A Study of Visual Studio Usage in Practice

Sven Amann, Sebastian Proksch, Sarah Nadi, and Mira Mezini
Software Technology Group
Technische Universität Darmstadt
Email: {amann,proksch,nadi,mezini}@cs.tu-darmstadt.de

***Abstract*—Integrated Development Environments (IDEs) provide a convenient standalone solution that supports developers during various phases of software development. In order to provide better support for developers within such IDEs, we need to understand how much time developers spend using various parts of a given IDE and how often they use available assistance tools. To infer useful conclusions, such information should be gathered for different types of IDEs for different languages.**

In this paper, we instrument the previously unexplored Visual Studio IDE and track the interactions of developers at an industry partner's software-development department. As a result, we capture interactions for more than 6300 hours of work time, from between 27 and 84 professional C# developers. Our work reports how much time professional developers spend on activities such as code editing and execution or navigation, as well as how often they use assistance tools provided by the IDE. We compare our findings to those of prior studies involving other IDEs and discuss the implications of the commonalities and differences for research on (integrated) developer-assistance tools.

I. Introduction

Integrated Development Environments (IDEs) are very popular among software developers since they provide support for many of their daily development or maintenance tasks. Modern IDEs provide integrated debuggers, automated refactorings, assistance tools like code completion, and even integrated version control. In order to further assist developers and improve IDEs, we need to understand how developers typically spend their time in an IDE and which tools they actually use.

Previous studies investigated developers' use of IDEs [1]–[3]. Those studies looked at different IDEs (Eclipse and Pharo) and had a mix of academic, professional, and free-time developers. To provide better understanding of how IDEs are used, more large-scale studies with various settings and different IDEs are needed. In this paper, we provide such a study that examines how developers use an IDE in an industrial setting. We focus on an IDE different from those used in previous studies, namely Microsoft's Visual Studio IDE (VS).

We perform a case study of how industrial C# developers use Visual Studio. We developed FeedBag, a tool that anonymously captures developers' IDE interactions. We deploy our tool at an industry partner's software-development department, in which more than 400 developers write software in C#. We collect more than 3.5 million interaction events over a total of 6,300 work hours. We transform the captured events into high-level activities such as development, navigation, IDE configuration, and project management in order to identify how much time developers spend on each activity. Additionally, we analyze their usage of the tools offered by the IDE. Specifically, the data we collect allows us to answer:

RQ1 How do developer spend their in-IDE time?

RQ2 Which IDE tools do developers use and how frequently?

By comparing the answers to RQ1 and RQ2 to those from previous studies on other IDEs, we additionally answer:

RQ3 How does IDE usage differ between IDEs?

From our observations in answering these research questions, we come up with ideas for next-generation IDEs.

To the best of our knowledge, we present the first large-scale study with professional C# developers using Visual Studio. Our analysis shows that our participants spend almost 30% of their in-IDE time on code editing and execution and another 22% navigating documents and source code. They spend little time on other activities, like IDE configuration or project management. Also developers are often inactive for short intervals (< 5 min) while using Visual Studio. We find that the code completion is by far the most frequently used assistance-tool, followed by the build system, the debugger, code search and navigation tools, the quick-fix, and version control. In contrast, unit-testing tools are rarely used. We compare our findings to other IDE-user studies and infer a set of actionable outcomes to drive future research on IDEs. We provide an online artifact page with supplementary information [4].

In summary, we make the following contributions:

1) An open-source tool, FeedBag, designed to anonymously capture developers' interactions with Visual Studio.
2) A case study of how professional C# developers use Visual Studio, involving more than 800 developer days.
3) Comparison of our findings to previous IDE user studies.
4) A discussion of opportunities to advance the research in IDEs and developer-assistance tools.

II. Related Work

Throughout the paper, we compare our setup and results to relevant studies that have also looked at developers' activities and tool usage. In this section, we introduce such related studies and compare their overall goals to ours.

General Work Habits: Perry et al. [5] conducted one of the earlier investigations into how developers spend their time, using time cards and an observation study. While we also want to understand how developers spend their time, we focus on how they spend their time *within* the IDE. Since we do not physically observe developers, we cannot (and do not aim to) come to conclusions about the activities they conduct outside the IDE (e.g., sending emails or talking to co-workers).

More recent studies include that by González et al. [6] who looked at how developers multi-task. They introduced the notion of working spheres. LaToza et al. [7] studied developers' typical tools, activities, and practices based on two surveys and eleven interviews. Their main goal was to investigate how developers understand code and keep track of the information they need. Singer et al. [8] also studied software practices of software engineers through questionnaires and developer shadowing. They mainly focused on activity switches and not on time duration of any of these activities. While all these studies aim to understand how developers get their tasks done, and often what they spend their time on, none of them instrumented the developer's working environment to precisely capture the interactions taking place.

IDE Usage: The studies closest to ours are Murphy et al.'s study [1] on the usage of the Eclipse IDE for Java and Minelli et al.'s study [2] on the usage of the Pharo IDE for Smalltalk. Both groups of researchers instrumented their respective IDEs to track developers' activities. Our work expands this space of knowledge by a study on the usage of Visual Studio for C#. This provides an interesting point of comparison between the results. A key difference between our work and both studies is that we focus on professional developers from industry and do not use any open-source or student developers as participants. A large-scale study by Beller et al. [3] reports on developers' usage of Eclipse with respect to unit testing. They analyzed how much time developers spent on editing test and production code. We also compare our findings to theirs, where applicable. Kersten and Murphy [9] also study IDE usage, but focus on how their proposed tool, Mylyn, affects developers. Snipes et al. [10] study how gamification impacts developers' usage of Visual Studio. Similar to FEEDBAG, their tool, Blaze, logs IDE interactions. Additionally, it provides developers with feedback about their IDE usage. We explicitly avoid this to capture the status quo of how developers use Visual Studio. Snipes et al. [11] recently presented a practical guide for IDE-usage studies. We support the value of such guides, although, unfortunately, as it was published only after we conducted our study.

Assistance Tools: Other studies specifically look at how developers use static analysis tools. For example, Johnson et al. [12] investigated developers' perception of static analysis tools and the reasons they might avoid using them. Similarly, Ayewah et al. [13] used online surveys and questionnaires to understand how developers use FindBugs. Both studies included industrial participants. While we use different research methodologies and do not focus on static analysis for bug detection, our findings about tool usage align with the findings of both studies. However, our work does not investigate why developers use certain tools versus others.

III. FEEDBAG: CAPTURING DEVELOPER INTERACTIONS

To collect information about interactions of developers with Visual Studio, we instrument the IDE. We do this through *ReSharper* [14] (R#), a widely used Visual Studio extension. R# is designed as an extensible platform. It provides access to a semantic model of the source code under edit, including, for example, a type-resolved abstract syntax tree.

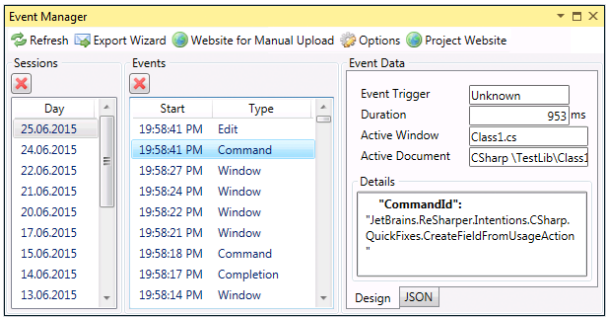

Figure 1: Screenshot of the Event Manager

A. FEEDBAG *in a Nutshell*

We implement FEEDBAG, a R# plugin that generates interaction events and allows users to manage them. FEEDBAG can be installed from R#'s official extension repository. Although targeted for our industry partner, FEEDBAG is general enough for any Visual Studio user and is publicly available [4], [15].

B. *Event Generator*

The *Event Generator* hooks into every UI control, executable command, window, and editor in Visual Studio. Whenever the user interacts with one of these elements, the generator creates an event and stores it on the user's machine. This process does not interfere with IDE functionality. The generator captures the following information:

Timing: Each event contains the timestamp of its occurrence and, if an event takes some time to finish (e.g., when a build of a project is started), its duration.

Location: Each event contains the active window and –if available– the active document at the point of its generation. This allows us to distinguish location-sensitive interactions.

Session: Each event contains a random identifier (session Id) that is shared by all events generated by the same FEEDBAG instance on the same day. This allows us to analyze interactions at the granularity of developer days without relating events to individual developers.

Type: Each event has an interaction type and captures specific information. We distinguish the following four types:

Build: When the user runs a build, we capture the target projects, the build duration, and whether it succeeds.

Commands: When the user invokes a command, like a native action (e.g., save file) or an action offered by an extension (e.g., R#'s quick fixes), we capture that command's unique Id.

Documents: When the user opens, edits, saves, or closes a document, e.g. a source file, we capture which of these actions she performed and the document's name. To reduce the number of edit events, we aggregate edits to the same document if less than two seconds pass between them. This results in one aggregated event whose duration is the time that elapsed between the first and the last aggregated edit.

Windows: When the user opens, closes, or switches between windows, we capture the window's title and which of these

actions she performed. We capture the same information for Visual Studio's main window, to identify when the user started, left from, returned to, or closed the IDE.

C. Event Manager

We want FEEDBAG to be comprehensible and controllable for the user. Therefore, the *Event Manager* allows her to manage stored events via the UI shown by Figure 1. She may delete events, but cannot alter them.

The generated events are stored locally on the developer's machine. The developer is regularly informed about how many events were collected and asked to upload them to our server. Once she decides to provide the data, the Event Manager bundles all local events and sends that bundle to our server. Subsequently, it deletes the local copy of the events.

IV. INVESTIGATING IDE USAGE

We use the data collected by FEEDBAG to investigate developers' activities as well as their assistance-tool usage on a typical day. To prepare both analyses, we split the events in our dataset by developer day. From the resulting developer days, we identify both activity intervals and tool usages.

A. Identifying Developer Days

Interaction data of developers is sensitive data. We intentionally designed FEEDBAG not to capture any information that identifies individual developers. Instead, we use a session Id to identify events created by one developer during one calendar day. Then, we group sessions that were sent to our server in a single upload to identify *developers*. We find that developers regularly work past midnight, but never between 2 A.M. and 5 A.M. Therefore, we do all subsequent analysis on *developer days*, which span from the first event after 3 A.M. to the end of last event before 3 A.M. the next calendar day. We refer to the length of a developer day as the *work time*.

Approximating Participants: For reporting purposes, we estimate the number of participants from the data FEEDBAG generates. Since we do not track individual participants, we cannot report a definitive number, but determine an upper and a lower bound instead. When a participant uploads multiple sessions in one bundle, we know they originate from the same developer. When she uploads them in separate bundles, we have to assume they belong to different developers. Hence, when we identify developers based on simultaneously uploaded sessions, we might count multiple developers that actually correspond to the same participant. Therefore, the number of developers based on this strategy is an upper bound to the number of participants. To determine a lower bound, we merge as many identified developers as possible, as long as no events from their corresponding developer days overlap. The number of remaining developers presents a lower bound to the number of participants in our study.

B. Identifying Activity Intervals

We derive *activities* from the low-level events captured by FEEDBAG (e.g. mouse clicks and key presses that occur within the IDE). To analyze how developers spend their time, we want an unambiguous mapping from events to activities. The first and second author created such a taxonomy of activities following an open-coding approach. First, they each separately created a mapping from events to at least one activity they defined adhocly. During the process, they considered the event category, the target window for window switching, and the command Id for command executions. Second, they merged their mappings by joining all activities either of them assigned to each event. Third, they removed duplicated activities, unifying activity names where necessary. Fourth, they found a single, more abstract activity for each event that was mapped to multiple activities and assigned this new activity also to every event previously assigned to either one of the more specific activities. They iterated through the last step until each event was mapped to a single activity. The complete mapping scheme can be found on our artifact page. At the end of this process we had the following taxonomy of activities:

Code Editing & Execution: Includes editing of documents; using automated refactorings, code generation, or find & replace; adding, renaming, and removing files, projects, or solutions; building projects; and using the debugger.

Navigation: Includes opening and closing documents; using searches, both textual and code specific (e.g., find usages); using arrow or position keys; and using bookmarks.

IDE Configuration: Includes opening, closing, or moving around windows; changing window settings, selecting filters in view, or configuring columns shown in a table; and opening dialogs (e.g., IDE options or file properties).

Project Management: Includes managing issues, tasks, or requirements; and working with source control.

Other: Includes all remaining interactions, which we could not group into any larger meaningful activities (e.g., activating FEEDBAG-related windows, the Visual Studio command shell, the Tips & Tricks window, or the start page).

Inactivity: Denotes phases where the IDE has focus, but no interaction occurs. We do not consider mouse movement as an interaction. While it may indicate activity, such as reading code, previous work has shown that developers perform isolated mouse movements for less than 4% of their time [2]. Therefore, we expect the impact on the time budget to be small. We also note that our notion of activity and inactivity does not equal nor necessarily correlate with developers working or not working. Since we only capture interaction within the IDE, we cannot account for other work activities, such as meetings, phone calls, discussion, and the like. It is neither our claim nor our goal to report on how much developers work.

We implemented a framework that groups events by developer day, orders them by timestamp, and iterates over them. The algorithm for computing activity intervals from the resulting event streams is shown in Figure 2. For each event, we either create a new interval, if none is running (Line 4), stop the current interval, if the user left the IDE (Line 7), or do both, if the current activity has changed (Lines 9 and 10). Intervals time out, when no event occurs for some time. Each

```
1  onEvent(Event e):
2    cancel timeout
3
4    if (no currentInterval)
5      openInterval at start(e) with activity(e)
6    else if (activity(e) == "Left IDE")
7      close currentInterval at end(e)
8    else if (activity(e) != currentActivity)
9      close currentInterval at end(e)
10     openInterval at start(e) with activity(e)
11
12   timeout 15 seconds after end(e)
13
14 onTimeout(Timestamp t):
15   close currentInterval at t
16   startInterval at t with "Inactivity"
17
18 onEndOfDeveloperDay:
19   if (exists currentInterval)
20     remove currentInterval
```

Figure 2: Activity-Interval Detection

new event cancels the active timeout (Line 2) and sets a new one, which ends 15 seconds after the event ends (Line 12). If this timeout is reached, we end the current interval and start an inactivity interval (Lines 15 and 16). If the developer leaves the IDE open overnight, the developer day ends with a running Inactivity interval. In this case, we delete that interval (Line 20), which leads to the same activity stream we would get if she had closed the IDE directly after her last activity.

Finally, to compute a developer's time budget, we sum the durations of intervals per activity and for Inactivity.

C. Identifying Assistance-Tool Usage

In Visual Studio, assistance tools are used via commands invocation. Our interaction events tell us which commands developer use. Before we analyze tool usages, we reduce noise in the respective set of command Ids:

Simple Keystrokes: We remove editing keystrokes, such as the arrow keys, enter, backspace, or delete, because -similar to character strokes during typing, which are also excluded- they do not represent special command behavior.

Equivalents: We find that commands are not consistently reused throughout Visual Studio, e.g., selecting `Close` from the file menu has a different command Id than closing the document using a key binding or via the `x`-button on the top of the document. Fortunately, such commands often result in a specific sequence of other events. For example, when closing a document, we also see a document-close event and a window-close event. We derive a mapping of equivalent commands by analyzing sequences of commands that follow one another within up to 100ms and group all such micro-sequences with a common suffix. The mapping was manually reviewed and then used to reduce equivalent commands Ids to a single one. Murphy et al. [1] encountered similar problems when analyzing command usage in Eclipse. They also manually created a mapping between command Ids to merge equivalents. We support their plea to IDE developers to consistently use command identifiers to simplify analytical work.

```
1  onStartOfDeveloperDay:
2    lastTool = "None"
3
4  onEvent(CommandEvent ce):
5    if (lastTool != tool(ce)
6          && tool(ce) != "IDE Core")
7      increment usages of tool(ce)
8      lastTool = tool(ce)
```

Figure 3: Assistance-Tool-Usage Detection

Duplicates: Some interactions trigger multiple commands, because extensions like R# install own commands for the same interaction. Both the original Visual Studio command and the R#-equivalent appear in our statistics. We mine commands that co-occur within 100ms, assuming that it is unlikely for a developer to actually invoke multiple commands in such a short time. From the results, we manually create a mapping of command pairs and use it to filter duplicates.

Then we identify tool usage frequencies in three steps:

First, we manually create an exhaustive mapping from commands to tools. The mapping is based on our understanding of the command's functionality, usually obvious from its name. There are two special tool categories in the mapping: `IDE core` and `Misc`. The first includes commands that constitute IDE core functionality, such as copy and paste or file open, close, and save. The second includes commands that we could not identify, because the command gave no clue as to which tool it supports. The mapping is available on our artifact page.

Second, we traverse the event streams of our developer days to compute tool usages as shown in Figure 3. At the beginning of each developer day, we reset the last tool used (Line 2). We then process all command events and increment a tool count the first time the developer switches to it from a previous tool (Line 7). We ignore all interactions with `IDE Core`.

Third, we rank the assistance tools by the average number of usages per developer day. Note that this ranking favors tools whose usages span longer time periods and encompass multiple invocations of related commands. For example, while using code completion happens almost instantaneous with a single command, using the debugger often takes several minutes and various commands. This increases the chances of the developer using other tools during a debugging session. If the developer starts the debugger, steps a few times, then uses a search to look up some code, and afterwards continues to step, we would count two usages of the debugger (even though it is technically the same debugging session). We believe that this calculation methodology reflects the actual impact a tool has on the developer day. Therefore, we accept this imprecision.

V. Industrial Case Study

In this section, we describe our industrial case study. We specifically describe how we used FeedBaG to collect data about professional developers' use of Visual Studio. For privacy reasons, we cannot name our partner so we will refer to them as *CompanyX* throughout the rest of the paper.

A. Company Background

CompanyX develops tax and accounting-related software as well as in-house software for 50 years. It employs more than 1,600 developers, out of which more than 400 write programs in C# and use R#. Development projects span from small training examples to core-business applications.

B. Incremental Rollout

To make sure that FEEDBAG works properly in CompanyX's settings, we deployed it in multiple steps.

1) Customized Development: We developed the tool in close collaboration with a single developer from CompanyX. Therefore, we got early feedback and ensured that technical requirements are met, according to CompanyX's environment.

2) Pilot Study: When we deemed the tool production-ready, two volunteers from CompanyX installed FEEDBAG as pilot users. Our goal was to ensure correct functionality of FEEDBAG in many different use cases and also to convince the management that the study does not interfere with the regular tasks of the developers. The pilot phase lasted about 2 months, during which we identified and fixed minor bugs.

3) Company-wide Study: The successful pilot study confirmed that FEEDBAG was production-ready. The management permitted a large-scale rollout. We prepared extensive documentation to inform developers about the project and to motivate why we track their IDE interactions. We also released FEEDBAG as open-source software so that developers can check themselves that no personal information is stored. Additionally, we provided the Event Manager (see Section III-C) to give them full control over their data.

We then sent a request for participation to the 400 R# users at CompanyX. In this email, we introduced the project and provided instructions. To encourage participation, we promised them a method-call recommender for their code completion, similar to our previous work on Java [16], but specifically trained for their in-house frameworks. We made clear that we would provide the recommender to all developers, independent of a contribution to our study. Participants did not receive any other benefits. All participants installed FEEDBAG voluntarily and otherwise followed their regular work schedule. They were not assigned to any special tasks for the study.

During the study, we continually posted project updates and intermediate results on the mailing list. Additionally, we attended community events of developer groups at CompanyX (e.g., Clean Code Community and Software Craftsmanship Community) to introduce FEEDBAG.

C. Collected Event Dataset

We tracked IDE interactions of developers for about six months, from mid January to mid July 2015. For the first two months, we had only our pilot users. In the middle of March, we started recruiting participants, which quickly raised their numbers to between 27 and 84 (see Section IV-A). The resulting dataset encompasses 3,505,858 events, amounting to over 6,355 hours of work time. From this time, participants spent about 2,103 hours inside Visual Studio. On average, they worked 7 hours and 12 minutes per day. This suggests that multiple full-time developers participated in our study.

Since we cannot publish our industrial dataset, we started to collect a second dataset from students and open source developers that is public and reusable by other researchers. However, since this dataset is still small, the statistics reported in this paper are only based on the industrial data.

Table I: Statistics on Identified Developers, Sorted by Active Time

Dev.	# Events	# Days	Work Time	In-IDE Time	Active Time	Avg. Daily Work Time
			in hours			
D01	198,272	40	344:56	159:40	89:45	8:37
D02	311,293	38	286:18	117:37	75:47	7:32
D03	119,323	17	169:19	70:26	63:08	9:57
D04	150,542	23	196:21	83:06	59:40	8:32
D05	140,438	23	203:38	81:04	55:58	8:51
D06	67,444	18	151:12	74:03	52:44	8:24
D07	294,388	17	131:26	62:21	40:41	7:43
D08	102,696	12	121:11	78:24	40:32	10:05
D09	85,839	20	130:34	51:44	35:51	6:31
D10	153,369	23	181:14	52:52	34:37	7:52
D11	99,351	43	297:32	56:24	34:22	6:55
D12	55,903	32	241:10	47:47	33:40	7:32
...						
D82	345	1	2:16	0:14	0:10	2:16
D83	434	1	16:28	0:20	0:08	16:28
D84	41	2	2:33	0:08	0:02	1:16
Overall	3,505,858	881	6355:15	2103:25	1302:09	7:12

D. Statistics on Identified Developers

While we base our analysis on developer days to be precise, we show statistics about the identified developers in Table I. The first column shows an identifier for the developer. The second and third columns show the number of events and days from her. The fourth column shows her total work time. The fifth column shows the time spent in Visual Studio, i.e., the work time minus the time any application other than Visual Studio had the application focus. The sixth column shows the active interaction time, i.e., the in-IDE time minus the time in which we record no interactions with the IDE. The last column shows the average work time per day. The full table is available on our artifact page.

E. Statistics on Tool Commands

From all developer days, we recorded usages of 2,493 different commands. After noise reduction, 1,346 unique commands remain. In a manual inspection, we found no further equivalents or duplicates.

VI. DEVELOPERS' USE OF VISUAL STUDIO

We analyze two aspects of developers' use of Visual Studio: their *time budget* (i.e., how much time they spend on each activity) and their *tool usage* (i.e., frequency of use of IDE tools). We report on both aspects and also compare to similar studies performed on different IDEs or in non-industrial settings.

For all subsequent consideration, we exclude developer days with less than 30 minutes of activity time. This leaves us with 588 developer days, accumulating 5021 hours of work time and 1255 hours of active IDE interaction. For these

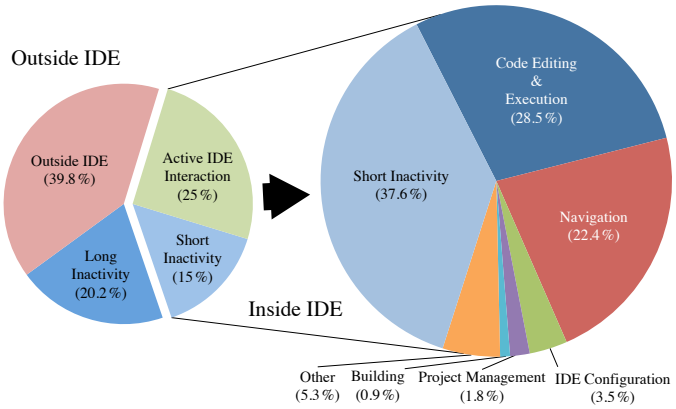

Figure 4: A Developer's Time Budget

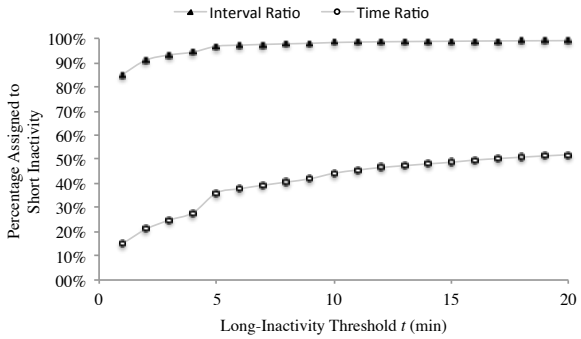

Figure 5: The Effect of the Threshold t on the Split of Inactivity

developer days, the average daily work time is 8 hours and 32 minutes. Figure 4 shows the average time budget of all such developer days. Figure 4a shows the overall time budget, while Figure 4b zooms in on the in-IDE time. Subsequently, when we talk about *a developer*, we refer to the average developer represented by this time budget.

The remainder of this section proceeds as follows: First, we present a high-level overview of a developer's working day. Second, we discuss her activities in the IDE. Third, we present a detailed analysis of her IDE-tool usage.

A. A Developer's Work Day

Outside the IDE: Based on Figure 4a, a developer spends 39.8% (3h24m) of her daily work time with the application focus away from the IDE. Since FeedBag does not track interactions during this time, we cannot differentiate between times where she does not interact with her machine from those in which she uses other applications (e.g., a browser or email client). A study by LaToza et al. [7] reports that developers indeed use many applications besides their IDE.

O1: Developers spend a considerable amount of time outside the IDE, potentially using external tools for their work.

Inactivity: For a total of 35.2% (3h) of her day, a developer is within the IDE, but does not interact with it (sum of long inactivity and short inactivity in Figure 4a). We see two kinds of (causes for) inactivities:

1) The developer is interrupted in her work, e.g., by a phone call, a colleague, a meeting, or the lunch break. Note that the application focus does not change, if the workstation is locked or the screensaver activates, which is why times during which the developer is actual away may appear as in-IDE inactivity in our statistics. Developers face many such (unplanned) interruptions of their work [5], [17].
2) The developer stops interaction to, e.g., read some code, think, or take a sip of her coffee. In such cases, we record inactivity, but the developer incurs no context switch.

Note that at the time of this study, FeedBag did not track mouse movements and scrolling. Such information might help to partially separate code reading and maybe even thinking (assuming occasional mouse movement) from actual inactivity. However, Minelli at al. [2] report that isolated mouse movements make up for only 3.5% of a developer's in-IDE time, indicating that this does affect our time budget much.

O2: Developers spent a third of their in-IDE time not interacting with the IDE.

Since we cannot directly determine the kind of inactivity from our events, we heuristically separate inactivity intervals by their duration, reasoning that longer inactivity is more likely to be an actual interruption of a developer's work in the IDE. We separate *short inactivity* from *long inactivity* using a threshold t. Figure 5 shows the effect of t on the percentage of inactivities and the total inactivity time that would be considered as short inactivity. We see a relatively small number of inactivities that last for more than a couple of minutes. In fact, 85% of all inactivities are shorter than 1 minute, while 97% are shorter than 5 minutes. At the same time, we see that the inactivities below 1 minute make up for only 15% of the total inactivity time and even those below 5 minutes for only 36%. In both figures, we observe the start of a saturation effect at about $t = 5$ minutes. This means developers who are inactive for 5 minutes are more likely to be inactive for even longer. With this threshold, we find a strong negative correlation between long-inactivity time and away time (Pearson's $r = -0.71, p = .01$), supporting our theory that long inactivity indicates that the developer left her machine. Therefore, we subsequently consider all inactivities of more than 5 minutes as long inactivities and exclude them from the in-IDE time. Interestingly, Minelli et al. [2] chose the same threshold to determine when a developer became idle. However, they do not report how much long-inactivity time they observe or how they determined their threshold.

During her day, a developer has 153 short inactivities, with an average duration of about 30 seconds, and 4 long inactivities, with an average duration of about 23:18 minutes.

O3: Developers have many very short inactivities and very few long, break-like inactivities.

Active IDE Interaction: Removing the time spent outside the IDE and both long and short inactivities leaves 25% (2h08m) of *active-interaction time* within the IDE. Kersten and Murphy [9] report a similar daily interaction time of about 2 hours from a diary study with six senior IBM developers. For our participants, this time is fragmented into 158 continuous-interaction periods (i.e., periods with no inactivity whatsoever) with an average duration of about 49 seconds.

B. What a Developer Does in Visual Studio

Figure 4b zooms in on how a developer spends her in-IDE time. Subsequent statistics consider this time only.

Short Inactivity: About 38% of a developer's in-IDE time consists of short inactivities. Recall that short inactivities are those with a duration of up to 5 minutes. The average duration of such inactivities is 30 seconds. A study by Minelli et al. [2] reports that 66% of their participants' in-IDE time is short inactivities. Possible causes for this big difference between their and our results include the work settings (industrial developer at work vs. Open Source developers in their free time), the IDEs (Visual Studio vs. Pharo), and the programming languages (C# vs. Smalltalk).

O4: The total short-inactivity time for Visual Studio users varies significantly from that of Pharo IDE users.

Code Editing & Execution: A C# developer in Visual Studio spends 28.5% of her time on code editing and execution. Minelli et al. [2] report that Smalltalk developers in Pharo spend a mere 5.8% of their time on comparable activities, while Beller et al. [3] report that Java developers in Eclipse already spend 30.5% of their time on only editing. We imagine that reasons for these huge differences include the programming language (C# vs. Smalltalk vs. Java), the study participants (e.g., experienced programmers vs. novices), the project lifecycle stage (e.g., development vs. maintenance), and the IDE UI concept (perspectives vs. floating windows).

O5: The reported times for code editing and execution varies significantly between Pharo, Eclipse, and Visual Studio users.

Interestingly, there is a strong correlation between the time for code editing and execution and the average duration of continuous-interaction periods per developer day (Pearson's $r = .69, p = .01$). This indicates that developers who are less frequently interrupted spend more time on code editing and execution or that focused developers are less likely to get interrupted. There is no significant correlation between the average duration of continuous-interaction periods and the time spent on any other activity.

O6: The average continuous-interaction time and the time for code editing and execution are strongly correlated.

Navigation: For about 22.4% of her time, a developer navigates the code base. Surprisingly, we find only a weak correlation between navigation and code editing and execution (Pearson's $r = .2, p = .01$). This suggests that navigation is not necessarily a means to reach code with the intention of editing or executing it. A developer might navigate a lot without editing much and edit much without navigating a lot. On the other hand, it is interesting to see that there is a strong correlation between navigation and the number of short inactivities (Pearson's $r = .9, p = .01$). We see two possible explanations for this: Either inactivities happen when the developer becomes unsure about some property of the codebase, which she then navigates to look up, or while navigating the codebase she regularly stops to read and understand. Both alternatives suggest that the amount of navigation may correlate with the need for code understanding.

O7: The amount of navigation is a likely indicator of the need for code understanding.

IDE Configuration: A developer spends 3.5% of her in-IDE time on configuring Visual Studio. In comparison, Minelli et al. [2] report that their participants spend almost 15% of in-IDE time fiddling with Pharo's UI. We speculate that this large difference might, in part, be caused by the different UI concepts of Visual Studio and Pharo. In Pharo, windows can be arranged independently and may overlap. In Visual Studio, windows are embedded into areas of the main window (perspective) and resizing one window automatically adjusts other windows such that they never overlap. Moreover, Visual Studio maintains and automatically switches between separate perspectives for debug and design mode, saving the need for rearrangement. The additional freedom provided by Pharo may result in developers spending more time on UI fiddling and configuration, to reach the perfect setup and layout for the (current) needs.

If we look at identified developers, the averages for Visual Studio configuration range from 0.4% to 21.9%. In comparison, Minelli et al. [2] report that their participants spent from 4% to as much as 30% of their time fiddling with Pharo's UI. A reason for the consistent high variance could be different experience of developers with the IDE.

O8: The time spent on IDE configuration varies significantly between Visual Studio and Pharo as well as between individual developers of each IDE.

Project Management: A developer spends only 1.8% of her time on project management. We know that CompanyX mandates Microsoft's Team Foundation Server (TFS) as the task management and versioning system. TFS is fully integrated into Visual Studio. Nevertheless, developers could choose to use IDE-external tools, like a standalone TFS client, over the integration, e.g., because of its specialized interface.

O9: Developers spend little time using the IDE-integrated project-management tools.

Building: A developer spends 0.9% of her time waiting for builds. Note that this includes only the time from the start of the build to the developer's next interaction. We find that the total build times are about four times larger. This supports our intuition that developers continue working during builds.

Table II: Top 10 Commands, Sorted by Absolute Usage Frequency

Rank	Command	Frequency	Usage
1	**CodeCompletion**	63,885	19.7%
2	**Debug.StepOver**	41,294	12.8%
3	**Edit.Paste**	25,840	8,0%
4	Debug.Start	23,563	7,3%
5	**Edit.Copy**	17,558	5,4%
6	**File.SaveSelectedItem**	14,417	4,5%
7	Window.NextDocumentWindowNav	7,223	2,2%
8	QueryResultsMaxRows	7,220	2,2%
9	AltEnter	6,649	2,1%
10	Build.BuildSolution	5,586	1,7%
...			

Table III: Top 10 Assistance Tools, Sorted by Usages per Dev. Day

Rank	Assistance-Tool	Dev. Days with Usage	Usages per Dev. Day	Usages per Usage Day
1	Code Completion	511 (87%)	78.8	90.5
2	Build System	546 (93%)	13.4	14.4
3	Debugger	520 (89%)	12.8	14.4
4	Textual Search	521 (89%)	7.7	8.6
5	AltEnter	403 (69%)	7.7	11.2
6	Code Search	486 (83%)	7.1	8.6
7	Version Control	521 (89%)	6.4	7.2
8	(Un)Comment Code	308 (52%)	2.3	4.5
9	Unit Testing	158 (27%)	1.9	7.0
10	Data Tools	50 (9%)	1.7	20.3

O10: Developers continue working while builds run in the background.

***Other Activities**:* Only 5.3% of a developer's in-IDE time is spent outside of the above activities. To determine our high-level overview of a developer's activities, we consider this fraction small enough to spend no more effort on assigning these interactions to our activities or come up with new ones.

C. How a Developer Uses Visual Studio's Tools

Before we analyze tool usage in Visual Studio, we look at the usage of individual commands. Table II show the top 10 commands by absolute usage frequency. We find that our top 10 commands for Visual Studio are very similar to the top 10 Murphy et al. [1] report for Eclipse. The commands printed in bold font in Table II appear in both lists. Apart from these, their top 10 includes only simple keystrokes, which our noise reduction filters. However, these keystrokes are also among the most-frequent commands in our unfiltered list.

Next, we analyze how developers use Visual Studio's assistance tools. Table III shows the top 10 tools. The full list is available on our artifact page. Note that this list does not contain Visual Studio's code editor, since we consider it a core feature of the IDE. This might be different for IDEs with different editing concepts. Subsequently, we discuss those tools we can make interesting observations about. We present usage frequencies as tuples (D, I), with D being the percentage of developer days the tool is used on and I being the average number of interactions per developer day.

***Code Completion**:* The code completion is by far the most frequently used tool in Visual Studio $(87\%, 78.8)$. Note, however, that Visual Studio's code completion opens automatically, whenever the developer starts typing. This usage frequency, therefore, represents an upper bound to the number of explicit triggers. Developers possibly ignore the suggestions provided by this automatic tool. To examine this more closely, we compute the frequency of code completion usages where the developer selected a proposal. This gives us a usage frequency of $(80\%, 69.5)$, which still ranks code completion as the most-frequently-used tool. When we consider only the manual invocations of code completion, the usage frequency is $(54\%, 3.7)$, which would still rank the tool 7th. This matches the results of Murphy et al. [1] who report that Eclipse's Content Assist is used about as frequently as standard editing commands like copy and paste. Since Eclipse's content assist opens only when the developer explicitly invokes it or when she types a dot, we believe that their numbers more accurately reflect the intentional usages. Unfortunately, we cannot compare usage frequency between IDEs, because Murphy et al. report only a relative frequency to other commands.

O11: Code completion is the most frequently used tool.

***Debugger**:* The debugger is the third most frequently used tool in Visual Studio $(89\%, 12.8)$. This is in line with the observations of Murphy et al. [1] who report `Debug.Step` among the top 10 most-frequently-used commands in Eclipse. In contrast, Meyer et al. [17] find industrial developers to only spend 3.9% of their time debugging.

***Searches**:* We find that both textual search $(89\%, 7.7)$ and code search $(83\%, 7.1)$ are very frequently used. Code search groups specialized searches, e.g., for usages or declarations. Singer et al. [8] make the same observation.

O12: Searches are frequently used to navigate the codebase.

***Quick Fix**:* R#'s quick-fix mechanism AltEnter is the 5th-frequently used tool $(69\%, 7.7)$. AltEnter offers a context-specific set of *simple* refactorings when the developer presses Alt + Enter. The three most often applied refactorings are `Organize Imports`, which removes unnecessary imports, `Change Name Fix`, which changes a name to match a naming convention, and `Use Var Fix`, which replaces a type name by C#'s `var` keyword. This aligns with the findings of Johnson et al. [12] who report that developers adore quick fixes that automatically resolve code or style problems.

O13: Developers very frequently use R#'s quick-fix tool.

***Version Control**:* A developer frequently interacts with the version control via Visual Studio's TFS integration $(89\%, 6.4)$. Moreover, she uses version control consistently, i.e., on 89% of all developer days. The only tool she uses more consistently is the build system. Since a developer spends little time on project management (*O9*), we conclude that the TFS integration supports her effectively in her regular usages.

O14: Developers interact with the integrated version control multiple times on almost every day.

***Unit Testing**:* Testing is considered one of the main activities accompanying software development [18]. We find that the unit-testing component of R# is the 9th most frequently used tool $(27\%, 1.9)$. However, it is used on little more than

a fourth of all developer days. This aligns with the findings of Beller et al. [3] who report that the majority of participants in their study do not actively practice unit testing.

A caveat to this finding is that some of the developers at CompanyX use NCrunch [19], an automated concurrent testing tool for Visual Studio. NCrunch detects and runs tests that exercise code changes automatically, thus, requires no explicit interaction once set up. The test results are shown in the editor, when a test class is open, and in a dedicated window. To estimate how many developers use NCrunch, we count how many identified developers either interact with an NCrunch results window or execute an NCrunch configuration command. We identify 9 such developers with occurrences on a total of 21 developer days. We deduce that the number of developers actively using NCrunch is small.

VII. Towards Next-Generation IDEs

The observations we make during our study highlight interesting questions about IDE design. In this section, we discuss research opportunities that can answer these questions and present possible ways to investigate them further.

A. To Integrate or Not to Integrate?

Developers spend a considerable amount of time outside of the IDE (*O1*). It is likely that, during this time, they use IDE-external tools for their work. Indeed, LaToza et al. [7] report that developers frequently use external tools. With several plugins available for most modern IDEs, many toolchains are now integrated into the IDE. However, it is unclear when such integration is effective and why certain tools are (not) used.

In their survey, LaToza et al. [7] find that tool usage often correlates with developer preferences. We know, however, that there are other factors that influence toolchains, like CompanyX's policy to use TFS. In this case, we observed that developers use Visual Studio's integrated client (*O14*). We assume that the reason for this is the efficiency the integrated solution (*O9*). That is, both policy and efficiency may be criteria for choosing a particular (integrated) tool.

To find out more about the criteria that guide tool choices, we could extend FeedBaG to capture the window name of the currently focused application and whether or not interactions occur. We could then generate a personalized survey to ask developers for their reasons to use the observed tools and their satisfaction in using them. Such a study would give valuable insight into which tools developers use and why. It would also help in identifying problems with developers' toolchains, and whether additional integrated IDE support can overcome them.

From the tools Visual Studio already provides, developers very frequently use code completion (*O11*) and quick fixes (*O13*). Both tools are provided through a comparatively simple-to-use dropdown that offers context-sensitive proposals. Nevertheless, we find that code completion is used much more frequently and on many more developer days than quick fix. One major difference we see between the tools is that code completion automatically opens on writing, while the quick fix requires an invocation by the developer. This aligns with the findings of Johnson et al. [12] that automation and workflow integration of tools are major factors for tool adoption. It obviously does not make sense for all tools, e.g. for automated refactorings, to open automatically. However, a profound understanding of how integrated-tool presentation impacts adoption would be valuable for IDE designers.

B. Why Are Developers Inactive?

Developers spend a third of their in-IDE time not interacting with the IDE (*O2*). Besides a few longer breaks, we observe that many short inactivities (*O3*) heavily fragment their activities. Although we cannot say whether such fragmentation is problematic, we find that the duration of continuous activity strongly correlates with time spent on code editing and execution (*O6*). Thus, eliminating the reasons for short inactivities might increase developer productivity.

Minelli et al. [2] assume that short inactivities occur when the developer has to understand code. Our data and intuition tell us there are additional reasons, like when she is waiting for a test run to finish. Since we only track IDE interactions, we cannot generally identify why a developer becomes inactive. However, we believe that we can derive inactivity reasons, to some extend, from the activities surrounding the inactivity. A first experiment, analyzing which individual commands frequently precede short inactivities, did not lead to interesting findings. However, mining for larger interaction patterns might. This would help to refine our understanding of the time developers spend on program understanding. It could also help identify when developers often wait for the IDE to finish some task, thereby guiding IDE designers towards bottlenecks.

A common approach to mitigate the impact of long-running tasks, such as builds or static analyses, on developers is to run them in the background. This strategy seems successful since we find, for example, that developers do not wait for build runs (*O10*). However, previous studies show that expensive background computations, such as static analyses or builds, often slow down a developer's work in the IDE or make her digress [12], [17]. To investigate such impacts, we would, first, identify which interactions trigger long-running background tasks and, second, analyze which kinds of interactions happen while such tasks are running. We could then identify developers' reactions to certain tasks, like switching away form the IDE, or the impact of tasks, like slowing down interactions.

C. What Causes Usage Differences between IDEs?

General aspects of IDE usage, such as the time spent on code editing and execution, on IDE configuration, or in short inactivities vary significantly between different IDEs (*O5*, *O8*, and *O4*). To enable the understanding of how project nature and developers influence these aspects, we incorporated a respective questionnaire into the new version of FeedBaG. We also hypothesize that the difference in the time spent on IDE configuration between Minelli et al.'s study [2] and ours might be partially caused by the different UI concept of Visual Studio and the Pharo IDE. Exploration of such usability indicators could guide IDE designers in creating future IDEs.

D. Why Do Developer Navigate the Codebase?

The frequent usage of search tools (*O12*) indicates that developers often need to navigate the codebase. Due to the strong correlation between navigation time and the number of short inactivities, we hypothesize that the amount of navigation may be an indicator for a developer's need for code understanding (*O7*). It would be interesting to explore this hypothesis with additional datasets, especially from other IDEs, to encourage better support for code understanding, e.g., using documentation miners [20] or example providers [21].

E. Should IDEs Distinguish Developer Persona?

The amount of time spent on IDE configuration varies significantly between developers (*O8*). A reason for this may be a developer's familiarity with the IDE. It would be interesting to see if future studies on various IDEs confirm such a correlation. If so, then IDE designers could use configuration time as a metric to provide specific support to new IDE users. Another reason for the observed variation in IDE configuration time might be the kind of tasks a developer performs, e.g., testing versus feature development. This information could also be used to personalize IDEs for different developer roles.

VIII. Threats to Validity

Internal Validity: It is possible that FeedBaG fails to track some interactions, which would lead us to false conclusions, especially about inactivities of developers. To mitigate this risk, we performed our own pilot phase of over six months with two of the authors and six students. During this time, we repeatedly analyzed the tracked interactions and improved both the correctness and completeness of FeedBaG.

One threat to our conclusions is that we did not conduct interviews with our participants to verify our interpretation of the data. It is also possible that some developers behaved differently while using FeedBaG, to better present themselves. However, since they were aware that the collected data is completely anonymous and that, legally, their managers cannot access it, we believe that it is very unlikely that their behavior was "staged" over the six months of data gathering.

Construct Validity: One threat to the statistics we collect is that we do not know the exact number of participants in our study, because we were not allowed to uniquely identify them. Privacy laws are very strict in Germany, and it took us the better part of a year to discuss our intents and our tool with the privacy council. To overcome this, we present a lower and upper bound to the number of participants in our study, but perform all our analyses on *developer days*. While we cannot state conclusions about individual developers, we can safely analyze what a typical workday for a developer looks like. For future studies, we are currently working on integrating a questionnaire into FeedBaG that will track information such as the years of development experience and attach it to each upload. Unfortunately, we could not get this feature admitted by the privacy council in time for this study.

FeedBaG allows participants to delete (parts of) the recorded sessions before uploading it to our servers. It is possible that the large amount of inactivity is due to participants making heavy use of this feature. However, we doubt that developers would go through the hassle of deleting many individual events. It is much more likely that they delete an entire day, should they be unwilling to share information about larger parts of it. Therefore, we assume that the days we received events for are close to complete.

External Validity: We conducted our study in a single company; our results may not generalize beyond that. To validate our findings, we started a complementary study with Open Source developers. Additionally, we track only IDE interactions for C# developers in Visual Studio. It is possible that the observations we make do not generalize to developers using other programming languages or IDEs. However, we believe that our results help in reasoning about differences between languages and tools, and we already compare our results to similar studies performed with other IDEs [1]–[3].

Since we do not know the experience level and roles of our participants, it might be the case that most of our participants fall into one experience level, for example. However, we believe that the large number of (measured) participants and the diverse project portfolio of CompanyX mitigate this threat significantly. The many similarities between the behavior of our participants and behavior reported in other studies [1]–[3], [7], make us believe that our participants are representative.

IX. Conclusions

To further improve Integrated Development Environments (IDEs), we need to understand how developers typically spend their time in an IDE and which tools they actually use. Previous studies investigated how developers use Eclipse [1] and the Pharo IDE [2]. We expand this space of knowledge by a study on how professional C# developers use Visual Studio. We presented FeedBaG, an interaction tracker for Visual Studio. We use FeedBaG to capture interactions from over 800 developer days, representing a total of 6,300 working hours. We analyze this data to understand how developers spend their time and which IDE tools they use.

Our findings provide insights for IDE designers on how to improve future IDEs. Possible opportunities include removing bottlenecks that slow down or interrupt developers in their work and creating more code-completion-like or quick-fix-like mechanisms that quickly and simply support developers in their workflow. We also find indications that new concepts for code understanding and exploration might be valuable for developers. Furthermore, we find indications that the IDE UI concept might significantly impact developers in their work.

X. Acknowledgments

We thank our students D. Albrecht, U. Fahrer, M. Kämmerer, S. Kemper, F. Weirich, and M. Zimmermann for their work on FeedBaG. Special thanks to K. Brockmann, A. Fischer, and M. Kutter, our liaisons to CompanyX.

This work was partially funded by the German Federal Ministry of Education and Research (BMBF) within the Software Campus projects *KaVE* and *Eko*, both grant no. 01IS12054. The authors assume responsibility for the paper content.

References

[1] G. Murphy, M. Kersten, and L. Findlater, "How Are Java Software Developers Using the Elipse IDE?" *IEEE Software*, vol. 23, no. 4, 2006. [Online]. Available: http://dx.doi.org/10.1109/MS.2006.105

[2] 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 the International Conference on Program Comprehension*, ICPC '15. IEEE Press, 2015, pp. 25–35. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820282.2820289

[3] M. Beller, G. Gousios, A. Panichella, and A. Zaidman, "When, How, and Why Developers (Do Not) Test in Their IDEs," in *Proceedings of the Joint Meeting on Foundations of Software Engineering*, ESEC/FSE '15, 2015, pp. 179–190. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786843

[4] "A Study of Visual Studio Usage in Practice – Online Artifact," http://www.st.informatik.tu-darmstadt.de/artifacts/vs-in-practice/.

[5] D. Perry, N. Staudenmayer, and L. Votta, "People, Organizations, and Process Improvement," *IEEE Software*, vol. 11, no. 4, pp. 36–45, 1994. [Online]. Available: http://dx.doi.org/10.1109/52.300082

[6] V. M. González and G. Mark, ""Constant, Constant, Multi-tasking Craziness": Managing Multiple Working Spheres," in *Proceedings of the Conference on Human Factors in Computing Systems*, CHI '04. ACM, 2004, pp. 113–120. [Online]. Available: http://doi.acm.org/10.1145/985692.985707

[7] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining Mental Models: A Study of Developer Work Habits," in *Proceedings of the International Conference on Software Engineering*, ICSE '06. ACM, 2006, pp. 492–501. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134355

[8] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil, "An Examination of Software Engineering Work Practices," in *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '97. IBM Press, 1997, pp. 21–35. [Online]. Available: http://dl.acm.org/citation.cfm?id=782010.782031

[9] M. Kersten and G. C. Murphy, "Mylar: A Degree-of-interest Model for IDEs," in *Proceedings of the International Conference on Aspect-oriented Software Development*, AOSD '05. ACM, 2005, pp. 159–168. [Online]. Available: http://doi.acm.org/10.1145/1052898.1052912

[10] W. Snipes, A. R. Nair, and E. Murphy-Hill, "Experiences Gamifying Developer Adoption of Practices and Tools," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014. ACM, 2014, pp. 105–114. [Online]. Available: http://doi.acm.org/10.1145/2591062.2591171

[11] W. Snipes, E. Murphy-Hill, T. Fritz, M. Vakilian, K. Damevski, A. Nair, and D. Shepherd, "A Practical Guide to Analyzing IDE Usage Data," in *The Art and Science of Analyzing Software Data*. Morgan Kaufmann, 2015.

[12] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13. IEEE Press, 2013, pp. 672–681. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486877

[13] N. Ayewah, D. Hovemeyer, J. Morgenthaler, J. Penix, and W. Pugh, "Using Static Analysis to Find Bugs," *IEEE Software*, vol. 25, no. 5, pp. 22–29, 2008. [Online]. Available: http://dx.doi.org/10.1109/MS.2008.130

[14] "ReSharper," https://www.jetbrains.com/resharper/. Last checked on November 13, 2015.

[15] "KaVE Project – Website," http://kave.cc. Last checked on November 13, 2015.

[16] S. Proksch, J. Lerch, and M. Mezini, "Intelligent Code Completion with Bayesian Networks," *ACM Transactions on Software Engineering and Methodology*, to appear, http://www.st.informatik.tu-darmstadt.de/artifacts/pbn/.

[17] A. N. Meyer, T. Fritz, G. C. Murphy, and T. Zimmermann, "Software Developers' Perceptions of Productivity," in *Proceedings of the International Symposium on Foundations of Software Engineering*, FSE '14. ACM, 2014, pp. 19–29. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635892

[18] F. P. Brooks, Jr., *The Mythical Man-month (Anniversary Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[19] "NCrunch," http://www.ncrunch.net/. Last checked on November 13, 2015.

[20] L. Moreno, G. Bavota, M. D. Penta, R. Oliveto, and A. Marcus, "How Can I Use This Method?" in *Proceedings of the International Conference on Software Engineering*, ICSE '15. IEEE Press, 2015, pp. 880–890. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2015.98

[21] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, "Mining StackOverflow to Turn the IDE into a Self-confident Programming Prompter," in *Proceedings of the Working Conference on Mining Software Repositories*, MSR '14. ACM, 2014, pp. 102–111. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597077