Enriched Event Streams: A General Dataset For Empirical Studies On In-IDE Activities Of Software Developers

Sebastian Proksch
University of Zurich
proksch@ifi.uzh.ch

Sven Amann
Technische Universität Darmstadt
amann@cs.tu-darmstadt.de

Sarah Nadi
University of Alberta
nadi@ualberta.ca

ABSTRACT

Developers have been the subject of many empirical studies over the years. To assist developers in their everyday work, an understanding of their activities is necessary, especially how they develop source code. Unfortunately, conducting such studies is very expensive and researchers often resort to studying artifacts after the fact. To pave the road for future empirical studies on developer activities, we built FeedBaG, a general-purpose interaction tracker for Visual Studio that monitors development activities. The observations are stored in enriched event streams that encode a holistic picture of the in-IDE development process. Enriched event streams capture all commands invoked in the IDE with additional context information, such as the test being run or the accompanying fine-grained code edits. We used FeedBaG to collect enriched event streams from 81 developers. Over 1,527 days, we collected more than 11M events that correspond to 15K hours of working time.

KEYWORDS

Integrated Development Environment, In-IDE, Interaction Tracking, Event Stream, Usage Data, Fine-Grained Change Information

ACM Reference Format:
Sebastian Proksch, Sven Amann, and Sarah Nadi. 2018. Enriched Event Streams: A General Dataset For Empirical Studies On In-IDE Activities Of Software Developers. In *MSR '18: 15th International Conference on Mining Software Repositories , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3196398.3196400

1 INTRODUCTION

Understanding how developers produce software is essential for creating tools that can support them in their daily tasks. This includes understanding, for example, how they change code, when they invoke builds or run test suites, and when such invocations lead to more code changes. Over the last couple of decades, many researchers have built tools and conducted studies to understand developers' actions and needs [1–3, 7, 10]. Typically speaking, they either studied developers in retrospective through mining software repositories or in vivo by observing the commands or actions they execute in an Integrated Development Environments (IDEs). Using either of these information sources alone provides different views on development activities. Analyzing code evolution in the version-control system, on the one hand, provides a view on potentially stable code snapshots, but usually misses more fine-grained changes. Analyzing in-IDE activities, on the other hand, may provide insights about the usage of IDE tools, such as automated refactorings, but often misses the evolution of the source code itself. Ideally, we want to combine and link both these perspectives. To capture such a holistic picture of developers' activities, we designed *enriched event streams* [6], a novel meta model for in-IDE development activities that captures not only all executed commands in the IDE, but also additional context information about them. Enriched event streams can help, for example, in answering the following research questions:

- Which commands do developers use?
- How are test cases executed?
- Does refactoring lead to more failed tests?
- How do developers navigate the code base?
- Are there common patterns in debugging behavior?
- What kind of changes do developers revert?

While previous work may have tackled similar questions, we are still lacking large public datasets and holistic representations that are not tailored to a specific research question. We hope that we can facilitate future research in this area by providing a general dataset that provides ample opportunity to study developer activities.

2 DATASET

Our "March 1, 2017" release contains 11M events that have been uploaded by a diverse group of 81 developers.[1] Out of these developers, 43 come from industry, three are researchers, five are students, and six are hobby programmers. Twenty-four participants did not provide this (optional) information about their position. The data covers a total of 1,527 aggregated days and was collected over eleven months, but not all developers participated the whole time. On average, each developer provided 136K events (median 54K) that have been collected over 10 days (median 18.9 days) and that represent 185 hours of active work (median 48 hours). In total, the dataset aggregates 15K hours of development work.

In our own work, we were most interested in the usage of code completion, test execution, and source-code evolution, for which we have provided the most advanced instrumentation. The dataset contains detailed data about 200K usages of the code completion, including a snapshot of the surrounding source code, as well as 3.6K test executions. An average user provides 2.5K usages of the code completion (median 640) and 44 test executions.

The dataset is available on our project page [4] and can be downloaded for local processing. We provide an API for both Java and C#

[1]Developers that have contributed less than 2,500 events are already filtered out.

that allows reading the data and we have created examples in both languages that help to get started. Technically, the dataset stores a *Javascript Object Notation* (JSON) representation of our collected events and can also be read and processed in other languages. The appendix contains a more elaborated description of the meta model.

3 APPENDIX

In this appendix, we elaborate more on the context of the project and on the concrete data. We start by providing a more detailed explanation of the data format, then present our interaction tracker FeedBaG and the setup of our field study, in which we have gathered the dataset from our users. Finally, we present details on the data organization and on possible resources to get started.

3.1 Data Format

Developers work in their IDE on various tasks and with many tools. In order to analyze their actions after the fact, it is necessary to store them in a form that can be serialized and shared. The typical way to store interactions is to model them as a stream of events (e.g., click-through data in the web or system log in applications), but these representations are usually "flat," in the sense that they do not provide any details about the event that occurred. If it is possible at all, researchers have to reconstruct the context of the interaction by analyzing the stream.

To improve over this situation, we have introduced a new meta model for in-IDE developer activities that we call *enriched event streams*. The representation is event-based and captures additional context information to facilitate later analyses of developers' activities.

Our conceptual design of the data structure is shown in Figure 1 and can be divided into two layers. The first layer represents the *process* through a hierarchy of `IDEEvents` that denote the different kinds of interactions. Each event has a second layer in which additional *context* information about a captured interaction is stored. The basic information that is universally (e.g., timing) used is stored as generic context information in the base class. In addition, each derived event class can capture specialized context (e.g., test results or code snapshots). Both the generic and a specialized context is available for each event. A third layer that is not depicted is a naming scheme for both *IDE components* (e.g., identifiers of windows or file names) and *code elements* (e.g., types or methods) that allows to unambiguously refer to locations, targets, or code elements [6].

Example. The design is illustrated in a simplified example of an enriched event stream in Figure 2 that shows the conceptual split into the two levels: *process* and *context*. For the example, assume that a developer is writing source code in a file, saves the file, and then commits it to the repository.

On the *process level*, we capture single events in a stream. Each event has an *event type*. The types that are relevant for our example are *source code change*, *file save*, and *versioning action*. Other events might happen in between (e.g., *window close*), which we omit here for brevity. The events build a stream that allows following the different activities of the developer after the fact.

On the *context level*, we allow the storage of additional information, to support the in-depth investigation of development activities. The context information includes generic information (e.g., the time

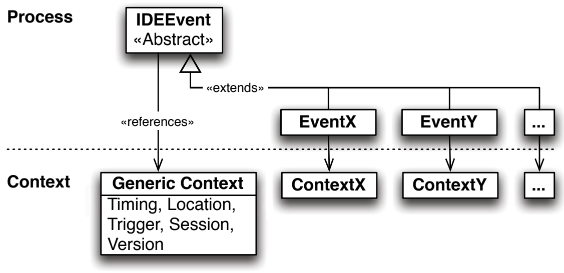

Figure 1: Conceptual Design of Enriched Events

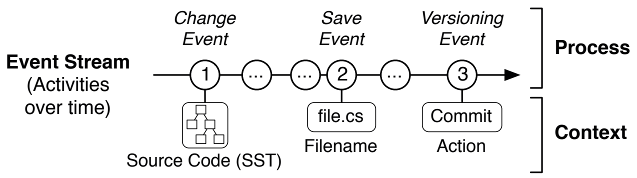

Figure 2: Simplified Example of an Enriched Event Stream

at which the events were triggered), but also additional context that is specific to the individual event types. For example, the *edit event* contains a snapshot of the current file under edit, which allows an analysis of the edited source code after the fact. The *save event* stores the name of the file that was saved and the way in which the save action was invoked (e.g., by shortcut or by clicking a menu entry). The *version-control event* indicates the name of the current solution and the version control action, i.e., that changes have been committed. All these events also contain other information that we omit for brevity in this illustration.

One example of extended context information is the code snapshots that edit events capture. We go beyond simple plain-text snapshot and capture a simplified version of the abstract syntax tree. These *Simplified Syntax Trees* [5, 9] combine two main ideas:

(1) Resolved typing information is preserved by storing references to types and type members in a fully-qualified naming scheme. For example, every method reference contains information about the declaring type (incl. the declaring assembly) and also about its full signature (i.e., the list of parameters and the return type). We also preserve information about the inheritance hierarchy of the edited type.
(2) The actual syntax tree is being normalized to facilitate static analyses by reducing the maximum depth of the tree. Nested expressions other than literals or references are being assigned to artificial variables and replaced with a respective reference.

3.2 Supported Activities

Our meta model supports a wide range of development activities. A complete categorization is presented in Figure 3. We now discuss the individual categories.

Activity. At the core, we are interested to model "what the developer is doing," which includes all tools and commands that get executed. Unsupported by the traditional system log, some "non-activities" do not leave a trace, because they either do not correlate to a development activity (e.g., scrolling) or because they do not involve user interaction at all (e.g., screen saver started). To improve over the traditional system log, we capture the following activity-related events to distinguish these cases.

Commands Invocation of an action. For example, clicking a menu button or invoking a command via shortcut.

(In-) Activity Interactions do not necessarily imply an execution of a command. We store the basic information that a developer has interacted with the computer in any way in order to allow the inference of real inactivities.

System Event Non-IDE events that indicate an inactivity of the developer, e.g., system sleep or screen lock.

Focus We store whether the IDE had the window focus.

File Management. Working on software naturally includes editing source code files and organizing them into folders and workspaces (i.e., *solutions* in Visual Studio terminology). We include the following events to model these activities.

Solutions Interactions that relate to changes of the solutions, e.g., adding or removing of projects or files.

Edits We preserve the information that a developer has edited a source file. After every change, we create an edit event that includes a source-code snapshot in the form of an SST.

Environment. To complete the picture of general IDE usage, we capture information about IDE sessions and about the environment in which the developer is working. More specifically, we preserve information about the state of the IDE (e.g., when it is being (re)started) and the configuration of the local working environment (e.g., moving or closing windows). We capture:

IDE State Explicit marker of start and shutdown of the IDE.

Windows Captures how developers place their windows and how they interact with them.

Documents Open, close, and switch of documents.

Navigation. Research distinguishes two kinds of navigation: unstructured browsing (e.g., when a developer does not know what to look for and tries to find it by navigating to related documents that are semantically linked) or directed searching (e.g., when a developer knows what to look for, but not where). We designed enriched event streams to capture enough information to enable reasoning about both kinds.

Structural Navigation We capture navigation that follows the structure of the type system. We store both the navigation within a document (e.g., moving the cursor to another method) as well as "ctrl-click navigation" to referenced files.

Search Tool Our event stream preserves invocations of a search tool. It is possible to distinguishing actual use of search results from aborting the search.

Specific Tools. Software development is a combination of various activities. One of the strengths of enriched event streams is that they do not stop at the general level, but that they are extensible to very low-level details. We added support for the following essential tools that are at the core of software engineering activities. We capture their detailed information in enriched event streams.

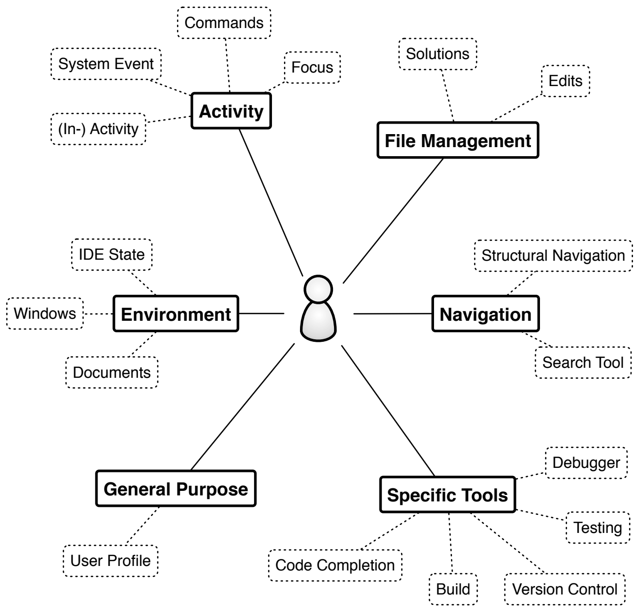

Figure 3: Categories of Supported Development Activities

Build In VisualStudio, building is an activity that has to be triggered regularly by the developer. We model build actions, targets, and results.

Debugger Debugging programs is one of the most common development activities, and we preserve all the information about the process.

Testing We model test executions and their results.

Version Control We capture the version-control activities of developers, which could be used in studies on source-code evolution, e.g., to infer the maturity of edited source files.

Code Completion Code completion is a widely used core feature of modern IDEs. We capture information about its invocation, the selection of proposals, and the context in which it is triggered.

We also request a simple user profile from our participants to get demographic information about our user base, which is also modeled as an event. You will find a more detailed description of all supported events on the website of the project [4].

3.3 Interaction Tracking

To capture developer activities, we built FeedBaG, a general-purpose interaction tracker for Visual Studio that captures developers' interactions with their IDE. FeedBaG is publicly available and can be installed as a plugin for ReSharper. The tool represents a general solution for capturing interaction data and is designed to cater for different use cases. It can be deployed in controlled experiments in a minimalistic release that just contains the instrumentation, but we also provide extensions for other settings (e.g., field studies), in which extensive infrastructure is required. This includes, for example, infrastructure for reviewing and sharing collected data.

We have designed FeedBaG to be as non-invasive as possible. Once installed, it runs transparently in the background and captures all performed in-IDE actions in an *enriched event stream*. It does not provide any value to the developer to avoid any potential bias in the collected dataset.

3.4 Field Study

We have conducted a field study to collect data from a large number of participants. The nature of FeedBaG made it hard to find participants for our field study, because no incentives exist apart from being a part of bleeding edge research.

To find our participants, we advertised the field study directly in social media channels and through advertising efforts (e.g., MSR year book), and indirectly through referring to the project in our papers. Our invitation to use FeedBaG and to participate in the data collection was open to any interested developer, which means we did not target a specific population of users. However, our assumption is that a high number of random participants and long observation times provide a rich dataset with a variety of projects and participant backgrounds.

To make installation as easy as possible, we released the tool in the ReSharper gallery [8], the standard repository for public ReSharper extensions. The extension can be installed by all interested developers directly from within their IDE using the ReSharper extension manager. Over the course of the project, it was necessary to update the tool several times to adapt to breaking changes introduced by updates to the ReSharper SDK. Since its original release, FeedBaG has been downloaded more than 1,200 times and is regularly listed among the "Top 100" ReSharper extensions.

3.5 Data Organization

The uploaded data of the participants is unstructured and the individual contributions of a single participant are scattered across several files. We performed a preprocessing step before releasing the dataset in which all events uploaded by the same user were merged, sorted chronologically, and filtered for obvious noise, such as duplicated events. After the preprocessing, the data is easier to process, e.g., all events that were shared by an individual developer are contained in a single file.

We encouraged users to use the same identifier across machines to allow us to group activities of the same user. As a result, active phases recorded on multiple concurrent IDE instances may overlap in the event stream. Depending on the research questions, it might be necessary to split these phases by programming sessions. This split can be achieved using the *session id* that is stored in each event.

The events are stored in plain text files that correspond to the *Javascript Object Notation* (JSON) of the event data structures that we have introduced above. All events of the same users are compressed in an archive, and all archives are again combined in a single file for easier distribution. The dataset can directly be opened and processed with the tooling that we provide for Java and C#. The challenge website [4] provides examples that illustrate the required steps to read and use the interaction dataset.

REFERENCES

[1] Philip M. Johnson, Hongbing Kou, Joy Agustin, Christopher Chan, Carleton Moore, Jitender Miglani, Shenyan Zhen, and William E. J. Doane. 2003. Beyond the Personal Software Process: Metrics Collection and Analysis for the Differently Disciplined. In *International Conference on Software Engineering*. IEEE.
[2] Mik Kersten and Gail C. Murphy. 2005. Mylar: A Degree-of-interest Model for IDEs. In *International Conference on Aspect-oriented Software Development*.
[3] Roberto Minelli, Andrea Mocci, Romain Robbes, and Michele Lanza. 2016. Taming the IDE with Fine-grained Interaction Data. In *International Conference on Program Comprehension*.
[4] Sebastian Proksch, Sven Amann, and Sarah Nadi. 2017. Website of the MSR Challenge Proposal. http://www.kave.cc/msr-mining-challenge. (2017).
[5] Sebastian Proksch, Sven Amann, Sarah Nadi, and Mira Mezini. 2016. A Dataset of Simplified Syntax Trees for C#. In *International Conference on Mining Software Repositories*. ACM.
[6] Sebastian Proksch, Sarah Nadi, Sven Amann, and Mira Mezini. 2017. Enriching In-IDE Process Information with Fine-grained Source Code History. In *International Conference on Software Analysis, Evolution, and Reengineering*.
[7] Will Snipes, Anil R Nair, and Emerson Murphy-Hill. 2014. Experiences Gamifying Developer Adoption of Practices and Tools. In *International Conference on Software Engineering*.
[8] KaVE Team. 2017. Release of the FeedBaG Interaction Tracker. https://resharper-plugins.jetbrains.com/packages/KaVE.Project/. (2017).
[9] KaVE Team. 2017. Website of the KaVE Project. http://www.kave.cc/. (2017).
[10] Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P Bailey, and Ralph E Johnson. 2012. Use, Disuse, and Misuse of Automated Refactorings. In *International Conference on Software Engineering*. IEEE.