a piece of code or observes the value of a local variable or field. Finally, edit events represent actual source code modifications, *e.g.,* adding a new class or editing the body of a method. In most of the cases, meta events have one or more program entities associated to them, *i.e.,* when the developer modifies the source code of the method `Foo`, the corresponding DFLOW meta event encodes this information for further analyses.

User input events are the events the developer performs using an input device, *e.g.,* mouse and keyboard. We distinguish two categories of events: mouse and keystroke events. All user input events recorded by DFLOW encode the position of the cursor when the event happens and other attributes. Keystroke events, for example, encode the current *key combination* that might be a single keystroke (*e.g.,* the key `'a'`) or a real key combination (*e.g.,* `Cmd + 'v'`).

Mouse events have different types and ad-hoc attributes. For example, the event that represents a *mouse move* encodes the initial and final position of the cursor before and after the movement. Other types of mouse events are clicks and interactions with the mouse wheel.

User interface events represent the interactions of the developer with the user interface of the IDE. In the case of PHARO, the target IDE for our study, the user interface mostly consists of *windows*. Thus, user interface events are *window events*, such as opening, closing, or resizing a window. Each event has its own attributes: A resize, for example, encodes the size of the window before and after the event itself.

Summing Up. Interaction data events consist of all the events that the developer performs inside the IDE. We believe these events are important to understand and support the development process. However, current IDEs neglect these events preventing further use of the information contained in them [8]. To persist interaction data, we developed DFLOW, which is briefly described in the next section.

B. DFLOW*: The Interaction Profiler*

To record and leverage the interaction data inside the IDE, we developed DFLOW, a non-intrusive interaction profiler for the PHARO IDE [7]. The profiler collects more than 30 types of events, organized as per our conceptual model (see Figure 1).

Once a developer installs DFLOW in her IDE, the profiler starts to collect data and periodically sends it to our server to support further analyses. In addition to retrospective analyses, on top of DFLOW we are also developing approaches to support the development process, *e.g.,* the PLAGUE DOCTOR, a prototype that automatically closes unneeded windows in the IDE [9] to mitigate the *window plague* [5].

In the last years we distributed DFLOW to different developers and collected hundreds of thousands of interaction events.

C. The Dataset

Developers working in academia and industry installed DFLOW and shared their interaction events with us. We collected hundreds of sessions composed of fine-grained interaction data events. Table I summarizes the subset of the dataset used for the current study.

TABLE I
THE DATASET

General				
Number of Sessions				765
Number of Developers				6
Duration				
Avg. Session Duration				39m 36s
Total Development Time				504h 57m 5s
Events (per Session)	Q_1	Q_2	Q_3	Avg.
Navigation	56	122	258	213.95
Edit	4	7	13	10.90
Inspect	2	13	44	50.98
Total	86	174	359	275.84
Involved Entities (per Session)	Q_1	Q_2	Q_3	Avg.
Navigated	10	19	39	28.85
Edited	2	4	6	5.46
Inspected	1	5	10	7.15
Total	14	25	43	34.30

A *development session* is a collection of contiguous interaction data events without interruptions lasting more than 5 minutes, (*i.e.,* representing inactivity), *i.e.,* whenever DFLOW detects and interruption, it immediately interrupts the session.

In this work we analyze a corpus of 765 development sessions totaling more than 500 hours (*i.e.,* 21 days) of actual development activity. The overall DFLOW dataset is larger, but for this study we filtered out short sessions, sessions with few navigation events, and sessions with a small working set, *i.e.,* number of program entities involved. In the table we only report information about meta events (see Figure 1), since our aim is to gather a better understanding of how efficient are developers in navigating source code.

The resulting dataset features sessions coming from 6 developers whose background is mostly academic (*e.g.,* researchers, PhD, and master students), but also includes sessions coming from professional open-source software developers. On average, sessions last for about 40 minutes (*i.e.,* 39'36") and involve 34 program entities (*i.e.,* methods, classes, and packages). In each session, on average, a developer modifies 5 program entities (*i.e.,* 5.46).

On average, a session features 214 navigation, 11 edit, and 50 inspection meta events, stressing the importance of the navigation over the other activities. In addition to average measures, Table I reports values for first, second (*i.e.,* median), and third quartiles. We observed a large variability, for example, in the number of navigation events: Q_1=56, Q_3=258, and median of 122. This may suggest that there are tasks in which developers require more (or less) intensive navigation effort. However, an investigation of the causes of such variability is part of our future work.

III. NAVIGATION COST AND EFFORT

This section briefly introduces the PHARO IDE and defines our model of navigation cost and effort.

A. Intermezzo: Pharo Object Model and Code Browser

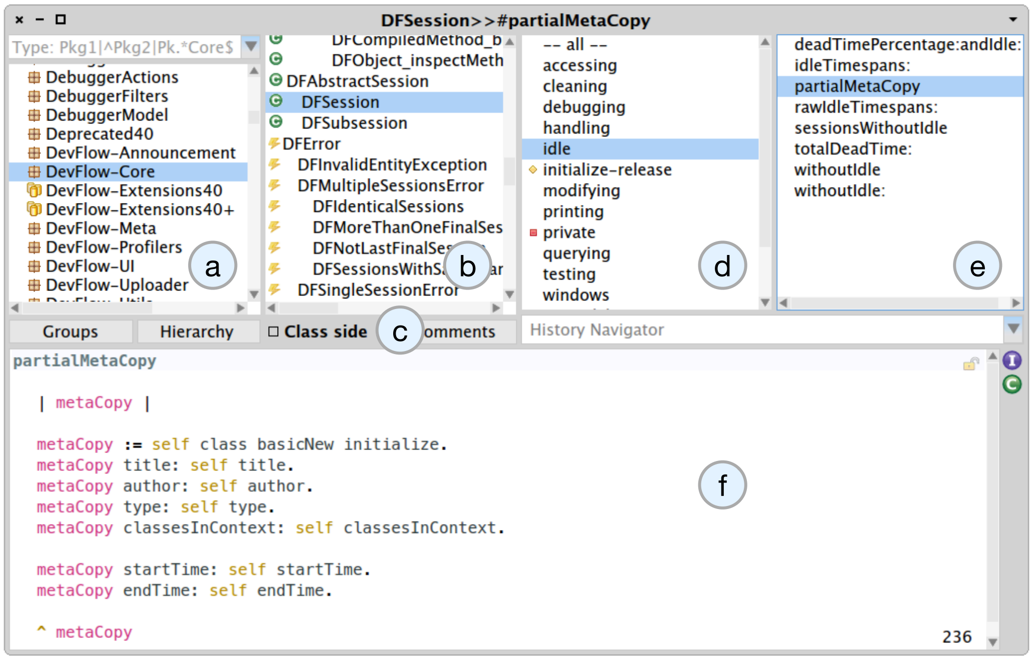
One of the most used UIs in the IDE is the code browser that lets developers navigate, read, and write code (see Figure 2).

The upper part of the browser is mostly used to navigate source code. In PHARO source code is organized as follows:

- **Packages.** The first column of the browser lists all the packages available in the current PHARO image (Fig. 2.a).
- **Classes.** Once the developer selects a package from the list, the second column lists all the classes belonging to that package (Fig. 2.b).
  - **Metaclasses.** In PHARO, each class is an instance of a *metaclass*. Thus, there is a metaclass hierarchy which is parallel to the standard class hierarchy. The developer can click on a button (Fig. 2.c) to reveal the so called *class side* and navigate (or modify) metaclasses and their methods. A common use of a metaclass is to create custom constructors, instead of using the ordinary `new` method to instantiate a class.
- **Protocols.** In PHARO, methods are grouped into protocols that document their intent. Once the developer selects a class (or metaclass), the third column of the browser lists all the protocols available in it (Fig. 2.d).
- **Methods.** Once the developers selects a protocol, the fourth and last column of the browser lists all the methods belonging to the selected protocol (Fig. 2.e).
- **Source Code.** Once the developer selects a method, the bottom part of the code browser displays its source code and lets the developer read and modify the code (Fig. 2.f).

B. Basic Navigation Cost

With the PHARO object model in mind, we define the basic navigation cost needed to reach a given program entity, expressed in terms of number of navigation events, from a newly open browser window.



```
partialMetaCopy

  | metaCopy |

  metaCopy := self class basicNew initialize.
  metaCopy title: self title.
  metaCopy author: self author.
  metaCopy type: self type.
  metaCopy classesInContext: self classesInContext.

  metaCopy startTime: self startTime.
  metaCopy endTime: self endTime.

  ^ metaCopy
```

Fig. 2. Code Browser: The User Interface to Navigate and Modify Source Code in the PHARO IDE

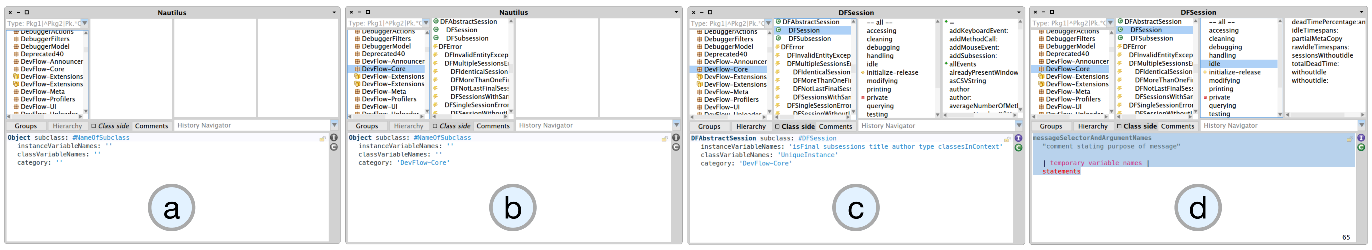


Fig. 3. Selecting Program Entities in the PHARO Code Browser: (a) no selection, (b) package, (c) class, and (d) protocol

The **basic navigation cost** to reach a package is 1.

Packages are directly reachable in the first column of the from the first column of the browser (Fig. 2.a), thus the developer requires only a single navigation, *i.e.,* the actual click on the package of interest. Figure 3.b shows the code browser with a package selected.

The **basic navigation cost** to reach a class is 2. The navigation cost becomes 3 if the developer navigates to the metaclass.

To reach a class, or metaclass, the developer has to select a package. Thus, the base cost is the basic navigation cost of a package, *i.e.,* 1. To this cost we sum the actual cost of the selection of the class. In addition, if the developer wants to reach the class side, an additional navigation is needed, *i.e.,* the click on the *"class side"* button (Fig. 2.c). Figure 3.c shows a code browser after the selection of a class.

The **basic navigation cost** to reach a method protocol is 3. The navigation cost becomes 4 if the developer navigates to a protocol of the metaclass.

To reach a protocol, the developer has to select a class (or metaclass). Thus, the base cost is the basic navigation cost of a class (or metaclass), *i.e.,* 2 or 3 depending if the developer selected a class or a metaclass. To this cost we sum the actual cost of the selection of the protocol. Figure 3.d shows the code browser after the selection of a protocol.

The **basic navigation cost** to reach a method is 4. The navigation cost becomes 5 if the developer navigates to a method of the metaclass.

Finally, to reach a method, the developer has to first select a protocol. Thus, the base cost is the basic navigation cost of a protocol, *i.e.,* 3 or 4 depending if the developer selected a class or a metaclass. To this cost we sum the actual cost of the selection of the method. Figure 2 shows the code browser after the selection of a method.

Advocatus Diaboli: *The basic navigation cost only considers list selections. In practice, a developer might have to* scroll *through the list before actually selecting an entity of interest*. This is true, but to make the comparison adequate, we only call "navigation events" the events that represent actual selections of entities in the code browser lists. Thus we believe our simplification is consistent with our goal, *i.e.,* to understand to what extent an ideal navigation effort differs from what we observe in the recorded navigation histories.

C. Navigation Effort

The *"navigation effort"* is the act of navigating source code elements. We define the *real* navigation effort as the number of navigation events that the developer performs to reach the edited entities. To measure the navigation *efficiency*, however, we define an *ideal* estimate as term of comparison, as follows:

The ***ideal* navigation effort** is the sum of the *navigation costs* needed to reach the *edited program entities* in a development session.

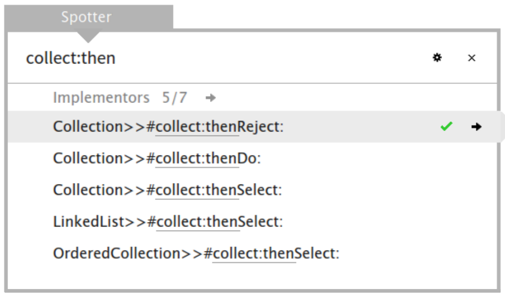
Navigation cost and edited program entities, can be treated in different ways, obtaining more (or less) ideal settings.

Edited Program Entities

- WORKING SET. In the most ideal and optimistic case, a developer knows the set of entities needed for her task, and navigates only the entities that she need to edit.
- WORKING SEQUENCE. In a more realistic case, a developer starts editing a set of entities, but some modifications (*e.g.,* refactorings) may involve re-editing previously modified entities, until a stable point is reached. This corresponds to a *sequence* of edited entities, potentially including duplicates.

Navigation Cost

- UNITARY. In the most ideal and optimistic case, the developer reaches each entity of interest with a single navigation, *i.e.,* directly jumping to it, for example using a spotlight-like interface, as the one depicted in Figure 4. In other words, this model of navigation cost assumes each navigation with a unitary cost.
- MAX COST. In the worst case scenario a developer opens a new code browser for each navigation, thus the cost to navigate each entity is its basic navigation cost.



```
collect:then
Collection>>#collect:thenReject:
Collection>>#collect:thenDo:
Collection>>#collect:thenSelect:
LinkedList>>#collect:thenSelect:
OrderedCollection>>#collect:thenSelect:
```

Fig. 4. PHARO Spotlight-like Interface

IV. NAVIGATION EFFICIENCY: FACTS AND FICTION

We define the navigation efficiency as follows:

$$\text{Navigation Efficiency} = \frac{\text{Ideal Navigation Effort}}{\text{Real Navigation Effort}}$$

where the real navigation effort is the number of navigation events happening during a session and the ideal navigation effort is a combination of edited program entities with their navigation cost, *e.g.,* the working sequence (or set) of edited entities with their unitary (or max) navigation cost.

A. The Facts

We compute the navigation efficiency in different scenarios combining the different interpretations of edited program entities and navigation cost. Please note that in Table II and Table III the headers Q1, Q2, and Q3 represent the Quartiles.

Working Set: Table II presents the navigation efficiency considering only the the set of edited entities (*i.e.,* working set), for the three different variants of navigation cost.

TABLE II
NAVIGATION EFFICIENCY WITH WORKING SET

	Q1	*Q2*	*Q3*	*Avg.*
Unitary Cost	0.018	0.032	0.060	0.051
Max Cost	0.073	0.129	0.238	0.206

UNITARY COST. On average, if we compare the real navigation effort with the most ideal configuration, only 5.1% of the navigation events performed by developers are required. In other words, a developer is performing almost 19 times more events than the ones needed in practice. The median (Q2) reveals even worse news: In half of the sessions the navigation efficiency drops to 3.2%, *i.e.,* developer are performing more than 30 times more events than needed.

MAX COST. On the other side, if we use the max navigation cost as a yardstick, the average navigation efficiency is 20.6%, meaning that developers are carrying on 4 times more navigations than needed.

Working Sequence: A more realistic scenario considers the sequence of edited entities as they appear in the interaction histories. Table III presents the navigation efficiency in this settings, for the three different variants of navigation cost.

TABLE III
NAVIGATION EFFICIENCY WITH WORKING SEQUENCE

	Q1	*Q2*	*Q3*	*Avg.*
Unitary Cost	0.036	0.060	0.112	0.096
Max Cost	0.137	0.242	0.450	0.387

UNITARY COST. If we consider unitary navigation cost, the efficiency of developers is 9.6%, which drops to 6% if we consider the median value.

MAX COST. On the other extreme, with maximum cost, the efficiency of developers raises to 38.7% on average, and 24.2% if we look at the median.

B. The Fiction — Summing Up

Our results provide empirical evidence that developers navigate source code in an inefficient way. On average, in the more realistic case (working sequence), developers perform 1.5 to 19 times more navigation events than the optimal settings. The situation is even worse when looking at median values where the number of wasted navigations reaches 3 to 30 times more than the ones required by the current task.

We know there are several components that impact navigation and make the ideal settings unfeasible. For example, Röthlisberger *et al.*, claimed that the navigation is is hampered by the fact that conceptually related entities are distributed in a very large software space and that relationships between them are often hidden [5], [10]. Furthermore, the construction of a mental model of a software system, requires that the developer visits more program entities than the ones that she effectively needs to modify (*e.g.,* [1], [2]).

We believe that these results suggest that the gap between the ideal and the real scenarios can be bridged by better supporting the navigation inside the IDE. The next section addresses how the related work started to mitigate this issue.

V. Related Work

We believe our results empirically motivate the need of reducing the gap between the ideal and the effective navigation effort. In this section we discuss different approaches to support the navigation through software in different ways.

Storey *et al.* developed SHriMP, a flexible and customizable environment to explore software systems [11]. SHriMP offers a catalogue of graph-based architectural visualizations that integrate data from different sources to provide a more structural exploration of code. Janzen and De Volder created a tool to reduce the confusion while navigating code [12]. Their approach consists in providing an explicit representation of the exploration process by means of *exploration paths*. Mylyn (formerly Mylar), instead, assigns a degree-of-interest (DOI) value to each source code artifact based on the interaction data involving that entity, *e.g.,* selections or modifications [8]. Mylyn uses an episodic-memory inspired interface that allows developers to see only the information they require for the current task, potentially reducing the effort needed to reach an entity of interest. Singer *et al.* developed NavTracks, a tool that keeps track of the navigation history of software developers to supports browsing through software [6]. In a nutshell, every time a developer selects an artifacts, NavTracks shows a list of artifacts possibly related to it. SmartGroups is a tool that helps developers to focus on relevant code entities for the current task [10]. SmartGroups keeps track of navigation and edit activities together with evolutionary and runtime information to provide developers with a more structured view of the entities needed to complete the current task. Augustine *et al.* investigated how to comprehend and maintain source code more efficiently by fostering structural code navigation [13]. Their Prodet tool provides a navigable visualization of the relevant parts of the call graph based on the current context.

VI. Conclusions

Recording interaction data events is fundamental to derive empirical evidence of phenomena that affect development, like the difficulty of navigating complex software entities. However, as data is raw, it must be interpreted according to some reasonable model to be valuable and leveraged.

We presented a series of models, from completely ideal to more realistic ones, interpreting interaction data to understand how efficient is the navigation of software entities supporting programming tasks in the Pharo IDE. According to our models, data supports the need of better mechanisms to navigate source code, as the efficiency of programmers is very far from an ideal setting.

Future Work. We plan to leverage interaction data events to provide developers with tools (*e.g.,* recommender systems, always-on interactive visualizations) to better support the navigation of source code. In addition, we want to collect further evidence by devising other models to interpret interaction data.

Acknowledgements. We gratefully acknowledge the support of the Swiss National Science Foundation (SNSF) for the project "HI-SEA" (No. 146734).

References

[1] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining Mental Models: A Study of Developer Work Habits," in *Proceedings of ICSE 2006 (28^{th} Int'l Conference on Software Engineering)*, 2006, pp. 492–501.

[2] M. A. D. Storey, F. D. Fracchia, and H. A. Müller, "Cognitive Design Elements to Support the Construction of a Mental Model during Software Exploration," in *IWPC-97*, 2007, pp. 1–17.

[3] R. DeLine, M. Czerwinski, and G. G. Robertson, "Easing Program Comprehension by Sharing Navigation Data," in *Proceedings of VL/HCC 2005 (Symposium on Visual Languages and Human-Centric Computing)*, 2005, pp. 241–248.

[4] A. T. T. Ying and M. P. Robillard, "The Influence of the Task on Programmer Behaviour," in *Proceedings of ICPC 2011 (19^{th} Int'l Conference on Program Comprehension)*, 2011, pp. 31–40.

[5] D. Röthlisberger, O. Nierstrasz, and S. Ducasse, "Autumn leaves: Curing the window plague in IDEs," *Proceedings - Working Conference on Reverse Engineering, WCRE*, pp. 237–246, 2009.

[6] J. Singer, R. Elves, and M.-A. Storey, "Navtracks: Supporting navigation in software maintenance," in *Proceedings of IWPC 2005 (13^{th} Int'l Workshop on Program Comprehension)*, 2005, pp. 325–334.

[7] R. Minelli, A. Mocci, and M. Lanza, "I Know What You Did Last Summer – An Investigation of How Developers Spend Their Time," in *Proceedings of ICPC 2015 (23rd Int'l Conference on Program Comprehension)*, 2015, pp. 25–35.

[8] M. Kersten and G. C. Murphy, "Mylar: a degree-of-interest model for IDEs," in *Proceedings of AOSD 2005 (4^{th} Int'l Conference on Aspect-Oriented Software Development)*, 2005, pp. 159–168.

[9] R. Minelli, A. Mocci, and M. Lanza, "The Plague Doctor: A Promising Cure for the Window Plague," in *Proceedings of ICPC 2015 (23^{rd} Int'l Conference on Program Comprehension)*, 2015, pp. 182–185.

[10] D. Röthlisberger, O. Nierstrasz, and S. Ducasse, "SmartGroups: Focusing on task-relevant source artifacts in IDEs," in *Proceedings of ICPC 2011 (19^{th} Int'l Conference on Program Comprehension)*, 2011, pp. 61–70.

[11] M.-a. Storey, C. Best, and J. Michand, "Shrimp views: An interactive environment for exploring java programs," in *Proceedings of IWPC 2001 (9th Int'l Workshop on Program Comprehension)*, 2001, pp. 1–4.

[12] D. Janzen and K. De Volder, "Navigating and querying code without getting lost," *Proceedings of AOSD 2003 (2^{nd} Int'l conference on Aspect-oriented software development)*, pp. 178–187, 2003.

[13] V. Augustine, P. Francis, X. Qu, D. Shepherd, W. Snipes, C. Br, and T. Fritz, "A Field Study on Fostering Structural Navigation with Prodet," in *Proceedings of ICSE 2015 (37^{th} Int'l Conference on Software Engineering)*, 2015, pp. 229–238.