

Taming the IDE with Fine-grained Interaction Data

Roberto Minelli[†], Andrea Mocci[†], Romain Robbes[‡], Michele Lanza[†]

[†]REVEAL @ Faculty of Informatics — Università della Svizzera Italiana (USI), Switzerland

[‡]PLEIAD @ Departamento de Ciencias de la Computación (DCC) — University of Chile, Chile

Abstract—Integrated Development Environments (IDEs) lack effective support to browse complex relationships between source code elements. As a result, developers are often forced to exploit multiple user interface components at the same time, bringing the IDE into a complex, “chaotic” state. Keeping track of these relationships demands increased source code navigation and cognitive load, leading to productivity deficits documented in observational studies. Beyond small-scale studies, the amount and nature of the chaos experienced by developers in the wild is unclear, and more importantly it is unclear how to tame it.

Based on a dataset of fine-grained interaction data, we propose several metrics to characterize and quantify the “level of chaos” of an IDE. Our results suggest that developers spend, on average, more than 30% of their time in a chaotic environment, and that this may affect their productivity. To support developers, we devise and evaluate simple strategies that automatically alter the UI of the IDE. We find that even simple strategies may considerably reduce the level of chaos both in terms of effective space occupancy and time spent in a chaotic environment.

I. INTRODUCTION

Integrated Development Environments (IDEs) are the vehicle used by developers to construct, debug, and evolve source code. IDEs leverage different user interface (UI) paradigms to support manipulation of source code artifacts: window-based, like in the Pharo IDE¹, or tab-based, like in the Eclipse IDE². Unfortunately, neither of these two main paradigms effectively supports the navigation of source code artifacts [1], mainly because no existing UI paradigm is able to effectively and efficiently tackle the complex relationships between source code entities. To move away from some known limitations of traditional IDEs, researchers developed different approaches to better support navigation through software (e.g., [1], [2], [3]) and alternative UI metaphors (e.g., [4], [5], [6]). In practice, most IDEs still adopt classical window-based or tab-based UIs.

The ineffectiveness of navigation support in IDEs has been observed and studied. Ko *et al.* found that up to 35% of development time is spent navigating code [7]. This phenomenon can take the form of the “window plague” [3], manifesting itself when developers are forced to open many UI components at the same time to reveal the relationships between code entities for the task at hand, and bringing the IDE into a “chaotic” state³. Since there is limited evidence that this chaos affects the developer’s productivity (e.g., [9]), the nature and amount of chaos experienced by developers is hard to characterize and quantify. This makes developing mitigating countermeasures challenging.

We start by looking for evidence of chaos by analyzing two different datasets. The first one comes from Pharo, a window-based IDE, and includes around 770 hours of development data coming from 17 developers. We observe that these developers spend on average 30% of their time in a chaotic environment. The key characteristic that makes our approach possible is the large number of fine-grained interaction data that we were able to record over an extended period. To get a more encompassing vision, we also analyzed the Mylyn dataset [10], which features several thousands of individual tasks coming from 179 developers. Due to the limited amount of information available in this dataset, a similarly detailed analysis was not possible. Instead, we quantified the number of entities (classes and methods) interacted with in Mylyn tasks. This analysis, corroborating the existing literature on the topic, provides sufficient evidence that chaos is also prominent in tab-based IDEs. Focusing on the finer-grained dataset, we were able to comprehensively investigate the chaos phenomenon, characterizing it in terms of the window space required to support development tasks, and the overlapping of these windows.

Intuitively, having a chaotic environment hinders development: The developer has to cope with the chaos of the IDE itself, while she needs to piece together information scattered in several entities displayed in different UI components. This chaos is visible in activities such as searching for windows, closing unneeded tabs, or moving windows around to reveal what is below. We found a statistically significant, positive correlation between the level of chaos, and the proportion of time that developers spend altering the UI of the IDE. We found another positive and significant correlation between chaos and the proportion of time spent performing program comprehension tasks. This corroborates the findings of the studies of Ko *et al.* [7] and Bragdon *et al.* [9].

These findings call for novel approaches whose aim is to reduce the chaos of the IDE and make more time and space available for the real essence of software development. Fine-grained interaction data is useful there too: To help developers coping with the chaos, we devised and evaluated simple strategies that automatically reshape the UI of the IDE. Our findings reveal that even simple strategies may considerably reduce space occupancy and time spent in a chaotic environment.

We can summarize the contributions of our paper as follows:

- A set of metrics to characterize and quantify the “level of chaos” of a programming session in an IDE, based on fine-grained interaction data;

¹See <http://pharo.org>

²See <https://eclipse.org>

³Not to be confused with “deterministic chaos” [8].

- A detailed analysis of two datasets coming from two different IDEs. We primarily focus on the Pharo dataset, due to the rich information at our disposal. To get further evidence, we investigate the size of the task contexts in a large number of development tasks in the Mylyn dataset.
- An evaluation of the effectiveness of elision and layout strategies to reduce the level of chaos in the IDE.

Structure of the Paper: Section II analyzes the literature and provides initial empirical evidence, through the analysis of Mylyn data, of the presence of chaos in the IDE UI. Section III details our primary dataset and explains how we modeled, characterized and measured the “chaos inside the IDE”. In Section IV we present our strategies to reduce the chaos present in the IDE and discuss their impact. Finally, Section V discusses the threats to validity of our work; we conclude and discuss further work in Section VI.

II. MOTIVATION

The complexity of building and maintaining working sets for typical development tasks can both explain the chaotic configuration of an IDE, and be impacted by it.

According to Wexelblat [11], the information path obtained from navigation in an information space reveals the user’s mental model of the system. In software engineering, developers spend a considerable portion of their time building and maintaining the working set of code fragments relevant to a task. This is challenging when the relevant code fragments are dispersed in several locations in the system. An observational study by Ko *et al.* reported that developers spend 35% of their time navigating the source code in search for information [7], and that 27% of the navigation operations are performed on already visited locations, indicating the necessity to periodically revisit these locations to recall information no longer visible on screen.

This study is not alone: The recent context model study by Fritz *et al.* [12], based on detailed observations of 12 developers, each solving three tasks, found that the average context model necessary to solve a task contained 4 classes. Further evidence is present in the study by Sillito *et al.* [13]: Developers ask a variety of questions during maintenance tasks; answering some of the questions involves inspecting several entities, increasing working set size.

A. Improving Management of Working Sets

Several approaches have been proposed to improve the management of working sets. Robillard and Murphy proposed to represent scattered concerns in source code as concern graphs [14]. Mylyn itself is such a tool, which monitors interaction data to automatically build a degree-of-interest model (DOI), altering the views of the Eclipse IDE by filtering out entities with a low DOI value [10]. Its effectiveness has been empirically demonstrated [15]. The Degree-of-Knowledge (DOK) model by Fritz *et al.* is an extension of the DOI, that also includes authorship [16]. Other tools monitoring interactions to help software exploration have been proposed, such as Navtracks [1], and Teamtracks [17].

The problem with Tab-based UIs. A key issue not addressed by these works is the fact that most IDEs do not properly support the work of maintaining one’s mental model by adopting a file-based representation of source code, while most working sets span several files. Moreover, the particular UI paradigm typically used to manipulate source code hinders the maintenance of complex working sets.

Eclipse is a good representative of widely used IDEs (such as Visual Studio, Netbeans, IntelliJ Idea) that adopt the tab-based metaphor. Each file is shown as an editor in the IDE, with navigation tools (Package explorers, search tools, etc) shown as views around the central editor. By default, the screen estate allows for at most one file to be visible at the same time, while other open files are shown as “hidden” tabs. This is the case of the overwhelming majority of the Java developers broadcasting their coding sessions on the “livecoding” website⁴: Out of several dozens of videos linked on the site, only a handful of developers stray away from the IDE’s default settings and use two code tabs at the same time, even if the vast majority of published coding videos show IDEs with several tabs open at the same time.

The study by Ko *et al.* [7] pointed out a considerable number of re-navigation to entities recently browsed (27%). It highlights patterns of *back-and-forth navigation* between two files to compare similar pieces of code, which is necessary if only one tab is visible at a time. In a series of controlled experiments investigating the influence of type systems [18], and of API documentation [19], [20], researchers observed that the treatments with higher code completion times also had larger working sets, and a larger number of tab switches in a tab-based IDE. All of these findings highlight the issues related to dealing with multiple tabs in an IDE when several scattered fragments need to be accessed at the same time.

B. Evidence from Mylyn data

To explore in a more systematic way the phenomenon of how tab-based IDEs support the management of complex working sets, we measured the size of the task contexts⁵ contained in the Mylyn dataset available in the Eclipse Bugzilla repository⁶. We downloaded 6,182 bug reports that had a Mylyn task context as attachment. For each task context, we counted the number of distinct Java files and methods that were interacted with. We applied filtering techniques to bypass some of the deficiencies of the data [21], namely removing massive selection events that could lead to an overestimation of the number of entities interacted with (*e.g.*, selecting an entire group of classes from the navigation panel, without actually opening them). In essence, we filter events that originate less than 100ms after the previous event.

We are left with the number of Java files and the number of methods that were interacted with during a task (we exclude other types of files, such as .class or XML files). The median task context has interactions with 3 Java files, that is, at least

⁴See <https://www.livecoding.tv/videos/java/>

⁵A set of artifacts that Mylyn considers relevant for the task-at-hand.

⁶See <https://bugs.eclipse.org>

half of the tasks involve interactions with at least three Java files. The upper quartile is 8, meaning that for at least 25% of the tasks, the developer needed to consult 8 or more java source code files to finish the task. Clearly, a large number of tasks demand non-trivial interactions with a large number of source code entities. Outliers are even higher; if we focus on methods, the median number of methods interacted with is 5, while the upper quartile is 21. This indicates that for at least 25% of the tasks, the developer had to piece together information from a large number of methods.

This is a likely sign that the typical size of working sets, together with the way that source code is represented by the tab-based UI paradigm, may generate chaos in the IDE, forcing developers to spend considerable time in interacting with the UI components to manipulate their working sets, for instance to revisit entities as documented by Ko [7].

Obtaining further evidence is hard, because of the Mylyn data itself, which presents several limitations for this particular investigation: It does not contain information on the visibility of elements on screen, so it is impossible to know how many tabs were open at distinct points during a task, or if developers had several tabs visible at the same time. There is no information to estimate the size of the screen, or the size of tabs. The data is aggregated, and it is not always possible to know the exact sequence of events (a sequence of events concerning an entity may be encoded as a time period where several events occurred, reducing precision, as documented by Ying and Robillard [22]). Finally, the Mylyn data is based on a files-and-tabs metaphor, which is in itself limiting in terms of possible optimizations.

C. Beyond Tab-based IDEs

Recent efforts have investigated better program representations and UI paradigms than the file-and-tab-based metaphor of most common IDEs. We can trace back this inspiration to the Lisp and Smalltalk IDEs of the 80's, whose most recent representative is Pharo. Efforts include Code Canvas [5] and Code Bubbles [4]. Code Canvas has seen parts of its functionality released in Visual Studio as Debugger Canvas, which also integrates parts of Code Bubbles' functionality [23]. These tools aim to reduce the amount of code navigation, by maximizing the number of entities visible at the same time.

The evaluation of CodeBubbles is particularly instructive. The authors showed that at a similar screen resolution, Code Bubbles was able to show more methods at the same time than the classic Eclipse view [4]. Furthermore, a controlled experiment showed that Code Bubbles users were both more successful and faster in completing maintenance tasks than Eclipse users [9]. Parts of this performance increase is attributable to a reduction of repeated navigations, such as the ones observed by Ko, according to the videos recorded during the controlled experiment (75.9% of all Eclipse navigation operations, compared to 37.6% for Code Bubbles), as more entities were visible on screen.

D. Strengthening the existing evidence

Approaches such as CodeBubbles, and the insights that one can obtain from its evaluation, motivate the need to evaluate the impact of IDE UIs in alternative metaphors to the classic tab-based approach. Moreover, window-based IDEs also suffer from UI-related phenomena like the *window plague* [3], [2], and our work lies in the same area of research. We leverage our experience in recording and mining interaction data in the IDE [24] to model and characterize the impact of chaos in window-based IDEs, evaluate possible techniques that can ameliorate the developer experience, and ultimately improve the support that UI components give in constructing and maintaining the working set by managing in a more efficient way the screen real estate. While the empirical evidence brought forth by CodeBubbles [9] shows that increasing the number of code fragment visible at the same time has a positive impact on productivity, it is lacking in several aspects. Beyond the simple focus on the more general setting of window-based IDEs like Pharo, our study complements it in several ways.

Variety of Tasks. Code Bubble's productivity benefits are shown in the context of a controlled experiment, with strong internal, but limited external validity [25]. In fact, the study was conducted on two well-defined development tasks only, while both our datasets do not impose any constraint on the tasks at hand. This increases the external validity of the findings, at the price of a lower internal validity as the conditions can not be controlled as in an experimental setting.

Duration of recorded data. The conclusions in the Code Bubbles experiment are drawn from around 30 hours of development. On the other hand, our DFLOW dataset contains more than an order of magnitude of data (around 750 hours).

Entities Displayed. Code Bubbles was also evaluated on the number of methods shown on screen [4]. This was however done on a limited number of methods.

Impact of UI components. We characterize the impact of chaos not only on navigations, but in general on the time spent on UI fiddling (*e.g.*, resizing windows) and the time spent on program understanding.

Recorded interaction data. Last but not least, our approach records enough interaction data to let us simulate the impact of various strategies on the chaos in the IDE, without needing to implement a prototype in the early stages.

These characteristics make our evaluation highly complementary to previous evaluations.

III. LOST IN THE IDE

To collect interaction data inside the Pharo IDE we built DFLOW [24], a non-intrusive profiler. When a developer enables DFLOW, it starts to silently capture the interactions of the developer with the IDE. This includes the ones performed on the IDE UI, *e.g.*, move, resize, or open/close a window. For each event, DFLOW records a timestamp down to millisecond precision. In addition, DFLOW records *meta events*, representing actions triggered by the developer inside the IDE, such as browsing the source code of a method or adding a new method to a class.

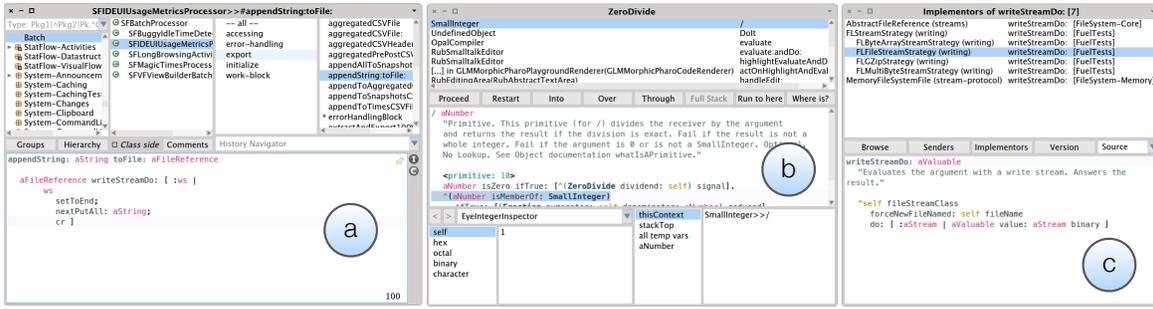


Fig. 1. Windows Displaying Source Code: (a) Code Browsers, (b) Debuggers, and (c) Message Lists

What is a Development Session? We call “development session” each sequence of IDE interactions captured with DFLOW which satisfy a series of constraints. In particular:

- All events happen in the same development environment/context (*i.e.*, an *image* in the Smalltalk jargon);
- There are no adjacent pairs of events such that there are more than 5 minutes of inactivity between them;
- When the user closes the IDE, the session terminates.

With this definition, each development session is a self-contained and focused development period without long interruptions. This removes the potential problem of considering major interruptions (*e.g.*, Skype calls, coffee breaks) as part as the development flow.

A. DFlow Dataset

Table I summarizes our dataset. It counts 771 hours of development time coming from 17 open-source and academic developers working on and around the Pharo project.

TABLE I
DFLOW DATASET

Metric	Value	
Number of Sessions	1,631	
Number of Developers	17	
Development Time	771h 10m 21s	
Avg. Session Duration	28m 22s	
Metric	Total	Avg. per Session
Number of Windows	40,140	24.61
Number of Browsers	6,833	4.19
Number of Debuggers	2,844	1.74
Number of Message Lists	3,870	2.37
UI Time	102h 15m 30s	3m 55s
Understanding Time	594h 54m 57s	22m 49s

We collected interactions with more than 40,000 windows, that we further refined according to their type. We only consider interactions with windows whose aim is to display and let the user interact with source code:

- **Code Browsers** are the core windows that let users navigate, read, and write source code (see Figure 1.a).
- **Debuggers** are the windows dedicated to debugging activities. They let users navigate the call stack, watch the state of variables, and read/edit in place the source code of a method (see Figure 1.b).

- **Message Lists** are all the UIs that display a list of methods and, upon selection, the source code of the method itself. Example includes UIs to browse implementors of a method or methods that invoke another method (see Figure 1.c)⁷.

Windows. We collected a total of 6,833 code browsers, 2,844 debuggers, and 3,870 message lists. Each session, on average, lasts for ca. half an hour and counts interactions with 24 generic windows. Considering only windows containing source code, on average each session features 4 browsers, 2 message lists, and 2 debuggers. However, these aggregate metrics are highly variable, with a considerable number of outliers. For example, considering sessions with windows containing code, the number of outliers (containing more than 18 of such windows) is 175, roughly corresponding to a tenth of the recorded sessions.

Activity Durations. In previous work we used interaction data to precisely measure the time spent in several programming tasks, like editing, navigating and searching for code artifacts, interacting with the UI of the IDE, and performing corollary activities, such as object inspection at runtime [24]. Two of these components are useful for our current study, when it comes to understand how the “level of chaos” correlates with the behavior of the user.

The first is the *UI Time*, devoted to fiddling with the UI of the IDE, *i.e.*, moving and resizing windows. Our dataset features more than 102 hours of UI Time (on average, ca. 3m 55s per session, ca. 14% of the total time).

The second is an estimate of the time devoted to program understanding. In our model [24], we consider as understanding the sum of three components: i) basic understanding time; ii) time spent inspecting objects at runtime; and iii) time spent doing mouse drifting, *i.e.*, the time the user “drifts” with the mouse without clicking, for example to support code reading. Essentially, the basic understanding time is composed of all the time intervals without any recorded event in the interaction data that are greater than a given reaction time⁸ (that in our model is equal to 1 second).

⁷Opening the call hierarchy of a method, in Eclipse.

⁸This approximates the *Psychological Refractory Period* [26] that varies among humans, depending on the task at hand, between 0.15 and 1.5 seconds. For further details on how we model understanding, see Minelli *et al.* [24].

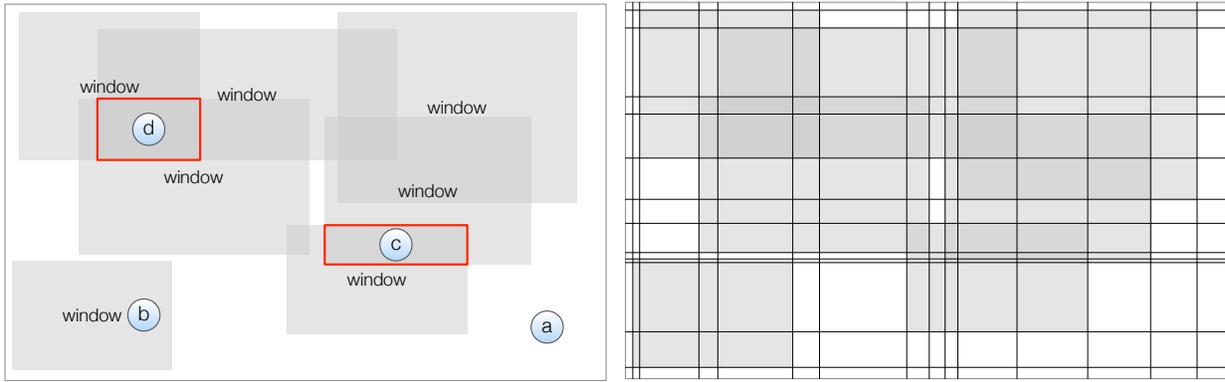


Fig. 2. A Visualization of a Snapshot of a Session (left) and the Screen Regions Used to Measure the Chaos (right)

TABLE II
SPACE OCCUPANCY METRICS

Occupied Space	The sum of the areas of all the screen regions occupied by 1+ windows
Free Space	The sum of the areas of all the screen regions not occupied by any window
Focus Space	The area of the screen region occupied by the active window
Needed Space	The total sum of the areas of all the windows, i.e., the space needed to display all windows
Overlapping Space	The sum of the areas of all the screen regions occupied by 2+ windows
Overlapping Depth	The number of overlapping windows in a given screen region
Weighted Overlapping Space	The sum of the areas of all the screen regions weighted by their overlapping depth

The reaction time models the time that elapses between the end of a physical action sequence, *i.e.*, typing or moving the mouse, and the beginning of concrete mental processes like reflecting, thinking, planning, etc. which represent the basic moments of program understanding. In total, we estimate more than 594 hours of understanding time, on average more than 22 minutes per session (*i.e.*, around 80% of the session duration).

B. Modeling Chaos

To characterize and quantify the “level of chaos” of a programming session in the IDE, we introduce a set of UI metrics related to the usage of the screen and we observe how they evolve throughout the sessions.

1) *Quantifying Chaos*: Our goal is to measure how developers exploit the screen space using the metrics of Table II.

Measuring chaos of a single layout. Consider a given moment during a development session, with a fixed layout of windows in the screen. All the metrics listed in Table II rely on the concept of *screen region*, a part of the screen obtained by creating a grid on the screen using all the coordinates (*i.e.*, position) of the visible IDE windows in the screen in a given *snapshot* of a development session, see Figure 2 (right).

Figure 2 shows a visualization of a snapshot of a session (left) and its decomposition in screen regions (right). In the view, each window is depicted with a translucent gray rectangle (*i.e.*, with size and position proportional to the real

window in the IDE). The white rectangle that contains all the windows represents the IDE space, *i.e.*, the screen.

In Figure 2 the darker the screen region, the more the overlapping between windows. The figure highlights different areas of the screen: (a) one with no windows (*i.e.*, free space), (b) one with a single window (*i.e.*, no overlapping), (c) one with low overlapping (*i.e.*, only 2 windows overlap), and (d) one with high overlapping (*i.e.*, 3+ windows overlap).

We quantify overlapping in three different ways: *overlapping space*, *overlapping depth*, and *weighted overlapping space*. The first measures the linear overlapping space expressed as the sum of the areas of all the screen regions occupied by more than two windows. The depth indicates, for each screen region, how many windows overlap. The weighted overlapping is a combination of the previous two measures that assigns more weight to regions with higher overlapping depth.

Aggregating layout changes. To observe the evolution of the chaos, we divide each session into *snapshots*. Figure 3 schematizes a development session, *i.e.*, a timeline of events. Among these events, there are events that induce a change of the layout of the IDE (depicted in red) and events that do not change the layout of the IDE (depicted in gray). The former are, for example, when the user resizes a window, opens a new window, or closes an existing window.

We call *snapshot* each sequence of events that begins with a layout-changing event and continues until the last event before the next layout-changing event (with the exception of the first and last snapshot that are delimited by the session start and session end respectively). In short, a snapshot is a period in which the layout of the IDE is fixed, thus the values of all the metrics listed in Table II are constant for the entire duration of the snapshot. In the example depicted in Figure 3, there are four snapshots, from S#1 to S#4.

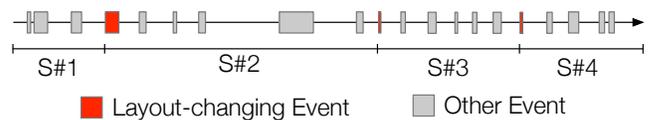


Fig. 3. Session Snapshots Explained

To aggregate and define the level of chaos of a session, we average the values of metrics for each snapshot. Table III summarizes the distribution of space occupancy metrics across all sessions. For each metric we report the average and quartile values across all the sessions. Values (except for the overlapping depth) are expressed in percentage with respect to the available screen space (*i.e.*, resolution).

TABLE III
DISTRIBUTION OF SPACE OCCUPANCY METRICS ACROSS ALL SESSIONS

Name	Avg.	Quartiles		
		Q1	Q2	Q3
Occupied	48.22%	36.16%	43.78%	61.37%
Free	51.78%	38.63%	56.22%	63.84%
Focus	32.66%	20.06%	35.22%	40.12%
Needed	96.83%	39.35%	73.47%	119.53%
Overlapping	20.95%	0.00%	17.08%	35.97%
Weighted Overlapping	69.05%	0.00%	39.61%	95.59%
Overlapping Depth	2.76	1.00	2.00	3.40

On average, the needed space is around 100%, which seems to imply that developers do not experience chaos. However, the average occupied space is around 50% of the screen, with 21% overlapping, indicating at least a need for a better management of the screen real estate. In 25% of the sessions (*i.e.*, Q1) windows do not overlap at all, while in the last quartile the average needed space is always above 120%.

To better analyze the impact of chaos in the recorded development sessions and shed light into how chaos impacts development activities, we group the metric values into categories and we take in consideration the time dimension.

2) *Categories of Chaos*: To characterize the level of chaos in a session we use the space occupancy metrics. The most important indicators are the needed space and the overlapping space. However, these values are *continuous*, and it is not clear at which thresholds of values the chaos is acceptable or not.

Having a high needed space (*e.g.*, $\geq 100\%$ of the available space) means that there is no way a developer can re-arrange the open windows to fit them all on screen. This forces her to have overlapping windows, which is suboptimal for an efficient working environment. In fact, this intuitive correlation holds across our entire dataset: Needed space and weighted overlapping have a strong positive correlation with the *Pearson Correlation Coefficient* $PCC=0.99$ (statistically significant at 95% confidence interval with p -value $2.2E-16$). Thus, we define chaos levels using only one indicator, the needed space, because it is a more intuitive metric than a weighted sum.

We identify two macro-levels of chaos: low- and high-chaos. The threshold that distinguishes between the two levels is 100%, *i.e.*, when the screen resolution is – ideally – enough to accommodate all the open windows, we say that the chaos is low. We use the term “ideally” because the needed space does not take into account overlapping, *i.e.*, having less than 100% of needed space does not imply that windows are uniformly distributed in the screen without overlapping. Conversely, if the screen resolution is insufficient (needed space $>100\%$) we say that the chaos is high.

We refined the two macro-levels into four distinct levels of chaos: *Comfy*, *Ok*, *Mess*, and *Hell*, detailed in Table IV.

TABLE IV
CHAOS-LEVELS: COMFY, OK, MESS, AND HELL

Comfy	$<75\%$ of the screen is required to layout windows. The user can still manipulate and rearrange windows in a comfortable manner, supporting the task at hand, which requires a likely small/reduced working set.
Ok	$>75\%$ and $\leq 100\%$ of screen is required to layout all windows
Mess	$>100\%$ and $\leq 200\%$ of screen is required to layout all windows
Hell	$>200\%$ of screen is required to layout all the windows, <i>i.e.</i> , a developer would need more than two screens (needed space $>200\%$) to arrange all the currently opened windows.

Justifying the Thresholds. To define the levels of chaos we chose three thresholds for the value of the needed space metric: 75%, 100%, and 200%. Defining these thresholds in a systematic and objective way it is far from trivial, if not impossible. The perceived level of chaos is subjective and depends on several factors, *e.g.*, resolution.

Time spent in Chaos Configurations. Table V summarizes the average values (per session) of the time spent in each of the four chaos-levels. For each level we report the percentage of the time spent (with respect to the total duration of each session) and the absolute value. All values are averages across all the sessions in our dataset.

TABLE V
AVERAGE TIME SPENT PER CHAOS-LEVEL

	%	Duration
Comfy	51.04%	10m 50s 126ms
Ok	16.98%	5m 10s 633ms
Mess	21.11%	7m 15s 16ms
Hell	10.88%	5m 26s 922ms

The results above show that for around 32% of the time developers work in a high-chaos setting, *i.e.*, for a session of around 30 minutes of work, more than 12 minutes are spent working with windows occupying more than 100% of the screen. Moreover, 5 out of 30 minutes are spent in a more critical setting, the hell configuration, where the needed space is over 200% the resolution. The total time spent in high-chaos amounts to ca. 331 hours, *i.e.*, 42.92% of development time.

How does chaos correlate with the time spent with UI fiddling and program understanding? For each session, we correlate the percentage of time spent in each configuration with the UI and understanding time, using the test for PCC. Table VI shows the results of these tests.

TABLE VI
CHAOS, UI, AND UNDERSTANDING TIME

	UI		Understanding	
	PCC	p -value	PCC	p -value
Comfy	-0.34	2.20E-16	-0.27	2.20E-16
Ok	-0.04	1.03E-01	0.05	3.36E-02
Mess	0.16	4.42E-10	0.11	1.940E-05
Hell	0.42	2.20E-16	0.26	2.20E-16



Fig. 4. Elision and Layout Strategies in a Nutshell

The moderate correlations are expected when considering that understanding time, for example, is influenced by a multitude of factors, including not only the size of working sets but the quality of code or the difficulty of the task at hand.

On high-chaos levels, developers likely spend more time fiddling with the UI and on program understanding. This is consistent with a likely presence of more complex working sets, spread on multiple windows, that require more attention and time to be managed. The correlation between the time spent on the hell configuration and the UI time is particularly strong (0.42 PCC, p-value 2.20E-16).

On the comfy configuration, there is statistically significant evidence of moderate negative correlation on both UI time and understanding time. This is consistent with the fact that smaller working set support likely more “productive” sessions, where less time is spent on managing the UI and where mental processes are more effective. There is no evidence of correlations in the ok level of chaos. Probably, on the typical configurations of windows corresponding to the thresholds of needed space of this category, other factors prevail.

C. Wrapping Up

We modeled chaos in window-based IDEs by considering the needed space to visualize all code containing windows without overlapping. We defined four categories of chaos, and by leveraging more than 770 hours of interaction data, we showed that developers spend more than 30% of their time in a high-chaos configuration, corroborating previous research along the same lines.

We also discussed how our data is also consistent with potential impact to the time spent by developers in program understanding and UI time. In the following section, we simulate and discuss how even simple elision strategies and automatic window layouts can improve the level of chaos experienced by developers.

IV. MAKE CODE, NOT CHAOS

Section III provided evidence that developers have to cope with chaotic environments during a third of their programming time, with negative implications both in terms of time spent fiddling with the UI, as well as additional time spent with program understanding. This section explains how we can tame the IDE, by adopting simple mechanisms to reduce the amount of needed space on the screen.

A. Elision and Layout Strategies

Figure 4 exemplifies our two strategies: Elision and Layout. The strategies are part of a two-step process: We first reclaim space by eliding (hiding) the redundant parts of each non-active window (in the figure, the in-focus or active window is depicted with a thick border). Then, we apply a new layout to occupy the space more efficiently.

In the example depicted in Figure 4, there is not enough space to position all the windows of the IDE; consequently, the overlapping between the windows is relatively high (4.a). After the application of the elision strategy (4.b) the free space in the IDE increases, but the overlapping is still present. Finally, with the new layout, all the (elided) windows are now positioned in the IDE without overlapping (4.c).

1) *Elision*: The elision strategy hides part of a window to reduce the visual cognitive load on the developer. It stems from the observation that at each instant there is only one active window; all the others are inactive, producing a considerable amount of visual noise. The underlying idea is to leave the active window untouched, while reducing the visual noise present in the background windows. The goal is to *keep the code displayed in all windows visible* while hiding the non-code elements displayed in the window (lists, buttons, etc.). These UI elements are mostly used for navigation, and are only usable while the window is active. When the focus changes, its elided elements are restored. Since different types of windows display source code in different ways, the strategy implementation depends on the window types.

Code Browsers and Message Lists display code in the bottom half of the window. The top half contains source code navigation elements. Our strategy elides the top part while keeping the code visible. Figures 5.a and 5.c illustrate how the strategy works on these cases, reducing the needed space of non-active code browser or message list by 50%.

Debuggers display the code of a method on the stack and let the user modify it. The source code pane is in the central part of the window (occupying roughly 1/3 of the window). Our strategy elides the top and the bottom parts while keeping the central part (*i.e.*, code) visible. Figure 5.b shows how the strategy works, reducing the space occupied by each non-active debugger by ca. 66%. Figure 5 shows our elision strategy, applied on the same windows of Figure 1, reducing the opacity of the elided parts, instead of hiding them.



Fig. 5. Elision Strategy Explained for (a) Code Browsers, (b) Debuggers, and (c) Message Lists

2) *Layout*: The elision strategy efficiently reduces the amount of needed space occupied by non-active windows. However, also the overlapping between windows contributes to chaos, *i.e.*, by hiding parts of the open windows that might be relevant for the developer. For the sake of simplicity our definition of chaos (see Table IV) only considers the needed space, however as discussed in Section III-B2 needed space and weighted overlapping have a very strong positive correlation. Thus, reducing the overlapping might contribute to the reduction of the chaos level.

To reduce the overlapping, we adopt a layout algorithm inspired by the rectangle packing layout. The idea is to stack all the windows in columns from the origin of the screen (*i.e.*, top-left) one below the other, as shown in Figure 4.c. If a window cannot be repositioned (*i.e.*, it does not fit in the screen if moved in the new position), it is left in the original place.

3) *Wrapping Up*: We discussed two strategies to tame the chaos in the IDE: Elision and Layout. Figure 4 summarizes, step-by-step, how these strategies work on a hypothetical development session. Intuitively, elision and layout strategies help to tame the chaos inside the IDE by reducing both the space needed to display all the windows and the overlapping between them. The elision strategy aims to reduce the amount of visual noise in the IDE while the layout strategy takes care of reducing the overlapping between windows. Next, we evaluate the impact of the strategies under different perspectives.

B. Impact of Elision and Layout Strategies

To determine the potential impact of elision and layout strategies, we simulate their application on our dataset of recorded sessions. For each snapshot, we applied both the elision strategy alone and together with the layout strategy. Then, we discuss how the strategies impact the occupied space, then how they impact the time spent in each chaos level.

On Space Occupancy. We compute the gain for the space occupancy metrics as percentages with respect to the baseline, *i.e.*, the metric value before applying the strategies, as follows:

$$\text{Gain (\%)} = \frac{\text{Metric}_{\text{after}} - \text{Metric}_{\text{before}}}{\text{Metric}_{\text{before}}}$$

Suppose that for a session, the value of the *free space* before applying the strategies (*i.e.*, the baseline) is 12.82%. If, after applying the strategy, the free space increases to 22.32%, the relative gain would be 74.10%.

Table VII reports the average of differences before and after applying each strategy (*i.e.*, $\text{Metric}_{\text{after}} - \text{Metric}_{\text{before}}$), together with the corresponding average gain.

TABLE VII
PERCENTAGE GAIN OF SPACE OCCUPANCY METRICS

Metric	Elision		Elision + Layout	
	Average Gain (%)	Average of Differences	Average Gain (%)	Average of Differences
Occupied Space	-13.44%	-7.28%	6.99%	1.38%
Free Space	27.82%	7.28%	4.82%	-1.38%
Needed Space	-24.74%	-32.09%	-24.74%	-32.09%
Overlapping Space	-34.61%	-9.61%	-54.68%	-13.53%
Weighted Overlapping	-36.64%	-34.30%	-58.13%	-47.60%
Overlapping Depth	-2.59%	-0.10	-26.43%	-0.93

The simple elision strategy is—as expected—able to significantly increase the amount of free space, by almost 28%. This is accompanied by a general improvement of all the occupancy metrics, *e.g.*, needed space drops by almost 25%. Moreover, overlapping space considerably drops (ca. 35%), even if this strategy does not try to consciously reduce it.

The effect of windows re-layout produces configurations which make better use of the screen real estate. After layout, the needed space does not obviously change (with the same relative decrease of about 25% due to elision), but the overlapping space is reduced by more than 50%. The more efficient layout better use of available space is visible on the free space metric, which drops considerably compared to elision alone. In addition, the occupied space actually increases compared to the default configuration. This is in line with the goal of distributed windows in a more space efficient configuration.

To verify whether the effects of strategies are significant, we performed a paired t-test between the values of each metric before and after applying the strategies. For all metrics but occupied and free space, we observe that the metric values are reduced with statistical significance with both strategies (confidence interval 95%, $p\text{-value} < 2.2 \cdot 10^{-16}$). The same happens for the occupied space metric after applying the elision strategy, and for the free space metric after applying elision and re-layout. Instead, for the free space metric value after applying just elision, and for the occupied space metric after applying elision and re-layout, we find a significant increase of the metric value (respectively $p\text{-value} < 2.2 \cdot 10^{-16}$ and $7.8 \cdot 10^{-8}$, again at 95% confidence interval).

On Chaos Time. Table VIII shows the impact of our strategies on the time spent in each of the chaos categories defined in our model. Since the categories are defined only in term of needed space, the results refer to either strategies.

TABLE VIII
PERCENTAGE GAIN AND DELTA TIME

	<i>Avg. Gain (%) per Session</i>	<i>Absolute Difference</i>
Comfy	17.73%	137h 50m 44s 882ms
Ok	-1.35%	3h 25m 58s 502ms
Mess	-8.08%	-40h 29m 38s 554ms
Hell	-8.30%	-100h 47m 04s 812ms

The elision strategy has essentially the effect of redistributing the time spent on each category towards less chaotic categories. The time spent in the most chaotic category (*i.e.*, hell) is reduced on average around 8% of the session time. The second high-chaos category, mess, is reduced again by 8% on average for each session. These times are redistributed mostly towards the comfy category, which gains around 18% of average time in each session, while the ok category changes slightly. Developers spent 30% of their time in chaos previously; using the elision strategy could reduce this amount down to 14%, less than half.

Looking at the variation in the total amount of time spent per category, we find that programmers spent 142h in the hell category in original settings, while they spend 100h less in the new settings, a reduction of 70%. Time spent in the mess state drops from 188h to 148h (21%). The ok state is relatively stable, from 134h to 131h (-2%). The winner is the comfy category, which increases from 282h to 419h, a 48% increase.

Overall, the total recorded time spent in high-chaos categories amounts to 42.92%, while after elision this time would drop to 24.60%. In addition to get a better understanding of the improvement of the layout strategy, we compute the average drop in overlapping space for each of the four chaos levels.

TABLE IX
AVG. WEIGHTED OVERLAPPING PER CHAOS-LEVEL

	<i>Original</i>	<i>Elision</i>	<i>Elision + Layout</i>
Comfy	17.39%	16.76%	1.67%
Ok	55.86%	53.19%	27.81%
Mess	114.71%	112.99%	88.04%
Hell	330.11%	259.22%	235.52%

Table IX shows the reductions of average weighted overlapping before and after each strategy. We find that after just elision, in almost all categories there is no large change in overlapping, except in the hell category where we see a 27% drop. A dual effect happens after laying out, where major effects happen in the comfy category (relative drop of 90%) and we see large drops in the ok and mess categories. The effect in the hell category is less pronounced. In addition to spending less time in high chaos levels, developers would additionally enjoy a better spatial organization, particularly in the comfy and ok categories.

C. Wrapping up

With our strategies the time spent in high chaos can be considerably reduced, and that indeed we could better manage the screen real estate. We do not evaluate how the reduction of time spent in high chaos level could impact the time spent in specific activities, but we have some confidence that this could indeed happen given the correlations we found in Section VI.

V. THREATS TO VALIDITY

Internal Validity. Our definition of chaos is based on overlapping and needed space. Potentially different developers might have additional indicators of chaos. To cope with this threat we plan to cross-validate our measures of chaos with concrete observations (*e.g.*, think-aloud) to better grasp the sensitivity of developers to chaos. Another threat concerns our layout strategy that messes up that spatial memory of developers. This naïve strategy is only a proof of concept that simple means can already achieve a lot. We are aware of the importance of user placement of windows [27], [28] and in our future work we will devise strategies that consider and preserve the spatial memory of developers. Another threat is that we only simulated strategies on our existing dataset: We only replay the past interactions of the developers in our dataset; were the developers to use our elision and layout strategies, they might behave differently. We expect that as a result of using the elision and layout strategies, developers would spend less time UI fiddling and revisiting previous source code locations, as these would stay on screen. To mitigate this threat, we performed a correlation study between the time spent by developers in fiddling with the UI of the IDE and the levels of chaos (see Table VI). We found evidence that UI and program comprehension time are positively correlated with the high-chaos levels and negatively correlated with low-chaos levels, supporting the fact that less chaotic environments might let developers spend less time in taming the IDE. We are of course aware that correlation is not causation: As future work, we plan to release our strategies in the Pharo IDE and collect feedback from developers and new measurements, similar to the experiment on CodeBubbles [9].

Statistical Conclusion. We considered more than 770 hours of development affecting more than 40K windows. Our dataset supported us in drawing statistically significant conclusions about correlation between chaos in the IDE and both the time spent by developers altering the UI of the IDE, and the time spent performing program comprehension tasks.

External Validity. We focused on the Pharo IDE and the fine-grained interaction data we collected. Results may vary for different programming languages and IDEs. However, as part as our motivation (see Section II) we extensively discussed the situation in tab-based IDEs (*e.g.*, Eclipse) and provided preliminary evidence, leveraging Mylyn data, that also this UI paradigm may generate chaos in the IDE. Unfortunately, due to the coarse nature of Mylyn data, we could not conduct analyses at the same granularity of DFLOW. For this reason, in the future we plan to implement a fine-grained interaction data

profiler on another IDE to give us confidence about the generalizability of our results. Bragdon’s study (which had Eclipse as a baseline) gives confidence that this would be the case [9]. Another threat is related to the distribution of recorded sessions among developers: Most of the sessions come from only 5 developers. This might influence conclusions about developer diversity, but this was not the focus of this paper.

VI. CONCLUSIONS

The UIs offered to developers to browse complex relationships between source code are often inadequate. Thus, developers are repeatedly forced to use multiple UI components at the same time, bringing the IDE into a chaotic state. It is unclear to what extent chaos impacts development, and more importantly it is unclear how to tame it.

We analyzed a considerably large dataset of fine-grained interaction data, counting more than 770 hours of development. We found that developers in our dataset spend more than 30% of their time in high levels of chaos. Furthermore, time spent in high levels of chaos is correlated with time spent fiddling with the UI. We proposed two simple strategies to reduce the chaos in the IDE. We found that our simple elision strategy could save a considerable amount of space the IDE needs to layout windows. One might argue that all these are mere user interface concerns, and not relevant for software engineering. However, while considerable efforts are spent in making mainstream end-user tools friendly, software developers are still using convoluted environments. We believe there is no good reason for why developers should be treated differently from “normal” users.

ACKNOWLEDGEMENTS

We acknowledge the support of SNSF for the project No. 146734, FONDECYT for the project No. 1151195, and CHOOSE for sponsoring our trip to the conference.

REFERENCES

- [1] J. Singer, R. Elves, and M.-A. Storey, “Navtracks: Supporting navigation in software,” in *Proceedings of IWPC (13th International Workshop on Program Comprehension)*, 2005, pp. 173–175.
- [2] R. Minelli, A. Mocci, and M. Lanza, “The plague doctor: A promising cure for the window plague,” in *Proceedings of ICPC (23rd IEEE International Conference on Program Comprehension)*, 2015.
- [3] D. Roethlisberger, O. Nierstrasz, and S. Ducasse, “Autumn leaves: Curing the window plague in IDEs,” in *Proceedings of WCRE (16th Working Conference on Reverse Engineering)*, 2009, pp. 237–246.
- [4] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeptura, and J. J. LaViola, Jr., “Code bubbles: Rethinking the user interface paradigm of integrated development environments,” in *Proceedings of ICSE (32nd International Conference on Software Engineering)*, 2010, pp. 455–464.
- [5] R. DeLine and K. Rowan, “Code canvas: Zooming towards better development environments,” in *Proceedings of ICSE (32nd International Conference on Software Engineering)*, 2010, pp. 207–210.
- [6] E. Kandogan and B. Shneiderman, “Elastic windows: Evaluation of multi-window operations,” in *Proceedings of SIGCHI 1997 (Conference on Human Factors in Computing Systems)*, 1997, pp. 250–257.
- [7] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, “An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks,” *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 971–987, 2006.
- [8] J. M. T. Thompson and H. B. Stewart, *Nonlinear dynamics and chaos*. John Wiley & Sons, 2002.
- [9] A. Bragdon, R. C. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeptura, and J. J. L. Jr., “Code bubbles: a working set-based interface for code understanding and maintenance,” in *Proceedings of CHI (28th International Conference on Human Factors in Computing Systems)*, 2010, pp. 2503–2512.
- [10] M. Kersten and G. C. Murphy, “Mylar: a degree-of-interest model for IDEs,” in *Proceedings of AOSD (4th International Conference on Aspect-Oriented Software Development)*, 2005, pp. 159–168.
- [11] A. Wexelblat and P. Maes, “Footprints: History-rich tools for information foraging,” in *Proceedings of CHI (SIGCHI Conference on Human Factors in Computing Systems)*, 1999, pp. 270–277.
- [12] T. Fritz, D. C. Shepherd, K. Kevic, W. Snipes, and C. Bräunlich, “Developers’ code context models for change tasks,” in *Proceedings of FSE (22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering)*, 2014, pp. 7–18.
- [13] J. Sillito, G. C. Murphy, and K. D. Volder, “Asking and answering questions during a programming change task,” *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 434–451, 2008.
- [14] M. P. Robillard and G. C. Murphy, “Representing concerns in source code,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 16, no. 1, 2007.
- [15] M. Kersten and G. C. Murphy, “Using task context to improve programmer productivity,” in *Proceedings of FSE (14th ACM SIGSOFT International Symposium on Foundations of Software Engineering)*, 2006, pp. 1–11.
- [16] T. Fritz, G. C. Murphy, E. R. Murphy-Hill, J. Ou, and E. Hill, “Degree-of-knowledge: Modeling a developer’s knowledge of code,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 2, pp. 14:1–14:42, 2014.
- [17] R. DeLine, M. Czerwinski, and G. G. Robertson, “Easing program comprehension by sharing navigation data,” in *Proceedings of VL/HCC (IEEE Symposium on Visual Languages and Human-Centric Computing)*, 2005, pp. 241–248.
- [18] S. Hanenberg, S. Kleinschmager, R. Robbes, É. Tanter, and A. Stefik, “An empirical study on the impact of static typing on software maintainability,” *Empirical Software Engineering*, 2014.
- [19] S. Endrikat, S. Hanenberg, R. Robbes, and A. Stefik, “How do API documentation and static typing affect API usability?” in *Proceedings of ICSE (36th International Conference on Software Engineering)*, 2014, pp. 632–642.
- [20] P. Petersen, S. Hanenberg, and R. Robbes, “An empirical comparison of static and dynamic type systems on API usage in the presence of an IDE: java vs. groovy with eclipse,” in *Proceedings of ICPC (22nd International Conference on Program Comprehension)*, 2014, pp. 212–222.
- [21] H. Sanchez, R. Robbes, and V. M. González, “An empirical study of work fragmentation in software evolution tasks,” in *Proceedings of SANER (22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering)*, 2015, pp. 251–260.
- [22] A. T. T. Ying and M. P. Robillard, “The influence of the task on programmer behaviour,” in *Proceedings of ICPC (19th IEEE International Conference on Program Comprehension)*, 2011, pp. 31–40.
- [23] R. DeLine, A. Bragdon, K. Rowan, J. Jacobsen, and S. P. Reiss, “Debugger canvas: Industrial experience with the code bubbles paradigm,” in *Proceedings of ICSE (34th International Conference on Software Engineering)*, 2012, pp. 1064–1073.
- [24] 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 IEEE International Conference on Program Comprehension)*, 2015.
- [25] J. Siegmund, N. Siegmund, and S. Apel, “Views on internal and external validity in empirical software engineering,” in *Proceedings of ICSE (37th IEEE/ACM International Conference on Software Engineering)*, 2015, pp. 9–19.
- [26] S. Pinker, *How the Mind Works*. W. W. Norton, 1997.
- [27] D. A. Henderson and S. Card, “Rooms: the use of multiple virtual workspaces to reduce space contention in a window-based graphical user interface,” *ACM Transactions on Graphics*, vol. 5, no. 3, pp. 211–243, 1986.
- [28] G. Robertson, M. van Dantzich, D. Robbins, M. Czerwinski, K. Hinckley, K. Risdien, D. Thiel, and V. Gorokhovskiy, “The Task Gallery: a 3D window manager,” *Proceedings of SIGCHI 2000 (ACM Conference on Human Factors in Computing Systems)*, vol. 2, no. 1, pp. 494–501, 2000.