Mining Sequences of Developer Interactions in Visual Studio for Usage Smells

Kostadin Damevski, David C. Shepherd, Johannes Schneider, and Lori Pollock

Abstract—In this paper, we present a semi-automatic approach for mining a large-scale dataset of IDE interactions to extract usage smells, i.e., inefficient IDE usage patterns exhibited by developers in the field. The approach outlined in this paper first mines frequent IDE usage patterns, filtered via a set of thresholds and by the authors, that are subsequently supported (or disputed) using a developer survey, in order to form usage smells. In contrast with conventional mining of IDE usage data, our approach identifies time-ordered sequences of developer actions that are exhibited by many developers in the field. This pattern mining workflow is resilient to the ample noise present in IDE datasets due to the mix of actions and events that these datasets typically contain. We identify usage patterns and smells that contribute to the understanding of the usability of Visual Studio for debugging, code search, and active file navigation, and, more broadly, to the understanding of developer behavior during these software development activities. Among our findings is the discovery that developers are reluctant to use conditional breakpoints when debugging, due to perceived IDE performance problems as well as due to the lack of error checking in specifying the conditional.

Index Terms—IDE usage data, data mining, pattern mining, usability analysis

1 Introduction

The majority of software developers nowadays rely on Integrated Development Environments (IDEs), sometimes extended via custom plug-ins, for much of their daily work. IDEs provide developers with an extensive toolkit that includes support for compilation, debugging, profiling, refactoring and other tasks. Most development tasks can now be completely accomplished within these environments. The complexity of modern IDEs warrants research into the usability of such environments to accommodate a broad set of tasks and developer skill levels. Studying developer behaviors within such IDEs in the field can often reveal interesting patterns to researchers, providing supporting (or dissenting) evidence for findings on developer behaviors on various software engineering tasks and use of related tools, obtained in controlled lab experiments.

Large scale IDE usage datasets, collected from the interaction histories of hundreds of developers in the field, have been available to researchers for a number of years. However, previous IDE usage data analyses have been either limited to specific parts of the IDE (e.g., refactoring tools [1], [2]) or have relied on simple analysis of individual IDE actions or transitions between pairs of actions [3], [4]. Broad, exploratory mining of longer sequences of developer behavior, consisting of several interactions have rarely been conducted using such datasets.

- *K. Damevski is with the Department of Computer Science, Virginia Commonwealth University, Richmond, VA, 23284, U.S.A. E-mail: damevski@acm.org*
- *D. Shepherd is with ABB Corporate Research, Raleigh, NC, 27606, U.S.A. E-mail: david.shepherd@us.abb.com*
- *J. Schneider is with ABB Corporate Research, Baden, Switzerland. E-mail: johannes.schneider@ch.abb.com*
- *L. Pollock is with the Department of Computer and Information Sciences, University of Delaware, Newark, DE, 19350, U.S.A. E-mail: pollock@udel.edu*

Manuscript received July 9, 2015; revised September 17, 2014.

In this paper, we describe a novel exploratory workflow to mine frequent *usage patterns (or sequences)* of interactions from a large repository of IDE usage data. Extracting common behaviors exhibited by a number of developers in the field can highlight sequences of interactions with the IDE, which are representative of software engineering micro tasks (e.g. comprehension, feature location) performed by developers. Such mined tasks can provide a novel way to examine the developer usage of the IDE, revealing approaches that developers take in accomplishing common activities, such as stepping through code with the debugger, searching for a location in the code base, or opening and editing sets of files. Other descriptive statistics, such as elapsed time and occurrence counts can also be computed to provide further insights into developers' behavior. These mined usage patterns can be used as motivation to conduct a subsequent survey of the developers, as we did in this paper, to identify *usage smells*, which more strongly point to usability issues with the IDE.

We apply our novel sequence mining workflow to a large-scale IDE dataset, consisting of all the IDE interactions of 196 developers at ABB, Inc. over a period of up to several months. Throughout this paper, we call this data set the ABB Developer (ABB-Dev) dataset. The results of mining sequential patterns from this large-scale dataset are used as a means of identifying deficiencies in IDE usability, due to flaws in IDE design, gaps in developer knowledge in using the IDE, or both of these factors. This type of usage analysis is commonly performed in web applications [5], where web server logs, which are easily available, are mined for insights into application usage. However, many of the mining approaches used in the web domain are incapable of managing the scale, heterogeneity, and noise in IDE usage data. Logged messages in IDE data include a broad set of asynchronous IDE events and intentional developer clicks relevant to interpreting high-level developer behaviors, in-

tertwined with noise from messages caused by unrelated IDE events and spurious clicks.

The described IDE usage data mining workflow has the characteristics of a funnel, in that, in its early stages, it performs general aggregation operations applicable to many IDE data analytic tasks, while, in the later stages, it focuses on identifying a few specific usability patterns and smells. Therefore, the contributions of this paper are in both the data analysis approach as well as the results from this analysis. First, our IDE usage data analysis workflow contains the following novel characteristics:

- Applies sequential pattern mining to noisy and large-scale IDE usage data,
- Clusters similar sequences for easier human interpretation using a tailored distance function that combines sequential similarity with the rarity of event occurrence, and
- Confirms pattern findings by cross-referencing with expert opinion and developer surveys to interpret the mined behavior, examine the IDE usage pattern, and remove any ambiguity due to imperfect log message capture.

Second, we discovered several interesting developer behaviors in Visual Studio via our usage data analysis, elucidated via a survey of the discovered behaviors from over 50 developers at ABB, Inc.:

- Developers search their code frequently and inefficiently, due in part to the poor presentation of search results in Visual Studio.
- Developers rapidly visit several files in their active file set before locating a relevant file, and do not report any hardship in doing so.
- Developers debug using a small command set and recognize the need for improved capabilities, though some, such as conditional breakpoints in Visual Studio, are poorly implemented.

The remainder of this paper is organized as follows. Section 2 describes the goals of our mining approach, while Section 3 outlines related approaches and studies. In Section 4, we describe our IDE usage data mining workflow. Section 5 describes a selection of the results of that workflow, which are further elucidated via a developer survey discussed in Section 6. Section 7 details the meaning and implications of the results of the usage smell analysis, while Section 8 describes the threats to validity. Finally, Section 9 summarizes the conclusions and future work of this project.

2 Data Mining Goals and Challenges

The goal of our data mining is to discover interesting sequences of user interactions within an IDE. Our study is exploratory in that we mine from a dataset consisting of all interactions in the IDE, rather than focusing on a subset related to a specific IDE capability. Like several researchers before us, we are interested in confirming or debunking researchers expectations of *developer behavior in the field, in their natural setting and without observational bias*. The main difference to prior work in this area is that we are interested in *sequences of IDE interactions*, rather than focusing on the usage of a single command or the transition between pairs of commands.

Sequences consisting of the IDE interaction patterns exhibited by many developers can be interpreted as a set of intentional commands and clicks performed by the user to accomplish a small task. These tasks are often trivial (e.g., stepping through code with the debugger, building code and fixing resulting errors), but sometimes reveal interesting and unexpected behaviors. Since these behaviors are exhibited by many developers, we can treat them as typical usage of the IDE, which when inefficient, may reveal problems in IDE usability design or deficiencies in developer training to use the IDE's capabilities.

Large datasets consisting of all IDE interactions by a set of developers have been available for several years. Most notably, The Eclipse Usage Data Collector (UDC) project collected such data from thousands of users of the Eclipse IDE [6]. The UDC project has been discontinued in recent releases of this IDE, so the collected data focuses on usage of older versions of Eclipse. However, the ABB Developer (ABB-Dev) dataset, which we use in our study, provides a more recent dataset, based on the Visual Studio IDE, at a considerable scale of nearly 200 developers. The dataset is collected by a Visual Studio extension, which has since become a part of the Codealike IDE developer productivity suite [7].

Both UDC and ABB-Dev record a similar, broad set of IDE commands that have been issued by developers, views (windows within the IDE) that have received developer clicks, and editor behaviors (without recording any of the actual code for privacy reasons). These logged items are provided in a time stamped stream indexed by an identifier assigned to each developer whose interactions were recorded. The datasets also contain certain events, triggered by the IDE itself, such as when compilation completes, a breakpoint is reached by the debugger, or when the IDE is put in the background or re-activated. The actions of many developers over several months of IDE usage generate very large datasets, which are challenging to analyze.

A significant challenge in analyzing sequences of messages in IDE datasets is in differentiating noise from signal. In this work, we define noise to be the logged actions that reflect spurious or irrelevant clicks or events that do not provide relevant information on the developer's task. For instance, consider a set of developer actions to fix a bug in the IDE interrupted by a message reporting that the project has finished building. In certain cases, this event may inform us about what the developer did in subsequent actions in the IDE, for instance, looked at the build error log, while in other cases, it may not provide useful information to the pattern. Some frequently occurring messages may be relevant to certain patterns (such as putting the IDE in the background), while they may be irrelevant to other patterns. The number and diversity of such event combinations and the difficulty in understanding the influence of each event in a developer's workflow make enumerating all of these scenarios prohibitive.

3 Related Work

Due to its scale and unbiased nature, IDE usage data analysis has frequently been used by researchers to exam-

ine developer behavior in the field, including highlighting which tools are commonly used by developers [3] and how refactoring tools are used [4]. This data has also been used as a means to build predictive models for a variety of software engineering purposes, such as detecting coupled source code artifacts [8] and recommending IDE tools to developers [9]. The majority of such analyses did not require deeper mining of developer actions in IDE usage data, relying mostly on counts of messages and transitions between pairs of messages (Markovian analysis).

The usability of the Visual Studio IDE has been investigated only a few times in the past. Snipes et al. examined efficent work patterns in Visual Studio and implemented a game-like environment to encourage such developer behaviors [10], [11]. Recently, Amann et al. studied developer usage of Visual Studio using a field dataset of over 6300 hours of developer time [12]. Their work focused primarily on measuring the time consumption and frequency of certain actions (e.g. code completion, navigation, etc.) in Visual Studio. Compared to this prior work, our approach focuses on pattern mining an even larger dataset of Visual Studio interactions, where the mined results are followed up with a developer survey.

Recently, pattern mining has been used for the identifying novel code transformations for inclusion in the IDE by Negara et al. [1]. While the researchers relied on pattern mining, as does this paper, they used a more conventional itemset mining technique that does not maintain the sequential order of the mined items. Negara et al. also gathered a dataset that included the code that the developers were maintaining, which is more difficult to obtain at scale and is categorically different from the dataset we use in our work. Such a dataset can be recorded by the Fluorite tool [13], which captures developer interactions in the IDE editor sufficient to reconstruct the program at each point in time. The dataset used in this paper, as well as other large scale IDE interaction datasets, such as the Eclipse UDC dataset, do not contain any of the code, due to privacy and other similar concerns.

Vakilian et al. studied usability problems with IDE refactoring tools [2] using the critical incident technique, based on analysis of interactions preceding and following a fixed set of specific IDE actions or events. The IDE usage dataset used was much more detailed than the one used in this paper in its reflection of user behavior in using refactoring tools. Compared to our mining technique, their approach was more focused rather than exploratory, and was reliant on a less ambiguous dataset.

Khodabandelou et al. used a hidden Markov model based approach to build a process model of developer behavior based on the Eclipse UDC dataset [14]. The approach used was similar to ours in both its exploratory nature and its dataset. The main difference with our work is that our technique focuses on mining individual behavioral patterns, while the goal of the approach by Khodabandelou was to build an abstraction (e.g. a finite state machine) of the overall high-level behavioral processes used by the developers.

Sequential pattern mining has been used for usability analysis in several related domains, including the mining of web logs for usability testing of web-based applications [15] and in improving the success factors in online collaborative learning [16].

4 Mining Approach

The inputs to our data mining workflow are time-ordered logs of IDE interactions for each developer, collected during their regular work activities, spanning months of activity. The output is a list of behavioral patterns, commonly exhibited by developers, that exemplify potential IDE usability problems. The mined patterns are selected to serve as points of improvement in IDE design and use. To process the inputs, while addressing the scale and noise typical in IDE interaction datasets, we propose a novel mining workflow consisting of several steps shown in Figure 1. We discuss each step below in detail.

4.1 Input IDE Interaction Dataset

The ABB-Dev IDE interaction dataset consists of over 8 million logged actions by 196 developers at ABB, Inc., representing 32,811 developer hours of usage of the Visual Studio IDE. The dataset is recorded by a Visual Studio extension that, after a straightforward installation, subscribes and records most of the IDE's published events. Each of the monitored developers had at least 5 hours of development time recorded by the tool.

ABB-Dev contains all of the user interactions in the IDE as timestamped log messages, including IDE commands (e.g., build, execute a test), views (e.g., click on the error list window), or events (e.g., build finished). Keypresses and clicks within the IDE editor, which edit the code or just move the cursor, are partially aggregated by the ABB-Dev tool to reduce log size, so that not each keypress is written to the log, but instead several keypresses are converted into a message, such as an editor scroll down event, tab insertion, deletion, etc.

To illustrate the dataset, a listing of two sample ABB-Dev messages is shown in Figure 3; each message consists of a type, timestamp, user id, and category. Figure 2 shows a visualization of each developer's contribution to the overall ABB-Dev dataset in terms of number of hours monitored and the number of interaction messages. There were groups of developers that allowed the tool to monitor their behavior for relatively short periods (days, visualized as circles), medium periods (weeks, visualized as triangles) and longer periods (months, visualized as squares).

4.2 Data Preparation

To prepare the dataset for the task of detecting interesting usage patterns in the IDE, and for the sequential pattern mining process (next step), while also removing unnecessary processing downstream in our workflow, we perform the following four actions:

1) *Separate the dataset into time-ordered sequences of individual developer interactions.* The original ABB-Dev dataset is stored in one very large log file, in time order. We separate the dataset by developer, while preserving the order of each developer's messages by their time order. The ordering of messages, but not their actual time, is important to sequential pattern mining.

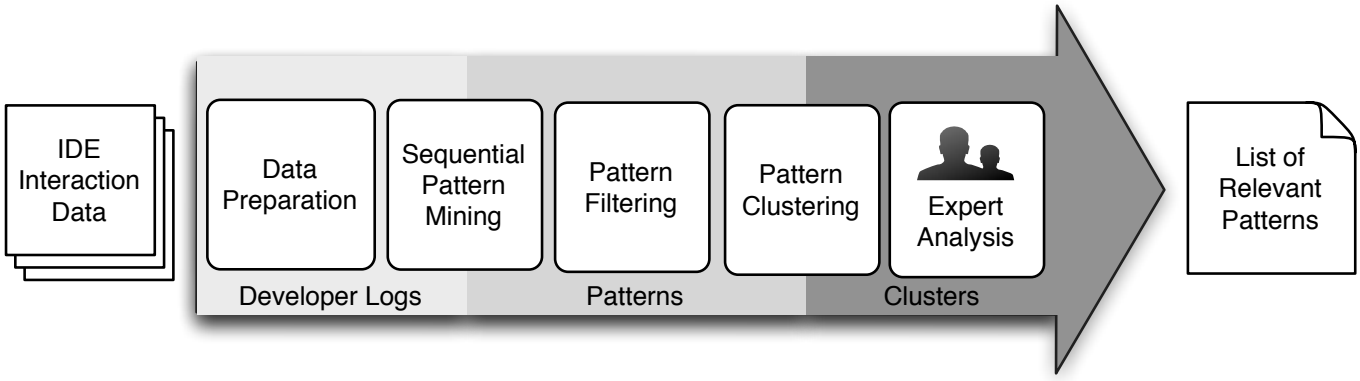

Fig. 1. A schematic of the sequence mining workflow for large-scale IDE log analysis.

2) *Add log messages to represent inactive time intervals.* While breaks in developer interactions with the IDE are common, the sequential pattern mining process ignores the actual event timestamps. To account for such breaks in time and avoid spurious associations between developer actions that are distant in time, we introduce a new message into the log to signify that a period of inactivity occurs for a developer in the log. In our analysis, we chose a threshold of 5 minutes as we believe that this constitutes a substantive period, which, when passed, likely indicates a break in the developer's activity. In our estimation, it is fairly unlikely that a developer is still performing a task and does not have any key press or click in the IDE for a 5 minute span. The inactivity message can be repeated to signify a longer break. In order to limit the repetition we choose a threshold of 30 minutes (or 6 consecutive messages) where we deem further repeats to be unnecessary, based on the gap size of the sequential pattern mining algorithm described in the next step of our workflow. Changing the gap size requires adjusting the thresholds for the inactivity messages in order to prevent long periods of inactivity from being ignored by the mining algorithm.
3) *Aggregate repetitive edit messages into a single edit message.* In ABB-Dev, the set of messages indicating the action of editing code and the messages signifying moving the cursor within the IDE are overly detailed, and individually irrelevant to our analysis. Note that large-scale field IDE usage datasets like ABB-Dev do not contain the actual code, and individual small-scale editing commends (e.g. insert tab, delete line) are difficult to interpret. Thus, we aggregate consecutive sets of edit-related messages into a single edit message for analysis.
4) *Aggregate repetitive move cursor messages within the editor messages into a single move cursor message.* ABB-Dev records both clicks within the editor window and scrolling up and down commands separately. Thus, we combine consecutive move cursor messages into a single move cursor message, except when a subsequence of actions contains both move cursor and edit messages. In this case, we aggregate it all into a single edit message. Often developers move the cursor before deciding where to perform edits. The edit message is the relevant action in this process, while the movements of the cursor constitute a detail irrelevant to our subsequent mining task.

4.3 Sequential Pattern Mining

At the beginning of the sequential pattern mining, the data consists of a set of time-ordered interaction sequences, one for each of the 196 developers. While the previous step shrunk the number of messages in the dataset, via aggregation of the numerous edit and move cursor messages, the size of most sequences continues to range between tens of thousands to hundreds of thousands of messages, a challenging scale for many common sequential pattern mining algorithms.

Sequential pattern mining algorithms identify a set of subsequences (or patterns) that occur in some percentage (or, with minimum *support*) of the input sequences. The per-developer split of the ABB-Dev data sequences allows us to specify a meaningful *support* parameter to the sequential pattern mining algorithm, i.e., at least 10% of the developers. Thus, if a sequential pattern is exhibited by 20 of the roughly 200 developers, then the algorithm should discover the pattern. In this way, we can search for IDE usage patterns that are common across a substantial population of the developers, allowing for greater relevance to the mined results.

To account for the noise in IDE data due to unrelated events and clicks, as described in Section 2, we mine sequences while tolerating a gap of one message in the input sequences. For example, a gap of a single message instructs the pattern mining algorithm to consider a candidate pattern $a \rightarrow c$ to have occurred in an input sequence $a \rightarrow b \rightarrow c$, thus contributing to the level of *support* needed to consider the $a \rightarrow c$ pattern relevant. We also experimented with longer gaps, but decided, based on observing both the input sequences and identified patterns, that a gap of one message avoids noise in most cases that we consider and produces few or no spurious patterns at the

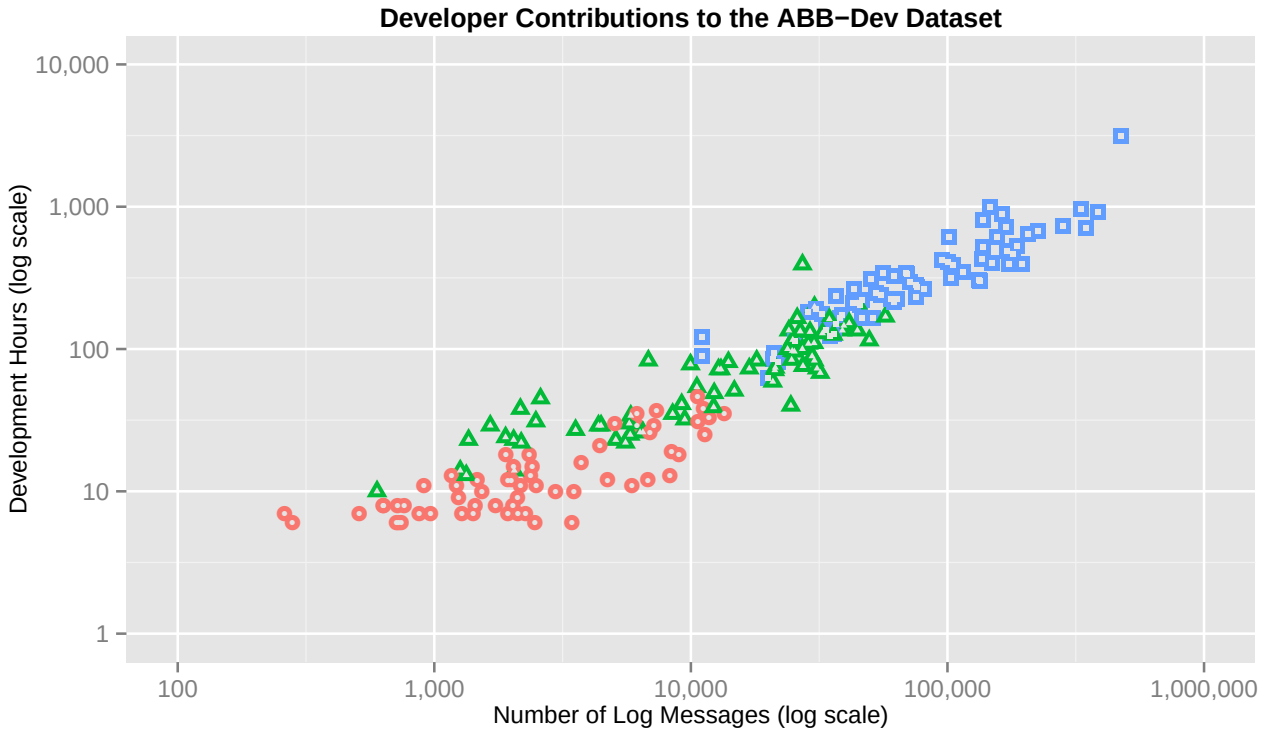

Fig. 2. Distributions of the developers according to the number of hours monitored and the number of log messages collected. The circles denote developers with less than a week of monitoring, triangles denote developers with between a week and a month of monitoring, while squares denote developers with more than a month of monitoring.

Interaction Event #100
Name: `Edit.Paste`
Timestamp: `2014-09-24 06:11:22.000 UTC`
UserId: `2122`
Category: `Command`

Interaction Event #101
Name: `View.SolutionExplorer`
Timestamp: `2014-09-24 06:12:51.000 UTC`
UserId: `2122`
Category: `View`

...

Fig. 3. A sample of two log messages from the ABB-Dev IDE usage dataset.

support level of 10%. Note that the concept of noise is task-dependent and thus other analyses might define and handle noise in different ways.

Sequential pattern mining algorithms produce an exhaustive set of patterns found in the input dataset. This results in all of the subpatterns a, b, c, $a \rightarrow b$, and $b \rightarrow c$ to be part of the output if the longer pattern $a \rightarrow b \rightarrow c$ is identified. To reduce the number of smaller subpatterns that are found by the sequential pattern mining algorithm, we require that the algorithm produces *closed* or *maximal* patterns, where patterns that are contained within longer patterns are ignored. Closed sequential pattern mining algorithms remove all patterns that exist within other identified patterns and occur at the same support level, while maximal pattern mining removes sub-patterns regardless of the support level. For our problem, the maximal pattern mining is preferable, as once we have identified patterns that meet the threshold of 10% support, further changes in the support levels are not relevant to the subsequent analysis that we perform.

Accounting for each of these requirements: relatively low support of 10%, a gap of one message and maximal patterns, while also processing the lengthy sequences in the ABB-Dev dataset is a challenge for most sequential pattern mining algorithms. Few of the commonly used sequential pattern mining algorithms are capable of even producing maximal patterns [17]. We considered a number of popular algorithms, finally identifying the recently proposed MG-FSM algorithm [18] as the only one that could satisfy all of these requirements. MG-FSM is parallelized using map-reduce (Hadoop) functionality which eases its use on a cloud infrastructure and aids desired scalability for larger datasets.

4.4 Pattern Filtering

The MG-FSM algorithm produces 84,145 patterns based on our requirements, which is too large for human examination, even though many of the patterns are quite similar.

For instance, one pattern may contain an edit, followed by a project build, followed by a click on the error view, while another may contain the same sequence with a click on the project explorer view preceding the build command. The next two steps in our workflow have the goals of first filtering out irrelevant patterns, and then grouping similar patterns to enable human analysis. Much of the filtering at this stage was motivated by our desire to detect usage smells in the IDE, while other goals may necessitate different filtering strategies.

The first filter removes overly short patterns that are less likely to provide a clear, unambiguous snapshot of developer behavior. Specifically, many patterns under 8 messages were difficult to interpret meaningfully or in a way that may provide evidence for a specific developer behavior, and thus we filter out all sequences less than 8 message long (approximately 34K patterns). We recognize that this may remove some interesting shorter behaviors, perhaps even removing the presence of patterns reflecting certain aspects of the IDE's functionality.

The second filter removes patterns that contain messages that are too ambiguous if interpreted as usage smells. Therefore, due to the lack of any evidence outside of the IDE usage stream, we remove all patterns that contain a time gap and all patterns that contain a command noting that Visual Studio was placed in the background. We also remove patterns with the move cursor message as the purpose of the cursor movement was unclear in many cases and it provided little to no help in identifying usage smells. In total, this resulted in a reduction of about 33K patterns.

While filtering these messages earlier in the workflow would be computationally cheaper, as they are of no relevance to the mining of usage smells, their removal could affect the results produced by the sequential pattern mining by adding spurious associations. This is because messages that would be ordinarily separated by time or cursor movement messages would be brought closer together, and possibly identified as patterns, with the removal of such messages earlier in the workflow.

4.5 Pattern Clustering

After the filtering step, less than a quarter (i.e., 17,867) of the initial 84,145 patterns remain. Many of these remaining patterns are still similar to each other, intuitively corresponding to groups of developer actions commonly performed to achieve a specific small goal (e.g., find a location in the code, debug). Grouping similar sequences allows examination of the different ways that developers commonly perform a task and enables identification of unexpected IDE usage workflows.

To cluster patterns, we use the k-medoids algorithm, a variant of k-means that uses existing points in the dataset as cluster centroids. K-means cannot be used directly because calculating a centroid of a set of mined patterns often results in a non-pattern, since numerical operations, like addition and division, cannot be performed on two patterns. K-means cannot proceed with a non-pattern centroid, as, then, the distance function is undefined between a non-pattern and a pattern.

For the distance function, we define a novel function that combines two characteristics of similar patterns: 1) similar patterns have long subsequences of IDE actions in common; and 2) when an action occurs rarely in the dataset, the sequences that contain that message are similar. To address the first characteristic, we use Longest Common Subsequence (LCS); to address the second characteristic, we use a function that takes into account individual action frequencies in the dataset. We combine both of these influences by a weighted sum.

Specifically, we define the distance function, $D(P_a, P_b)$, for any two mined IDE interaction patterns P_a and P_b as follows. *Note that a value of 0 for the distance function indicates that two patterns are completely alike, while a value of 1 indicates that the patterns are completely different.*

$$D(P_a, P_b) = \alpha D_{LCS}(P_a, P_b) + (1 - \alpha) D_{Occ}(P_a, P_b)$$

Using a factor α, ranging between 0 and 1, we weigh influences from two separate measures of distance, one based on simple Longest Common Subsequence (LCS), D_{LCS}, and the other, D_{Occ}, which accounts for the occurrence frequencies of the messages that constitute the two patterns.

$$D_{LCS}(P_a, P_b) = 1 - \frac{|LCS(P_a, P_b)|}{|P_a \cup P_b|}$$

where $|\cdot|$ is the modality of a set, $LCS(P_a, P_b)$ is the longest common subsequence between the two patterns without regard for starting position, and $P_a \cup P_b$ is the union of the messages in the two sequences.

$$D_{Occ}(P_a, P_b) = \begin{cases} 1 & \text{if } P_a \cap P_b = \emptyset \\ \sum\limits_{e \in P_a \cap P_b} \frac{F_e}{F_{total}} & \text{otherwise} \end{cases}$$

where $P_a \cap P_b$ is the set of the common messages between patterns P_a and P_b, and e is each message in this common message set. F_e is the occurrence frequency of each message e in the ABB-Dev dataset, while F_{total} is defined as the following:

$$F_{total} = \sum_{j \in P_a \cup P_b} F_j$$

To illustrate how this distance metric is computed, consider an example consisting of two sequential patterns: $P_a = a \rightarrow b \rightarrow c \rightarrow d$ and $P_b = a \rightarrow b \rightarrow d$. Assume also that the messages $\{a, b, c\}$ occur 100 times in the dataset, while the message d occurs only twice. We can compute $|LCS(P_a, P_b)| = |\{a, b\}| = 2$ and $|P_a \cup P_b| = |\{a, b, c, d\}| = 4$, and, therefore, $D_{LCS}(P_a, P_b) = 1 - \frac{2}{4} = 0.5$. To compute, $D_{Occ}(P_a, P_b)$, we first compute $F_{total} = 100 + 100 + 100 + 2 = 302$, resulting in $D_{Occ} = \frac{100}{302} + \frac{100}{302} + \frac{2}{302} = 0.67$. These two distance measures are combined, based on a weight α to form the final distance value.

After experimenting with a few different ratios between the two constituent distance measures, we settled on an α value of 0.25. In this way, we weigh the frequency (or rarity) of messages as three times more important than the longest common subsequence between two patterns in clustering them. The distribution of messages in the ABB-Dev dataset is exponential, so we found the existence of the same rare

messages to be an important characteristic of the similarity between two patterns.

The distribution of message types in the dataset is exponential, so we found message rarity to be an important characteristic of the similarity between two patterns.

To specify the number of clusters (i.e., the k in k-medoids), we consider the optimum average silhouette width for different k. The average silhouette width reflects the average closeness to the centroid for the clustered data points, relative to its closeness to the centroids of neighboring clusters. This metric ranges between -1 and 1, where a higher value is indicative of more separable clusters. In order to have a human-interpretable number of clusters, we only examined the average silhouette size for between 10 and 50 clusters. While the overall strength of the clustering was relatively poor, indicated by a negative silhouette width (in the range [-0.3,-0.1]), we chose a value for k=*20*, where there is an "elbow" in the average silhouette width curve and adding more clusters does not improve the results. It should be noted, however, that given the high number of input sequences, values of k that are an order(s) of magnitude higher than the ones we examined may produce a significantly better clustering, but would also require much more effort for human interpretation.

4.6 Expert Analysis

Two of the authors examined the pattern clusters discovered by the pattern clustering step. The clusters ranged between 1 and 15 different message (or IDE interaction) types in each cluster, and generally represented similar interactions, or sub-patterns consisting of a few interactions, occurring in slightly different order. The composition of the patterns allowed straightforward human observational analysis of the behavioral patterns associated with each cluster, without the need to perform additional sequence filtering.

The authors were presented with a listing of all of the patterns in each cluster, and asked to characterize each of the clusters in terms of the developer behavior that it represents. There was widespread agreement on the characterization of the behaviors of the clusters, which we show in Table 1. In certain cases, where there was disagreement or ambiguity, the authors used the data collection tool and Visual Studio to verify that a pattern corresponded to the expected sequence of interactions in Visual Studio. This was also used as a means to determine whether there exists more than one way to produce the same sequence of messages, to reduce the possibility of misleading results.

After characterizing the dataset, the authors used a complementary dataset to provide further statistical information on the time that developers took when performing the behaviors corresponding to each pattern. To calculate these measures, we revisited the original ABB-Dev dataset and identified occurrences of the relevant pattern for which we extracted this additional statistical information. For the measures related to time, we identified occurrences of the pattern both with and without a gap, to avoid the influence on time of any additional noise messages. This ensured that the time statistics for each pattern were not skewed by any sequences with a gap. The result of this analysis was a set of interesting usage patterns that correspond to potential usability smells in the IDE, which we discuss in the next section.

5 Mining Results

In this section, we highlight a few interesting developer behaviors mined using our workflow and the ABB-Dev IDE usage dataset. [1]

5.1 Cluster 9: Usage of grep-like search tools

This cluster captures developers' typical use of the Find In Files dialog, which implements a string matching (grep-like) search over the entire open project in Visual Studio. Such a dialog is available in most modern IDEs.

5.1.1 Developers' Workflow

The patterns in this cluster show typical usage of Find In Files consisting of users opening the Find In Files dialog, entering a search query, and pressing the search button. This triggers the Find Results view, which displays a list of all of the lines that match the given query. The user can review the lines displayed in this window, which are unranked and unordered. Single-clicking a line in this window opens the file in an IDE editor as a temporary tab, while a double-click creates a permanent presence of the file in the editor tabs.

All of the patterns in this cluster involve behaviors based on the two views: Find in Files dialog where the query is entered, and Find Results where the results were examined. Other commands such as editing (indicative of search success) were present in some of the mined patterns, as well as query reformulations (indicative of search failure) - re-visiting the Find in Files dialog after clicking on a result in the Find Results view. Each of the patterns was observed across at least 30 developers.

5.1.2 Potential Usage Smell

One behavior that stands out when examining this cluster is that developers commonly review several results, spending considerable time, in each search session. Developers viewed as many as 8 consecutive search results before either finding the desired place in the code base or, more likely, abandoning the search. Prior lab studies support this finding, indicating that many searches of this type fail [19]. There were 65 instances in our input ABB-Dev dataset of developers following a search by viewing at least 8 search results with a median time of 50 seconds for this activity.

Analysis of the time that individuals spent focused on each clicked result indicates that the time between result clicks is quite short, an average of 10 to 12 seconds. This provides further proof that most of the individual clicked results were not of value to the developers, for their current task.

Our analysis shows that patterns of multiple search result views are: 1) common among developers, 2) relatively time consuming, wasting a median time of almost a minute of developer time, and 3) likely to be the result of suboptimal retrieved search results, as the developers moved fairly quickly between each click.

1. The dataset of mined interaction patterns is available at: http://vcu-swim-lab.github.io/mining-vs

TABLE 1
Extracted clusters and characterization of the IDE usage behaviors they represent.

Cluster	Characterization	Percentage of Patterns
1	*debugging:* shorter length stepping sequences	19%
2	*navigation:* shorter length usage of ABB's custom call graph navigation tool	13%
3	*debugging:* medium length stepping sequences	13%
4	*debugging:* activities related to stopping at a breakpoint	10%
5	*editing:* repetitions of opening several files in the solution explorer and edit	8%
6	*editing:* long length solution explorer and editing interactions	8%
7	*navigation:* longer length usage of ABB's custom call graph navigation tool	5%
8	*debugging:* a mix of stepping and running to breakpoints	4%
9	*searching:* usage of Visual Studio's Find in Files tool	3%
10	*navigation:* medium length usage of ABB's custom call graph navigation tool	2%
11	*debugging:* a mix of activities related to starting to debug (e.g. setting breakpoints, starting the debugger)	2%
12	*debugging:* finishing debugging followed by setting breakpoints, editing, building and related views	2%
13	*editing:* combinations of editing, building and related views (e.g viewing the error list)	2%
14	*debugging:* finishing debugging followed by editing, debugging and other activities	1%
15	*debugging:* longer length stepping sequences	1%
16	*editing/debugging:* editing followed by starting to debug	1%
17	*editing/building:* general editing behavior (e.g. building, looking at errors, editing and recompiling)	1%
18	*navigation:* very long usage sequences of ABB's custom call graph navigation tool	1%
19	*editing/navigation:* editing and navigating using the next and previous IDE buttons	$< 1\%$
20	*searching:* find all reference and find symbol searches	$< 1\%$

5.2 Cluster 15: Stepping numerous times with the debugger

5.2.1 Developers' Workflow

When using Visual Studio to debug, after the debugger has stopped their program at a breakpoint, developers commonly proceed executing the program in the debugger using the STEPOVER and STEPINTO commands. Both commands execute a single statement of code, and while the STEPINTO command follows method invocations, the STEPOVER command remains at the same level of the call path. All of the patterns in Cluster 15 consist of sequences of this set of stepping commands.

5.2.2 Potential Usage Smell

The most surprising finding in examining this cluster is the sheer number of step commands that users tended to execute in succession. While we expected that users may step through several statements after encountering a breakpoint, this cluster contained 19 pattern instances with more than 50 debugging step commands. These were also frequent in the original ABB-Dev dataset; a pattern of 75 steps occurred uninterrupted 23 times in the dataset.

To detect potential usage smells, our main goal was to detect long sequences of repetitive STEPOVER or STEPINTO commands where subsequent commands occurred within a very short time frame. There were numerous instances of the commands being issued with one second or less between them. The reasoning is that, if the timespan between two commands is very short, then the developer could not have performed any substantial analytical task, so the developer was most likely just pressing buttons at a high pace for a prolonged time.

Users displayed a tendency to use only specific debugging commands, even though more efficient commands are available. Developers rapidly pressed a step command tens of times in a sequence, while use of Visual Studio debugging commands such as RUNTOCURSOR or conditional breakpoints could replace all or many of these individual commands.

Additional analysis of individual pattern occurrences in the original ABB-Dev datasets highlights a few related observations. One such observation is that there are large differences between how often certain debugging commands are used by the developers in our dataset. A relatively small set of debugging events and commands occur in almost all developers, i.e., seven debugging commands/events occur in 90% or more of all developers. These are the common set of debugging commands and related events, such as setting and stopping on breakpoints, STEPINTO, STEPOVER, and starting, stopping, and restarting the debugger. Most of the remaining debugging commands occur in a smaller range of developers ($30 - 80\%$ of all developers). These include the attach-to-process command, the local variable declarations window, and the call stack window. Most of these commands are of broad applicability and should be considered for recommendation to other developers. Another, very small share of debugging commands occurs in less than 30% of all developers. These are very domain specific, such as viewing the hexadecimal display, thread activity, etc., and perhaps are better distributed via an IDE extension.

We also investigated whether developers using a richer palette of debug commands exhibited less repetitive behavior by using linear regression relating each developer's number of distinct debug commands to the time they require to execute a fixed length sequence of stepping commands. More precisely, we measured the mean timespan $y(i)$ until 16 stepping commands occurred for each developer i. We obtained a mean value of y of 20.1s and a standard deviation of 14.3s, leaving on average less than 1.5s between two stepping commands. Let $n(i)$ be all debug commands executed by developer i. This variable is used to normalize among developers with different amounts of data contribution to the data set. The variable $x(i)$ is the number

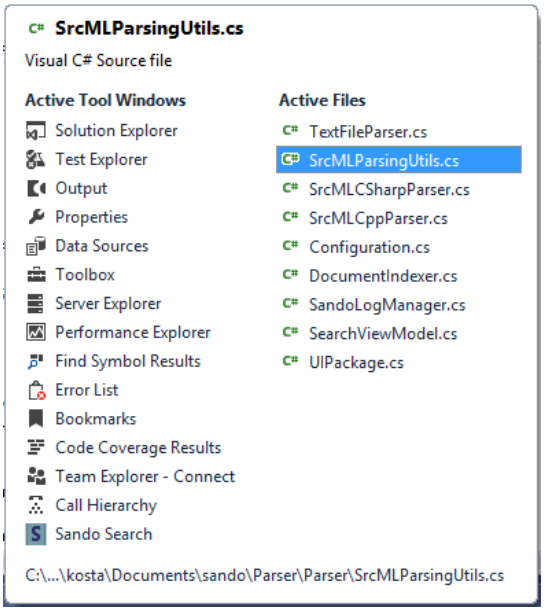

Fig. 4. A screenshot of the IDE NAVIGATOR DIALOG in Visual Studio.

of different commands used by developer i. We used the following model $y = a \cdot x/n + \epsilon$, where ϵ is assumed to be Gaussian noise. We obtained a p-value for a of 0.000027 and an adjusted R-squared value of 0.202.

Therefore, we conclude that developers using fewer distinct debugging commands behave more repetitively.

5.3 Cluster 19: Editing files in the working set

5.3.1 Developers' Workflow

The working set, consisting of all of the currently open tool windows and files in Visual Studio, is continually managed by the environment. The IDE NAVIGATOR DIALOG, shown in Figure 4, is commonly invoked via the Ctrl+Tab keystroke to navigate to the next file in the working set (Ctrl+Shift+Tab navigates to the previous file). The patterns in this cluster show developers opening and subsequently editing one or more files in their current working set using the Ctrl+Tab key and the IDE NAVIGATOR WINDOW.

The mined patterns in this cluster consist of repetitions of only two commands: editing and navigating via the IDE NAVIGATOR DIALOG. The behavior is consistent with making small changes across several of the files of the working set. Patterns ranged between editing one to five files.

5.3.2 Potential Usage Smell

The most notable finding when analyzing this cluster is that users often open unnecessary files when using the working set dialog, exhibited by consecutive invocations of the IDE NAVIGATOR DIALOG without editing. These unnecessary files are opened either in error, or as means of quickly reminding the developer of something, perhaps the name of a type or a method [19], which could have been preserved via some other means, such as active task context [20] or code bubbles [21]. For instance, in one particular pattern in this cluster, developers use the dialog to navigate to 5 separate files to ultimately find the file they wanted to edit. These navigations are relatively quick, spanning only a few seconds, but may still consume valuable developer cognitive energy.

We analyzed the patterns in this cluster to measure the success rate in reaching the final destination using the working set dialog, by identifying subpatterns where a dialog invocation preceded an edit. We considered one use of the dialog preceding an edit to be optimal, as the user found the file to edit, and any additional uses of the dialog preceding the edit to be unnecessary, as the user had to continue using the dialog to reach the relevant file. Using this approach, we found that opening files via the working set dialog was unnecessary 47% of the time in the extracted patterns. Ideally, the IDE and its working set support should enable developers to rapidly transition to edit files, without the need for several stops along the way.

Visual Studio working set support is not optimal, as many times files are opened without subsequent editing.

6 SURVEY STUDY

To assess and shed more light on the findings from our pattern mining approach to analyzing IDE interactions, we conducted a survey of 51 developers at ABB, Inc., to query their self-efficacy in using the IDE for three purposes: searching code with FIND IN FILES, navigating within their active file set, and stepping with the debugger. To solicit participation, the survey was advertised via e-mail to 661 developers at several of the company's product development sites across the globe. The 51 respondents had an average of 8.6 years (standard deviation of 5.9 years) of experience in using Visual Studio. The majority ($>$ 80%) of developers were using one of the three most recent versions of the environment (i.e., Visual Studio 2012, 2013 & 2015) for their daily work. On a scale of 1 to 5, where 1 denotes not usable and 5 denotes extremely usable, the developers found Visual Studio as more usable than not, giving it an average score of 3.9 (standard deviation of 0.9).

TABLE 2
Developer self-expressed frequencies of performing the three activities of interest.

ACTIVITY	NUMBER OF DEVELOPERS					
	Never	Monthly	Weekly	Daily	Hourly	>Hourly
FIND IN FILES *search*	7	1	9	19	8	7
stepping with debugger	7	6	3	10	15	10
active file navigation	30	1	6	4	7	3

6.1 Find in Files Search

We first asked the survey respondents a series of questions related to their use of the FIND IN FILES tool to perform project-wide (solution-wide in Visual Studio language) searches. The tool was used for this purpose by 44 of the 51 developers surveyed, with many developers using the tool frequently, or at least once daily, as shown in the first row of Table 2, which lists the number of developers that reported performing each activity of interest to our study at specific time frequencies.

The developers indicated a wide distribution of the number of results that they reviewed in each FIND IN FILES search session. Table 3 displays the distribution of respondents' answers, which indicates that there are 20% of developers (9 out of 44) that examine many (more than 5) results before locating a relevant block of code to their task.

TABLE 3
Responses to question: 'On average, how many search results do you need to examine before locating a useful program element?'.

NUMBER OF DEVELOPERS					
One Result	Two Results	Three Results	Four Results	Five Results	More than 5
7	14	7	5	2	9

We also asked the respondents to describe any difficulties they may have faced in using FIND IN FILES and received 10 such responses. While some respondents complained about the tool's speed, or its key bindings in different Visual Studio versions, several ($n = 8$) developers complained about the simplistic text-based UI that this tool uses to present its results, stating, for example:

> *The output is too cluttered. Results could be bit more organized in terms of folder/solution/projects [.It] needs to be better than notepad++ for an IDE.*
>
> *It is difficult to find anything in Find Results since the results and their line numbers are not put nicely into columns. The results look like a mess.*

6.2 Stepping with Debugger

For the part of the survey that inquired about debugger use, we intended to determine if long stepping chains were prevalent among developers and the reason for the absence of usage of conditional breakpoints and the RUN TO CURSOR command. Recall that in Table 2, the results shows that debugger use is indeed prevalent among developers, with a few exceptions. Interestingly, developers self-reported usage of many consecutive stepping commands, as shown in Table 4, which ranged relatively similar to what we observed in our interaction log analysis.

TABLE 4
Responses to question: 'On average, how many StepOver or StepInto commands do you execute in one sequence before you resume execution or end the debugging session?'

NUMBER OF DEVELOPERS					
0-10 Steps	10-20 Steps	20-30 Steps	30-40 Steps	40-50 Steps	More than 50
15	14	10	2	1	2

As indicated in Table 5, the usage of conditional breakpoints was relatively infrequent, though the feature was something that many developers had used in the past. On the other hand, there were a few more developers using the RUN TO CURSOR command frequently.

On an open ended question of how they used stepping and the RUN TO CURSOR to debug a program, a few of the respondents indicated that they rarely need the RUN TO CURSOR command, and that its functionality can be duplicated by adding another breakpoint, such as:

TABLE 5
Developer self-expressed frequencies in using conditional breakpoints and the RUN TO CURSOR debugger command.

CAPABILITY	NUMBER OF DEVELOPERS					
	Never	Monthly	Weekly	Daily	Hourly	>Hourly
conditional breakpoints	13	11	11	8	0	1
RUN TO CURSOR command	11	4	7	11	7	4

> *Break to the beginning of interesting segment. Hitting F10 to the end and checking values between. I use new break point and F5 rather than RunToCursor. [...]*

Another respondent hinted at the purpose for long stepping sessions, which is to comprehend what a block of code is doing at runtime, one statement at a time:

> *I use the step-commands to explore the code and what is happening. Until I get an "AHA" experience [...]*

For conditional breakpoints, on an open-ended question asking for any difficulties they may have faced in their use, the respondents raised two issues: 1) the slow-down that these breakpoints inflict on the debugger and 2) the fact that the conditions are difficult to express and are not error checked. Specifically, developers stated that:

> *The debugging is slowed down so much that I have stopped using it. It would be very useful for me, since I often want to see what happens at some specific state in a longer run.*
>
> *I often had the problem that the condition statement could not be evaluated. Then the debugger did not stop and I had to check the condition again.*

6.3 Active File Navigation

The survey asked developers about their usage of the active file navigation capability in Visual Studio, usually triggered via the Ctrl+Tab or Ctrl+Shift+Tab keystrokes, which navigate to the next and previous file in the list, respectively. The frequency of use is shown in the second row of Table 2, and is considerably less prevalent than the use of the search tool.

As before, we asked the developers who used active file navigation about how many files they typically visit before locating the relevant file. The distribution of responses, shown in Table 6, clearly shows that most developers do not experience finding the necessary file on the first try.

TABLE 6
Responses to question: 'On average, how many files do you navigate to before opening the file you are looking for?'.

NUMBER OF DEVELOPERS					
One File	Two Files	Three Files	Four Files	Five Files	More than 5
5	4	5	3	1	3

When asked the open-ended question about the difficulties they may have faced in using active file navigation in Visual Studio, most of the respondents gave no answer or stated they had experienced none. Only one respondent described a usability problem with the following statement:

> *The list is too long sometimes.*

7 Discussion and Implications

In general, the surveyed developers' self-reported tendencies confirmed the observations made via our pattern mining of IDE interactions, providing more confidence in the validity of this analytic technique for IDE interaction data. Surprisingly, many developers seemed aware of the high number of results that they examined during search with Find in Files, the number of files they navigated to before reaching their destination, and the multitude of consecutive steps they performed with the debugger. However, developers often did not necessarily consider these behaviors as inefficient or a nuisance, often answering "No difficulty" to our questions about problems with a specific feature. For instance, in active file navigation, a number of the surveyed developers reported no difficulty, when opening several unrelated files before locating the file they were looking for. For stepping with the debugger, while the surveyed developers seemed to always find value in numerous stepping actions, we believe that, in practice, a sequence of useful step commands where the state of a (set of) variable is observed by the developer is followed or preceded by a (potentially large) sequence of quick steps to get past irrelevant statements. We further believe that developer is usually unaware or does not consider these extra steps as additional work.

This unawareness of inefficient workflows is common in usability analysis of software, especially in situations where users are not actively performing the task that they were being surveyed about [22]. In these cases, software users commonly forget and ignore minor nuisances that are quickly resolved via an extra set of clicks within a short timespan. It is also difficult for the surveyed developers to imagine the possibility of an alternative interface design, where that specific interaction with the software would be made more efficient.

7.1 Find in Files Search

When searching using Find in Files, several of the developers in the survey agreed that the tool's UI had several imperfections in the way that it presents its results. One reason for this poor presentation is that the retrieved result set is unranked, which can result in an overwhelming amount of information when the query string frequently occurs in the code base. Several researchers have proposed within-IDE software search tools based on information retrieval technologies that, like web search tools, provide a ranked list of results making it easier for developers to reach relevant code quicker [23], [24]. Others have suggested improved user interfaces to allow quicker click-less review of the search results [25]. While such advances, among others, in code search have been discussed in the software maintenance research community for several years, most IDEs, in their default configurations, still rely on string matching approaches like Find in Files.

7.2 Stepping with Debugger

When debugging, the surveyed developers justified their low usage of conditional breakpoints as due to poor performance and the difficulty in catching errors in the conditions. We believe that the poor performance is accidental, and specific to the Visual Studio versions used by the developers in our studies, as evaluating a condition at a specific point in the code, unless it is performed within a tight loop, should present little difficulty to modern hardware. The difficulty in catching errors in the conditional can be remedied by logging the range of values of the constituent variables of the breakpoint's condition, enabling post-hoc analysis if the breakpoint is never triggered.

There was a significant number of developers that reported that they never used conditional breakpoints or the Run to Cursor command, likely because they were unaware of them. Our data analysis also showed that the distribution of debugging command use was skewed, and many useful commands are rarely used. Ideally, developers would know every command and use that command at the appropriate time. In reality, as shown by Murphy-Hill et al. [9], they often know and use only a small subset of the overall command set.

While it is unrealistic to expect developers to memorize every command, we feel that there is a minimum subset that should be known. We suggest two solutions. First, developers should be trained to use an IDE. Unfortunately, according to our experience, training on how to use this powerful tool is surprisingly sparse in most industrial or academic environments. Secondly, the commands themselves should be made more visible and intuitive to invoke. Visual Studio 2015 has already begun to do this. Conditional breakpoints, formerly an almost hidden feature in the UI is now highlighted through an icon that is shown as soon as a breakpoint is created.

7.3 Active File Navigation

As we examined active file navigation in Visual Studio, we noticed that it is easy to open the incorrect files due to the lack of additional information about each file on the IDE Navigator Dialog screen. For instance, if the working set dialog would provide a preview of a code snippet for a selected file, developers may be more likely to choose the correct file on the first attempt. Additionally, the dialog only shows the file name instead of the relevant method or field name, which makes the user map between files and program elements themselves. In the past several versions of Visual Studio, and especially the 2015 release, increased visual support, using the progressive disclosure interaction technique [26], has appeared in many of the IDE's tools (e.g., the call graph feature surfaced in the CodeLens extension).

8 Threats to Validity

Our IDE usage data analysis is potentially susceptible to several threats, both internal and external. One internal threat is that our ABB-Dev dataset only records certain IDE commands and events, while it does not, most notably, record the contents of the IDE editor or interactions with any other applications outside of the IDE. Therefore, it is possible that we missed or misunderstood certain regularly occurring developer behaviors. To mitigate this threat, two of the authors recreated the patterns that we focused on to ensure their reproducability and plausability. The developer

survey also lent support to the validity of some of the mined behaviors. The ABB-Dev dataset we used is also unbalanced, representing the interaction data of some developers more than others. This threat is mitigated by the substantial number of developers in the dataset and by the probabilistic nature of the sequential pattern mining we perform.

Another threat comes from the fact that we did not control for different versions of the Visual Studio IDE. While we know from internal ABB information that most of the data reflected the most recent 2 versions of the IDE (Visual Studio 2012 and 2013), even these could have differed, affecting our final conclusions. Finally, another threat comes from the choice of parameters in our analysis, including the length of patterns to filter and the weight of the measures in the final distance function. For each of these parameters, we examined samples of data or considered relevant statistics to choose reasonable values to detect interesting usage patterns and smells. However, more study is needed to verify that these parameter choices yield the strongest results.

Externally, we collected data from 196 developers for a period of less than 12 months from a single company. The results, drawn from a limited number of users and time length, may not be generalizable to other Visual Studio users for a longer period of time. Another external threat is that we investigated only the Visual Studio IDE and therefore the conclusions may not be applicable to other IDEs. However, a few of the most popular IDEs offer a very similar set of tools, so some of the mined usage smells may apply more broadly than Visual Studio.

9 Conclusions and Future Work

This paper presents a novel approach for mining sequences of actions from IDE usage data. Our approach uses scalable sequential pattern mining, coupled with pre-processing and post-processing steps intended to reveal interesting IDE usage scenarios from many developers. A selection of those scenarios is thoroughly analyzed, via expert opinion and a developer survey, to yield more understanding of those patterns, which span code search, file navigation and debugging. The result is a set of recommendations for Visual Studio improvement in each of these categories.

As future work, we plan to investigate different ways to tailor the mining approach to a more fixed set of behaviors (e.g., related to debugging). We will also explore approaches to changing some of the post-processing steps to include shorter sequences in the subsequent manual analysis.

Acknowledgments

We would like to thank Will Snipes for help in administering the study and for collecting the usage dataset. We also thank the numerous developers at ABB, Inc. that contributed their IDE usage data and answered our survey.

References

[1] S. Negara, M. Codoban, D. Dig, and R. E. Johnson, "Mining Fine-grained Code Changes to Detect Unknown Change Patterns," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 803–813. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568317

[2] M. Vakilian and R. E. Johnson, "Alternate refactoring paths reveal usability problems." ACM Press, 2014, pp. 1106–1116. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2568225.2568282

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

[4] E. Murphy-Hill, C. Parnin, and A. P. Black, "How We Refactor, and How We Know It," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, Jan. 2012. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6112738

[5] M. Spiliopoulou, "Web usage mining for Web site evaluation," *Communications of the ACM*, vol. 43, no. 8, pp. 127–134, Aug. 2000. [Online]. Available: http://portal.acm.org/citation.cfm?doid=345124.345167

[6] The Eclipse Foundation - Filtered UDC Data, "http://archive.eclipse.org/projects/usagedata," accessed Feb 4th, 2016.

[7] Codealike: Powerful Metrics for High-Performance Developers, "https://codealike.com," accessed Feb 4th, 2016.

[8] Lijie Zou, M. Godfrey, and A. Hassan, "Detecting Interaction Coupling from Task Interaction Histories." IEEE, Jun. 2007, pp. 135–144. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4268248

[9] E. Murphy-Hill, R. Jiresal, and G. C. Murphy, "Improving software developers' fluency by recommending development environment commands." ACM Press, 2012, p. 1. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2393596.2393645

[10] W. Snipes, V. Augustine, A. R. Nair, and E. M. Hill, "Towards recognizing and rewarding efficient developer work patterns," in *Proceedings of the 2013 International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2013, pp. 1277–1280. [Online]. Available: http://portal.acm.org/citation.cfm?id=2486983

[11] W. Snipes, A. R. Nair, and E. Murphy-Hill, "Experiences gamifying developer adoption of practices and tools," in *Proceedings of the 36th International Conference on Software Engineering*. New York, NY, USA: ACM, 2014, pp. 105–114. [Online]. Available: http://doi.acm.org/10.1145/2591062.2591171

[12] S. Amann, S. Proksch, S. Nadi, and M. Mezini, "A study of visual studio usage in practice," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER '16)*, 2016.

[13] Y. Yoon and B. A. Myers, "Capturing and analyzing low-level events from the code editor," in *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*, ser. PLATEAU '11. New York, NY, USA: ACM, 2011, pp. 25–30. [Online]. Available: http://doi.acm.org/10.1145/2089155.2089163

[14] G. Khodabandelou, C. Hug, R. Deneckre, and C. Salinesi, "Unsupervised discovery of intentional process models from event logs." ACM Press, 2014, pp. 282–291. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2597073.2597101

[15] A. Vargas, H. Weffers, and H. V. da Rocha, "A method for remote and semi-automatic usability evaluation of web-based applications through users behavior analysis." ACM Press, 2010, pp. 1–5. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1931344.1931363

[16] D. Perera, J. Kay, I. Koprinska, K. Yacef, and O. Zaiane, "Clustering and Sequential Pattern Mining of Online Collaborative Learning Data," *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 6, pp. 759–772, Jun. 2009. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4564464

[17] P. Fournier-Viger, A. Gomariz, T. Gueniche, A. Soltani, C.-W. Wu, and V. S. Tseng, "Spmf: A java open-source pattern mining library," *Journal of Machine Learning Research*, vol. 15, pp. 3389–3393, 2014. [Online]. Available: http://jmlr.org/papers/v15/fournierviger14a.html

[18] I. Miliaraki, K. Berberich, R. Gemulla, and S. Zoupanos, "Mind the gap: Large-scale frequent sequence mining," in *ACM SIGMOD International Conference on Management of Data (SIGMOD 2013)*, Association for Computing Machinery (ACM). New York, USA: ACM, June 2013.

[19] 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 Trans. on Soft. Eng.*, vol. 32, no. 12, pp. 971–987, 2006.

[20] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2006, pp. 1–11.
[21] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola Jr, "Code bubbles: Rethinking the user interface paradigm of integrated development environments," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, 2010, pp. 455–464.
[22] J. Nielsen and J. Levy, "Measuring usability: Preference vs. performance," *Commun. ACM*, vol. 37, no. 4, pp. 66–75, Apr. 1994. [Online]. Available: http://doi.acm.org/10.1145/175276.175282
[23] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Soft. Maint. and Evolution: Research and Practice*, 2011.
[24] D. Shepherd, K. Damevski, B. Ropski, and T. Fritz, "Sando: an extensible local code search framework," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE, 2012, pp. 15:1–15:2.
[25] J. Wang, X. Peng, Z. Xing, and W. Zhao, "Improving feature location practice with multi-faceted interactive exploration," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 762–771. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486888
[26] J. Nielsen, "Progressive disclosure," *Jakob Nielsen's Alertbox*, 2006. [Online]. Available: http://www.nngroup.com/articles/progressive-disclosure/

Kostadin Damevski is an assistant professor at the Deparment of Computer Science at Virginia Commonwealth University. Prior to that he was a faculty member at Virginia State University and a postdoctoral research assistant at the Scientific Computing and Imaging institute at the University of Utah. His research focuses on software maintenance and empirical software engineering, applied to a variety of domains, ranging from industrial software systems to high-performance computing. Dr. Damevski received a Ph.D. in computer science from the University of Utah in Salt Lake City.

David C. Shepherd is a Senior Principal Scientist with ABB Corporate Research where he leads a group focused on improving developer productivity and increasing software quality. His background, including becoming employee number nine at a successful software tools spinoff and working extensively on popular open source projects, has focused his research on bridging the gap between academic ideas and viable industrial tools. His main research interests to date have centered on software tools that improve developers search and navigation behavior.

Johannes Schneider completed his PhD at ETH Zurich in 2011. He was a PostDoc at IBMs Zurich Research Laboratory in the Information Analytics Group. Now, he is a Scientist at ABB Corporate Research, Switzerland. In October 2016 he is going to join the University of Liechtenstein as Assistant Professor in Data Science.

Lori Pollock is a Professor in Computer and Information Sciences at the University of Delaware and ACM Distinguished Scientist. Her research focuses on software artifact analyses for easing software maintenance, testing, and developing energy-efficient software, code optimization, and computer science education. She leads a team to integrate CS into K-12 through teacher professional development in the CS10K national efforts. She was awarded the ACM SIGSOFT Influential Educator award 2016 and University of Delaware's Excellence in Teaching Award, E.A. Trabant Award for Women's Equity in 2004. She serves on the Executive Board of the Computing Research of Women in Computing (CRA-W), which was honored with the National Science Board's 2005 Public Service Award to an organization for increasing the public understanding of science or engineering.