Chapter 1
Collecting and Processing Interaction Data for Recommendation Systems

Walid Maalej, Thomas Fritz, and Romain Robbes

Abstract. Traditional recommendation systems in software engineering (RSSE) analyze artifacts stored in large repositories to create relevant recommendations. More recently, researchers have started exploring interaction data as a new source of information—moving closer to the creation and usage of the artifacts rather than just looking at the outcome. In software engineering, interaction data refers to the data that captures and describes the interactions of developers with artifacts using tools. For instance, the interactions might be the edits or selections that affect specific source code entities or web pages (artifacts) using an integrated development environment or a web browser (tools). Interaction data allows to better investigate developers' behaviors, their intentions, their information needs, and problems encountered, providing new possibilities for precise recommendations. While various recommendation systems that use interaction data have been proposed, there is a variety in the data being collected, the way the data is collected, and how the data is being processed and used. In this chapter, we survey and summarize the major approaches for RSSEs that create recommendations based on interaction data. Along with this, we propose a conceptual framework for collecting and processing interaction data for the purpose of recommendation.

1.1 Introduction

Online retailers such as amazon.com or booking.com use data on how their users interact with the websites to automatically recommend potentially interesting items. A common scenario is "other users who looked at this products considered buying these products too ...". Similarly, search portals aggregate the web navigation history into a user profile to improve the relevance of search results [7].

Software engineering researchers also started looking into using developer's interaction data to make a variety of recommendations. The idea is that the single interactions such as document selections, code changes, command executions, or web

NOTE: This is the authors' version of the work.
Springer holds the copyright for the original publication, which can be accesses via it web site
http://www.springer.com/computer/swe/book/978-3-642-45134-8

searches allow for a better understanding of a developer's work and thus for more fine-grained and precise recommendations. Conventional software repositories such as version control systems provide only aggregated, high level information on a developer's work.

For example, a developer might work all day to fix a bug. While the version control system only stores the few code changes committed at the end of the day, the developer did a lot more than just perform the committed changes. For instance, the developer might have used the debugger to reproduce the bug, navigated through other parts of the code, read documentation, ran tests, or performed web searches to get help. A more fine-grained tracking of interaction data can be used to reflect the problems encountered by the developers and eventually recommend relevant documents, actions, people, or even pieces of code.

The ability to monitor almost every single interaction of a developer with modern tools, in particular within the integrated development environment (IDE), provides new and manifold opportunities for recommendation systems. Many RSSEs that use interaction data have been proposed. For instance, Mylyn [13] tracks the selections and edits of source code artifacts to filter most relevant artifacts for the current task. Other systems use the interaction data to suggest reusable pieces of code [32], predict defects [16], raise awareness amongst developers [6], or prevent conflicts in teams [15].

These recommendation systems vary mainly along the types of interaction data gathered, the artifacts concerned by the interaction, as well as how the interaction history is collected, aggregated, and used to recommend information of interest. In this chapter, we describe the general principles for collecting and processing interaction data for the purpose of recommendation. Along with this, we survey and summarize major approaches for recommendation systems in software engineering that are based on interaction data.

The remainder of this chapter is structured as follow. Section 1.2 presents three tools that use interaction data in order to support developers in their daily work. Section 1.3 defines interaction data, its main concepts, and granularity levels. Section 1.4 proposes a general framework for creating interaction data collection tools. Section 1.5 summarizes the main approaches to *process* interaction data, including the sessionization, filtering, and aggregation of interaction events. Section 1.6 discusses the main usage scenarios addressed by RSSEs that use interaction data: productivity and awareness. Finally, Sect. 1.7 presents the main challenges in the field and sketches future research directions.

1.2 Examples

After summarizing early foundational work, we present three tools that use interaction data to support developers in their work: Mylyn, Switch, and OCompletion.

NOTE: This is the authors' version of the work.
Springer holds the copyright for the original publication, which can be accesses via it web site
http://www.springer.com/computer/swe/book/978-3-642-45134-8

1.2.1 Early Work

Many recommendation systems that collect and use interaction data are based on early work from the human-computer interaction community. Hill et al. [12] monitored edits, selections, and scrolls to compute ”edit wear” and ”read wear” metrics of documents. Edit wear measures how often a given line in a document was edited, while the read wear measures how often it was read. The idea was derived from the wear in physical objects, which gets visible due to interactions with the objects. Similarly, the edit and read wear are shown in the scroll bar of the document, allowing to spot which parts of the document were changed and read most frequently. Wexelblat and Maes [38] presented a tool for tracking interactions with web documents to support navigation. The goal was to capture and reuse navigation patterns of web pages to make new web investigations on similar topics more efficient. The collected information can be displayed as a map of web pages, showing how often a page was visited.

In the software engineering community, DeLine et al. [3] proposed TeamTracks, a tool that reuses the read wear metaphor to filters the list of artifacts displayed in the IDE. The tool monitors the previous transitions between source code files to recommend related files when a file is browsed. Evaluation studies showed that the tool helped developers in program comprehension tasks. The tool also helped experienced developers, working on large systems, to remember related artifacts to the one they are currently browsing. Singer et al. [37] proposed a similar tool, which recommends files to developers during maintenance tasks. The assumption is that files involved in *short navigation cycles* are related. A repository of association rules is built based on the observed cycles and is mined thereafter to recommend files related to those currently being browsed.

1.2.2 Mylyn

Mylyn is one of the most popular software productivity tools that uses interaction data. Mylyn is a plugin for the Eclipse IDE that allows users to focus only on the code elements that are relevant for their current tasks. For this Mylyn maintains for each task a “task context”, which consists of the interaction data for that task. Based on this interaction data, Mylyn calculates a degree of interest (DOI) value for each code element [13]. This value represents the interest of a developer in the element for the given task. Whenever a developer selects or edits an element in the IDE, the element’s DOI value increases accordingly. At the same time, the DOI values of other elements decrease over time since interaction with them lies further back in the past. This recency aspect of DOI allows for the model to adapt to changing interest. Mylyn uses these DOI values to determine, filter, and highlight the most relevant code elements for a task at hand to counteract the information overload developers face in their IDE with the thousands of code elements that are usually displayed for

NOTE: This is the authors' version of the work.
Springer holds the copyright for the original publication, which can be accesses via it web site
http://www.springer.com/computer/swe/book/978-3-642-45134-8

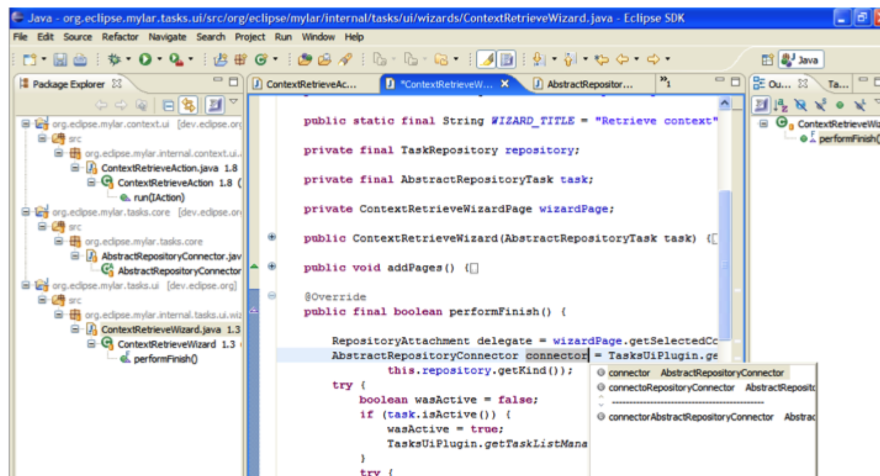

Fig. 1.1: Mylyn uses interaction data to recommend source code artifacts relevant for the current task

a single project. When using Mylyn, only the elements with a DOI value exceeding a certain threshold are shown in the IDE (see Fig. 1.1).

1.2.3 OCompletion

OCompletion [32] improves code completion tools based on a fine-grained analysis of previous edit interactions. When a developer is typing the beginning of a long method name, code completion tools generate suggestions to help the developer complete the name, making it easier and faster to complete the method name, and avoiding spelling mistakes. In many cases however, the list of suggestions is long and ordered alphabetically, making it time consuming for the developer to go through the list and find the relevant suggestion.

OCompletion addresses this issue by analyzing the changes made during the development session. It prioritizes the suggestions based on the recency of fine-grained interactions a developer previously had with the code. This approach makes it possible to have a short and accurate lists of relevant suggestions instead of long lists (see Fig. 1.2).

NOTE: This is the authors' version of the work.
Springer holds the copyright for the original publication, which can be accesses via it web site
http://www.springer.com/computer/swe/book/978-3-642-45134-8

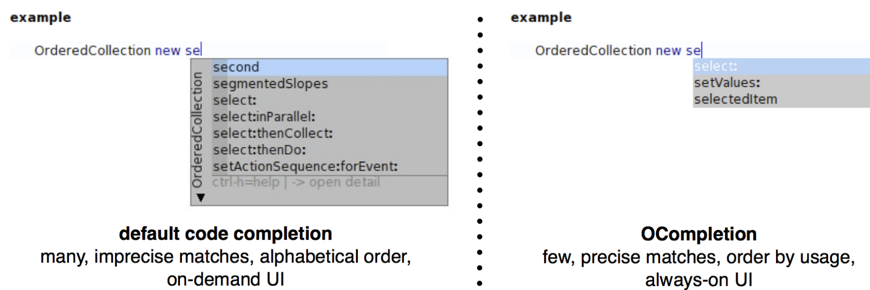

Fig. 1.2: OCompletion uses interaction data to recommend code completions

1.2.4 Switch!

Developers work with a variety of tools and artifacts, not just the IDE or artifacts within the IDE. For instance, developers frequently consult API documentation on the web, communicate with other developers via email or chat, and use specifications, diagrams, and plans best viewed and changed with specific tools. Often there are dependencies between the various artifacts that require the developer to switch back and forth between the artifacts to complete a given task.

Switch! [24] is a recommendation system that automatically infers these dependencies based on the sequence of interaction events and the types of artifacts. Unlike Mylyn and OCompletion, Switch! gathers the interactions a developer has with all tools in an operating system.Switch! uses the interaction data to create a reactive graphical interface (see Fig. 1.3) that allows developers to quickly switch to the artifacts that they will most probably need next.

1.3 What is Interaction Data?

Interaction data refers to a record of the actions taken by a user (in our case a software developer) with a tool. These actions are usually performed in a context, such as a specific task. Interaction data typically involves four types of data: interactions, artifacts, tools, and contexts as illustrated in Fig. 1.4:

Interactions. The actions (i.e., the interaction events) taken by a developer, such as the clicks of specific buttons, the changes to code entities, or the views of documentation pages;

Artifacts. The entities a developer is interacting with, such as a source code entity, an issue report, an email document, or even physical artifacts and people;

Tools. The software applications developers use during their work, such as the IDE, the browser, the issue tracking system or the email client; and

NOTE: This is the authors' version of the work.
Springer holds the copyright for the original publication, which can be accesses via it web site
http://www.springer.com/computer/swe/book/978-3-642-45134-8

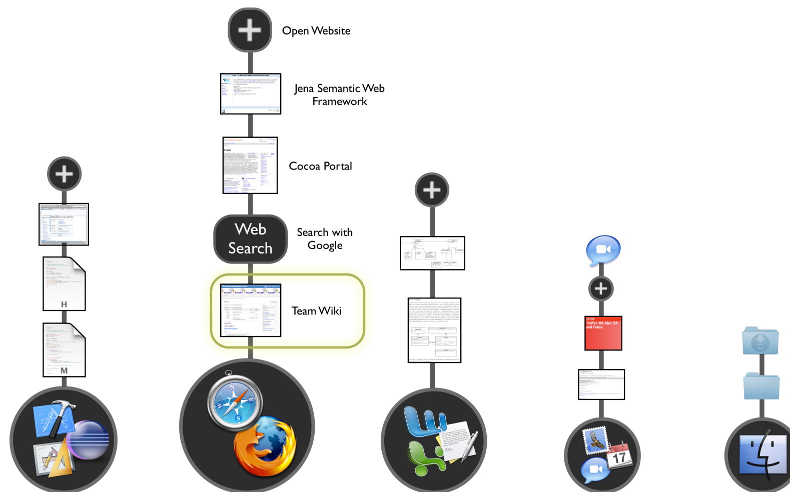

Fig. 1.3: Switch! uses interaction data to recommend the artifact needed next

Contexts. The circumstances in which the developer is performing an interaction, such as the task a developer is working on or the issue being encountered.

1.3.1 Interactions

Interaction data is typically recorded as a stream of interactions, or interaction events. Each interaction event denotes a single interaction a developer performs on an artifact using a tool. For example, a developer might open a source code entity using an editor, run a specific test case within a testing tool, edit a requirement using a text editor, or change a release plan using an issue management system. For this chapter, we focus on interactions that are observable on a developer's computer.

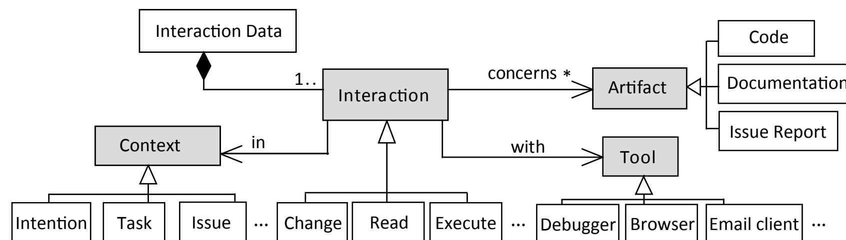

Fig. 1.4: The main concepts of interaction data

NOTE: This is the authors' version of the work.
Springer holds the copyright for the original publication, which can be accesses via it web site
http://www.springer.com/computer/swe/book/978-3-642-45134-8

However, a more general definition could capture any interaction a developer has with any virtual or physical artifact, including calling a customers over the phone, taking notes on a piece of paper, or drawing a model on a whiteboard.

In the following we introduce common types of interaction events. This list is by no means exhaustive. It is rather open-ended and only intends to give an idea of the possibilities.

Select. A select interaction refers to the explicit selection of an artifact by a developer, such as selecting a specific class in the IDE, a particular issue report in the issue management system, or a particular tab in the web browser.

Edit. An edit refers to the creation, removal, or modification of an artifact by a developer. While on a fine granular level each individual keystroke might be recorded as a separate event, on a higher granular level, an edit event might represent the whole modification (e.g., to a source code element, a part of a document, or a package).

Read. A read interaction represents the developer acquiring the information in an artifact. This typically involves selecting the artifact and scrolling its content.

Open and close. An open or a close event refers to the explicit opening or closing of an artifact, such as opening a file from disk, the attachment of an email, or accessing a website. Open and close can also concern a tool such as opening the email client or the web browser.

Reference. A reference event represents the indirect usage of a specific artifact, e.g., through importing a library or calling a method in a source code.

IDE command. IDEs offer a variety of commands to the users, each with a specific semantic and functionality. A command interaction refers to a developer executing one of these commands in the IDE. The exact set of possible commands depends on the specific IDE used and its plugins. Commands are typically grouped into user interface menus, including:

- **Debugging.** A debugging command refers to specific debug actions, such as "set breakpoint", "step over", "inspect", "change variable"....
- **Versioning.** These refer to specific commands in the versioning system, e.g., "checkin", "checkout", "synchronize"...
- **Issue tracking.** These refer to commands in the issue tracking system such as "create a bug report", "close a bug report", "add a comment"...
- **Refactoring.** These commands include common refactoring operations such as "rename", "move method", "extract method"...
- **Testing.** These commands refer to the running and managing of test cases.

Text input. A text input event refers to a user entering text into a specific field to perform a command. Examples are web searches, IDE searches, or rename commands. These interactions have a different semantic than edit events and are therefore often treated differently.

Use. This is a general type of interaction, which might, e.g., concern tools or application, such as using a debugger, or using an email client.

Other. Finally, there are commands specific to applications other than an IDE, such as starting a chat session, sending an email, or playing a video file. These are

NOTE: This is the authors' version of the work.
Springer holds the copyright for the original publication, which can be accesses via it web site
http://www.springer.com/computer/swe/book/978-3-642-45134-8

similar to the IDE commands in the sense that they are an open set of actions that may vary from user to user, from platform to platform, and even from application version to application version.

Interaction events depict a certain interest of a developer in the concerned artifact. Depending on the type, an event might indicate a different degree of interest in the artifact. For example, an edit event might indicate a higher interest in a source code method than a selection of the same method [14].

1.3.2 Artifacts

Developer interactions affect various types of artifacts. The artifacts range from source code entities (such as classes and methods), to models, documentation pages, and emails. Artifacts vary in their level of granularity. For instance, a developer might open a whole class file in a code editor, than select a single method therein, then change one call to another method, and then interact with a code navigation tool to navigate to the called method.

One of the most common types of artifact used in RSSEs are source code artifacts, i.e., the entities that a software system is composed of. Recorded artifacts may range from packages or binaries at a higher-level, down to files, classes, modules, methods, procedures, attributes, variables, and even individual expressions or statements, depending on the purpose of recommendations. For instance, while Mylyn stops at the method and variable level, OCompletion distinguishes interaction events at the code statement level.

In addition to source code entities, recommendation systems might also monitor developer interactions with other project artifacts, such as bug reports, test cases, documentation, build and configuration files, models of the system, and requirement specifications. More and more web sites are also considered as very important artifacts in software development and the interaction with them reveal information about the developers' interests, their intent, or their problems encountered. These web sites range from online API documentation, over question and answer web sites, such as Stack Overflow, to the results of web or code searches. From the perspective of developers, we can distinguish between two types of artifacts: documents that can be read and edited by the people (such as text documents, images, or videos) and binaries that can be executed or used.

1.3.3 Tools

Tools are the software applications developers interact with to perform their work. These tools might run as a separate process in the operating system or as a specific plugin in the integrated development environment, such as plugins for source code analysis, for version control systems, or bug tracking systems. Tools might also

NOTE: This is the authors' version of the work.
Springer holds the copyright for the original publication, which can be accesses via it web site
http://www.springer.com/computer/swe/book/978-3-642-45134-8

be remote applications, which are accessed, e.g., via a web browser or a console. An example of such a remote tool is the Bugzilla issue tracking system, which is typically used through the web browser. External tools that are often outside of the IDE but still important to a developer's work include web browsers, email clients, instant messaging, video-conferencing platforms, design tools, or requirement tools.

Typically, a specific artifact type is accessed and maintained by a specific tool type [17]. For example, editors and debuggers are used to manipulate source code. Diagramming and visualization tools are used to create and maintain models. Issue tracking systems are used to gather and process issue reports. Finally, email clients and chat programs are used to share information and coordinate work. Some recommendation systems collects interaction data from a single tool. A more sophisticated recommendation system should take into account the variety of tools that developers use.

1.3.4 Context

Context is a loosely defined term and can refer to manifold concepts, such as artifacts related to the one the developer is interacting with all the way to the mood of the developer. We define the context of a developer's interaction as to the conditions or circumstances in which the developer interacts with an artifact. Context allows to better understand why certain interactions happened. For instance, a text input interaction might be part of a refactoring command which might be part of a task to clean up the code.

For any interaction there is some context, e.g., the preceding interactions that are relevant for the current interaction or some more abstract goal or intention of the developer. This context can be used to interpret developers' interactions and provide better recommendations. However, not all types of context are easily *observable*. For instance, if a developer accidentally hits a keyboard button when trying to catch a fly, the reason for the accidental edit would in most cases not be recorded and thus relevant context will be missing.

The context of an interaction can be on multiple levels of granularity, such as a developer's interaction preceding the event, a higher-level activity, or the more abstract task a developer is working on. Typically, higher-level context information is interpreted by processing interaction data, as described in Sect. 1.5. Common kinds of context which are of interest to recommendation systems are (a) the concrete *tasks* or *intentions* the developer is having (e.g., fixing a specific bug) and (b) a recurrent activity or situation in the developer's work (e.g., encountering a problem versus applying a solution).

Tasks and intentions. In software engineering a task is commonly defined as an atomic and well-defined work assignment for a project participant or a team [14**?**]. A task includes a description and an assignee; it typically includes a duration and time-frame. Tasks describe what developers should do. An example of a task is

NOTE: This is the authors' version of the work.
Springer holds the copyright for the original publication, which can be accesses via it web site
http://www.springer.com/computer/swe/book/978-3-642-45134-8

"Task #123: implement the XML Export Feature" assigned to Alice or "Weekly integration test for web server" assigned to Bob. Recommendation systems like Mylyn [14] associate every interaction event with a specific task, which has been previously activated by the developer (to express that this task is being worked on). The interaction data associated to a task is called task context. It is used to provide recommendations tightened to that task—e.g., the most relevant artifacts, the related bug reports, or the people who should work on the task.

A significant amount of developers' work is rather informal and thus not always associated to specific predefined tasks. Maalej [17] previously found that about half of developers interactions are not related to specific tasks. This kind of context is called intention [18]. It refers to the mental state that underpins the user interactions but is not necessarily stated explicitly, e.g., "explore a new API", "assist a colleague" or "fix the discovered but unreported bug". There are several approaches that aims at detecting and describing the intention of the user [e.g., 18, 36]. However, most of them are still exploratory and experimental.

Activities and situations. Activities are coarser-grained types of interactions, typically referring to a class or a set of interaction events. Example of activities include navigating, coding, testing, debugging, specifying, planning, documenting, designing, amongst others. An activity typically includes more than one interaction event and last for at least a few minutes. Activities are often part of a task. For instance, in fixing a bug, developers might navigate through the code; once they find the right code, they make changes to it and then test it. Activities can also be more coarse-grained, for instance, if two tasks are about *documenting* two parts of a user interface. Activities reveal a recurrent development situations with well defined semantics. They are interesting for recommendation systems to suggest items typically relevant in such situations.

Other types of situations include the "phases" of a task or the "states" in a mental model of the developer. For example, a typical change task includes an initiation phase, a concept location phase, and an impact analysis phase [29]. Phases might also be oriented towards a problem-solution cycle, such as locate cause, search solution, and test solution [34]. The better these concepts can be automatically inferred from a developer's interaction, the more precise the recommendations based on interaction data might become and the broader the approaches might become.

1.3.5 Interaction Granularity

Interaction data might include various levels of abstractions, often called *granularity levels* or granularity spectrum [33]. For instance, to perform a refactoring a developer might have to enter text or edit and select parts of the code in between. In this case, the refactoring event represents a higher-level of granularity than the edit and select events.

NOTE: This is the authors' version of the work.
Springer holds the copyright for the original publication, which can be accesses via it web site
http://www.springer.com/computer/swe/book/978-3-642-45134-8

All types of interaction data including interaction events, artifacts, tools, and contexts present different levels of granularity. For instance, an interaction event might be a step-in or a debugging event. An artifact involved in the interaction might be a package, a class, a method, a section, or even a single line in a document. A tool might be the whole IDE such as Eclipse or a single plugin in the IDE. Finally, a context might be the task activated by the developer, a certain release, or the current project phase. Typically, interaction data with a low level of granularity can be collected programmatically, while interaction data with a higher level of granularity needs to be inferred by processing the low level data. One way to more formally describe the granularity levels of interaction data is to use ontologies [e.g., 21].

From the perspective of a developer, interactions with input devices such as mouse clicks and keystrokes represent the lowest level of granularity. An interaction with a single widget of the user interface, such as entering text in a text field or clicking a button, represents a higher level of granularity. A single widget interaction consists of multiple interactions with the hardware periphery such as multiple key presses on the keyboard to enter text in a text field. More precise information about the interaction can be derived at the widget level, since clicked widgets typically are associated with a name and a given purpose. For example, a mouse click can now be identified as pressing a button to create a project or the selection of a window part. The next level of granularity is an aggregation of several single widget interactions, e.g, creation of a new project using an IDE wizard. Finally, several single-widget or multiple-widget interactions can represent a user activity such as refactoring the code, which in turn is a step of a task.

1.4 Collecting Interaction Data

Researchers and tool vendors have proposed several approaches to collect developers' interaction data, including:

- the Eclipse usage data collector,
- the Mylyn monitor [14],
- SpyWare, to record fine-grained code changes [31], and
- Teamweaver, to record interactions with tools inside and outside the IDE [19].

In the following, we discuss the general procedure underlying these approaches. While different recommendation systems require different types and granularity levels on developers' interactions, they are generally all comprised of three parts: a monitor with a set of listeners (also called sensors), a component to generate interaction events, and a component to log these events and enable their processing. Figure 1.5 shows the general procedure for collecting interaction data.

NOTE: This is the authors' version of the work.
Springer holds the copyright for the original publication, which can be accesses via it web site
http://www.springer.com/computer/swe/book/978-3-642-45134-8

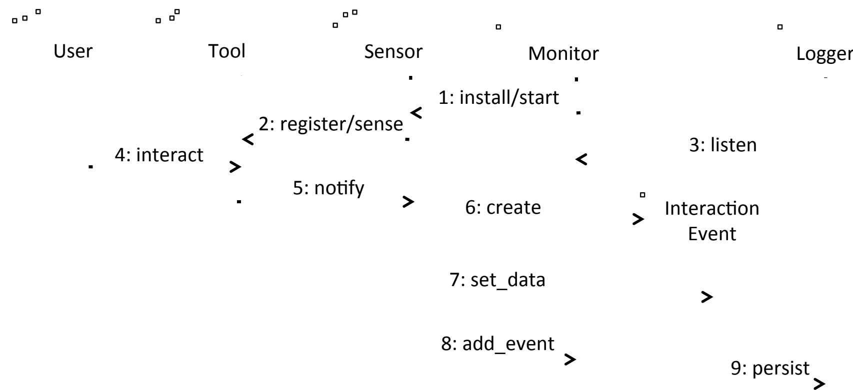

Fig. 1.5: Main components for collecting interaction data with a simplified flow

1.4.1 Monitoring Developers' Interactions

The monitoring component varies depending on the tools, the artifacts, and the interactions which should be collected. While some approaches might, for example, only monitor coarse grained IDE actions, others monitor every single keystroke. In general, the monitoring component manages several listeners, which are also called sensors [19]. These sensors instrument the work environment, such as the tool, the IDE, or the operating system where the relevant events are triggered. The sensors continuously monitor their targets and whenever a new and relevant interaction happens, they collect the necessary information such as the name of artifacts concerned, its type, or the duration of the interaction.

The implementation of the sensors is often specific to the type and the technology of the tool being instrumented. For example sensors might operate on the operating systems or the virtual machines level. These offer interfaces to listen to particular types of events such as opening a file with a tool. The Microsoft Windows operating system, for example, provides hooks, while OS X offers an Apple Script interface to implement such functionality. Also the Java virtual machine and the Eclipse runtime environment provide libraries to observe the interactions with the user interface elements. In addition, program-monitoring and tracing frameworks (such as DTrace or SystemTap) are deeply integrated into the operating system and execution environments with the purpose of tracing program execution that can provide further information on interactions. Sensors might also operate on the application level. This is particularly convenient if the application provides means for installing the sensors as plugins. Sensors should generally provide an interface to install/uninstall and active/deactivate them. This allow the users to have the full control and reduces the privacy concerns for collecting interaction data.

NOTE: This is the authors' version of the work.
Springer holds the copyright for the original publication, which can be accesses via it web site
http://www.springer.com/computer/swe/book/978-3-642-45134-8

1.4.2 Generating Interaction Events

Once the sensor has captured an interaction, it generates an interaction event. Most commonly, an interaction event is composed of the following information:

- the type of the event, e.g., a select event or an edit event;
- the timestamp denoting when the event occurred;
- the duration or end time of the event;
- the artifact concerned by the interaction, e.g., `setName()` or "Issue #234"; and
- the type of the artifact concerned by the interaction, e.g., a method or a bug report;
- the tool used to perform this interaction.

An example of an XML representation of interaction data is shown in Fig. 1.6. In addition, the generated event might contain information on the context, such as the task of which the event is a part. All of this information is gathered by the component and aggregated in a newly generated interaction event.

1.4.3 Logging Interaction Data

The final component is a logger that persists the generated interaction events. Different approaches use different techniques for the logging with respect to compression and the segmentation of the data. Mylyn, for instance, collects a set of interaction events and compresses them by collapsing similar events into one. This approach minimizes the use of disk space and write operations. However, it makes it difficult to recover the exact sequence of interaction events (see Sect. 1.5.4). In addition, Mylyn logs the interaction data related to the task a developer is working on in a so called task context. Task contexts then provide a means to easily recover all interactions for a specific task. Other approaches log interaction events sequentially into a file without compressing or segmenting it in any particular way. Additional processing steps applied to the log file later can then also help to recover task boundaries (see Sect. 1.5.1).

```
<pre:e1 rdf:type interaction:JavaElementChange />
<pre:e1 interaction:hasTimeStamp 1222002002 />
<pre:e1 interaction:hasDuration 200 />
<pre:e1 interaction:concerns pre:java?name=myMethod />
<pre:java?name=myMethod rdf:type artefact:Method />
<pre:java?name=myMethod artefact:partOf pre:java?name=myProject.
    myClass />
```

Fig. 1.6: Example of an XML representation of interaction data

NOTE: This is the authors' version of the work.
Springer holds the copyright for the original publication, which can be accesses via it web site
http://www.springer.com/computer/swe/book/978-3-642-45134-8

The logging component might be on the developer's machine [14] or on a server [27]. Typically, the interaction logger provides additional functionally such as the obfuscation of the data to reduce the risks of misusing it or the archiving of the data to reduce its size. Evaluations have shown that, for a tool like Mylyn, the size of a log file for the interaction data of a full workday typically includes 1–10 megabytes of data [14].

1.5 Processing Interaction Data

Interaction data in its raw form is usually not what is needed. For recommendation purposes further processing of the collected data is often necessary. We discuss common data processing approaches and summarize pitfalls, which should be avoided when processing interaction data. Processing interaction data is an active research field. Other approaches might emerge in the future.

1.5.1 Sessionization of Interaction Events

The raw interaction data is typically in form of a stream of events, which might need to be split into individual sessions. We call this process *sessionization* of events. Ideally, the developers will explicitly define the start and end of a session, e.g., to indicate what task they are working on. For example, Mylyn users can sessionize their work by explicitly activating and deactivating a specific task from the task list. However, since developers typically work on different tasks in parallel and frequently change their focus back and forth [28], they might not be willing to invest extra effort to indicate when they start and finish a specific work, or might simply forget to do this. The problem of sessionization is also present in conventional repository mining. It has been shown that developers occasionally perform several tasks in one commit, as discussed by Herzig and Zeller [11] in Chap. **??**.

To identify sessions retrospectively, there are several approaches or heuristics that might be considered:

- Several empirical studies have shown that work sessions with a particular goal in mind typically last between 30 and 90 minutes [20].
- Period of sustained inactivity, i.e., consecutive events that are separated by large amounts of time, can be used to split the stream of events into sessions. The threshold of one hour has shown good results, i.e., less than a lunch break or a meeting, but more than a coffee break.
- Shorter interruptions can be detected as well, and processed accordingly. The process is similar, only the threshold retained is lower (e.g., 5 minutes).
- Specific events represent strong indicators for switching the work session, such as a committing event, starting a new tool, or viewing the task list or the issue management system.

NOTE: This is the authors' version of the work.
Springer holds the copyright for the original publication, which can be accesses via it web site
http://www.springer.com/computer/swe/book/978-3-642-45134-8

- Individual work sessions can be focused on one task, or be composed of several tasks. Several sessionization algorithms defined in the literature are based on time information and the artifacts that constitute the task [e.g., 28, 36, 40].
- Some tasks are too large to be finished in one development session. In this case, sessions that involve related artifacts may be linked together, forming a "macro-session" if the entities in common between both sessions are above a certain threshold.

1.5.2 Filtering of Events

Depending on the goal of the recommendation system, some events might be undesirable and considered as noise. These events must be detected and removed. Examples of these events include the following.

- "Transient changes" are changes that do not survive a development session, for instance, when a developer inserts debugging statements in the code in order to find a bug, and removes them once the bug is fixed. Other examples are errors in the code (incorrect method calls) that are corrected later in the session. Transient changes strongly depend on the usage scenario of the interaction data. These events might be irrelevant in specific cases but relevant in other situations.
- Events that are not originating from the developer but rather from the tools that the developer is using may need to be treated separately. For instance, changes occurring from a refactoring tool do not represent developer interactions. Using these changes to evaluate the performance of code completion algorithms would misrepresent them. Changes performed by tools are performed much faster than changes performed by developers, making it possible to mark them as such (e.g., a rename refactoring will change the name of a method, and update all references to it in rapid succession, on the order of milliseconds).
- "Bulk events", e.g., events of type "selection", may originate from selections of many artifacts in the IDE. If the developer selects all the classes in a given package, they may be marked as individually selected, yielding a very large number of selection events in a short time.

Filtering is performed when the spurious events are deemed to be irrelevant for the task at hand. In that case, events are simply removed from the stream of events.

1.5.3 Aggregation of Events and Inference of Context Information

RSSEs might also aggregate interaction events to infer a higher level of granularity (see Sect. 1.3.5). A typical purpose is to infer the current task or situation of the developer from the low level interaction events. We distinguish between three ma-

NOTE: This is the authors' version of the work.
Springer holds the copyright for the original publication, which can be accesses via it web site
http://www.springer.com/computer/swe/book/978-3-642-45134-8

jor aggregation approaches: semantic approaches, heuristic-based approaches, and probabilistic approaches (i.e., using machine learning).

Semantic approaches use a type hierarchy (i.e., a specific taxonomy) to aggregate interactions or artifacts to a higher-level type in the hierarchy [19, 24]. For instance, observed "step-in" and "step-out" events have "debugging" as the common higher-level type and can thus be aggregated into a more general debugging interaction. Similarly, a "method" and a "class" are subtypes of the higher-level type "code". The two events "edit the method X" and "edit the class Y" can be aggregated to "edit the code X and Y". The interaction and artifact taxonomies might also define cross-relationships to allow for reasoning. For example, the interaction type "implement" might be associated with the artifact type "method", whereas the type "specify" is associated with the artifact type "class". From observing an "implement" event that concerns an artifact of type "class", we can infer that the event is of type "specify". The main disadvantage of taxonomy-based approaches is the maintenance of the taxonomy, which is difficult and time consuming. Moreover, interaction types can have multiple higher-level types (i.e., multiple inheritance). It is thus nontrivial to navigate the taxonomy up and down to select the right type.

Similar semantic approaches without taxonomies aggregate interactions concerning the same artifact or the artifacts concerned by the same type of interaction. For instance, in Mylyn multiple events concerning the same code entity are sometimes represented as an aggregated event with a start date, an end date, and a number of events (i.e., the total number of events between the first and the last, both included). Similarly, the events originating from the clicks on items of the same menu (e.g., view, edit, or debug) can also be aggregated to an event describing that menu. Finally, all edit events that concerns methods of the same class, can be aggregated to an edit of that class.

Heuristic-based approaches typically use assumptions and metrics to aggregate events and infer context. For instance, the degree of interest model underlying Mylyn aggregates all the events concerning an artifact and compute an interest value that is updated over time based on the recency of the interaction [14]. Similarly, the defect prediction approach of Lee et al. [16] computes a variety of metrics over session data, aggregating interactions in one development session. These metrics then serve as input to a metric-based defect prediction model. Likewise, Robbes and Lanza [30] classification of development sessions to one of 5 categories is based on metrics. Finally, Ying and Robillard [39] suggest to use the interaction style (i.e., the distribution over time) of the edit events to determine whether the developer is working on an enhancement task, minor, or major bugs fixes. Development sessions consisting of Edit-Last events are most likely enhancement tasks. Edit-First interaction style is most likely an indicator for minor bug fixes while Edit-Throughout is an indicator for major bug fixes. In general, the *duration*, the *recency*, the *type* and the *frequency* of interaction events can reveal "important" context. Heuristics based on these features can be used to label sets of events in a developer session.

Finally, *probabilistic approaches* might use data mining and machine learning algorithms (such as those introduced in Chapter **??** [**?**]) to aggregate interaction data. Generally, these approaches try to identify in the interaction history recurrent

NOTE: This is the authors' version of the work.
Springer holds the copyright for the original publication, which can be accesses via it web site
http://www.springer.com/computer/swe/book/978-3-642-45134-8

patterns, which characterize specific situations. When these patterns are observed in future interaction data, the system predicts the situation with a certain probability. For example, RSSEs might define the set of development situations to be inferred. In a training phase, the system learns the probability to move between two situations when certain interaction events occur. This can be, for example, to move from a testing to a debugging situation when a "read error message" event occurs. Later, the RSSE infer the current situation based on the observed events. Roehm and Maalej [34] suggested a similar approach using a hidden Markov model. Other machine learning approaches such as time series analysis, or frequent itemset mining might also be used.

1.5.4 Pitfalls when Processing Interaction Data

Processing the interaction data can make it more useful to the recommendation task at hand, but certain pitfalls have to be kept in mind:

Over-processing. Each processing step may introduce noise. Algorithms detecting patterns in the data rarely have perfect precision and recall, especially algorithms that rely on thresholds: slight changes to the threshold return different results for borderline cases. Therefore, composing processing steps can potentially compound the inherent imprecisions of each algorithm. We recommend double-checking the results with care and if possible using a semi-automatic approach that corrects wrong processing results.

Destructive operations. When aggregating events, we recommend keeping the original data intact as much as possible, as it is hard to predict what information will be needed. When the Spyware tool detects a refactoring operation, it creates an aggregated refactoring event, but keeps the actual changes as a part of this event in case a future RSSEs need to consult this data. Mylyn's aggregation of events loses detail on the specific interactions so that only the start and end time of a sequence of interaction is known, but the timestamps of intermediate events is removed. This makes it difficult for other approaches that need a full sequential list of interaction events to use the Mylyn monitor. As a workaround, Ying and Robillard [39] assumed that the intermediate events were equally distributed between the first and the last timestamp.

Tool limitations. Data recorded about what the developer is doing may still be inaccurate. Each interaction data collection tool has issues that should be known to avoid false interpretations (e.g., that a very large number of artifacts are manually inspected by developers in a large amount of time). When using existing monitoring tools, we recommend to carefully review the data produced by the tool, in order to have a clear understanding of what kind of events are producing what kind of data. If possible, the data should be preprocessed to attenuate data quality issues.

Developer inactivity. Developer inactivity is hard to assess, as it may simply be due to missing interaction data. For instance, the IDE sensor may not register

NOTE: This is the authors' version of the work.
Springer holds the copyright for the original publication, which can be accesses via it web site
http://www.springer.com/computer/swe/book/978-3-642-45134-8

any activity because the developer is browsing the web or because the developer is carefully reading a visible piece of code on the screen. Treating these moments as breaks in the work may introduce imprecisions.

1.6 Using Interaction Data

In this section, we discuss scenarios where interaction data can be used to provide recommendations to developers. We focus on approaches to increase developers' productivity and to support awareness and collaboration amongst development teams.

1.6.1 Productivity

Interaction data can be used to improve developers' productivity. Current approaches can be grouped into four main scenarios: (1) reducing information overload and helping developers to focus, (2) recommending a particular piece of information that is needed in the current task and that will help in satisfying developers' information needs, (3) suggesting a relevant source code, and (4) predicting a particular project metric, such as the bug-proneness of a module.

Mylyn [14] aims to *reduce information overload* for developers by optimizing the user interface of the Eclipse IDE (see Sect. 1.2). The core idea is that only a subset of all code artifacts in large software projects is relevant for working on a given task. Thus, Mylyn hides or blurs code artifacts, which are less relevant. The tool collects all interaction events that a developer performs while working on a task and calculates a "degree-of-interest" (DOI) value for each code element. This value reflects a certain interest level of the developer in this code element. The DOI assumes that the more frequent and more recent an element is interacted with, the more interesting it is to the developer for the current task at hand. The DOI value is then interpreted to visually indicate task-related files in the IDE. Kersten and Murphy [14] evaluated the influence of Mylyn on the personal productivity of developers by calculating the edit ratio of 16 subjects with and without using Mylyn. The edit ratio is the relative amount of edit versus select interactions for a certain period of time. The authors found that Mylyn significantly increased the edit ratios of their subjects, on average by 50%. Mylyn is already part of the most common distribution of the Eclipse IDE and being used by a large population of software developers.

Reverb [35] is a tool that recommends *web sites including relevant information* for developers based on the code they are currently editing. The tool assumes that people often revisit the same web sites and takes into account two kinds of interactions: a developer's web browser history and the editor window the user is currently interacting with in the IDE. Any website a developer visits for at least 5 seconds

NOTE: This is the authors' version of the work.
Springer holds the copyright for the original publication, which can be accesses via it web site
http://www.springer.com/computer/swe/book/978-3-642-45134-8

is considered relevant and therefore indexed. When a developer interacts with the Java code editor in the IDE, Reverb extracts the Abstract Syntax Tree elements from the currently visible source code in the editor, and queries the developer's browser history with these code elements. An evaluation showed that 51% of code-related revisits can be predicted by Reverb, which reduces the time developers need to find and open the website needed. Murphy-Hill et al. [26] introduced a similar approach, based on very fine-grained interaction data for improving developers' fluency by recommending specific commands in the IDE which might save time and of which the developer might not be aware of.

Since *code completion tools* are commonly used by developers, we can assume that increasing their accuracy will increase the productivity of the developers. Robbes and Lanza [32] evaluated using fine-grained change interactions to improve the accuracy of code completion tools. The authors used a large data set including a list of fine-grained changes performed by developers while working on tasks in their IDE. From this data set, the insertions of method calls and class names are identified. When detecting such an insertion, the code completion engine is simulated, as if the developer was asking for a code completion. The code completion engine returns an ordered list of recommendations for the completed identifier. The proposals of the completion engine are compared with the actual identifier which is included in the sequence of pre-recorded changes. The authors found that the default algorithms ordered their recommendations alphabetically, which yielded very poor accuracy. Ordering the suggestions based on their usage recency gave much better results, increasing the score fivefold.

Finally, Lee et al. [16] investigated developers' interaction history for *defect prediction*. Based on select and edit interaction events, they define 56 micro-interaction patterns. In an experiment they compared the predictive power for regression and classification of these patterns against source code and history metrics. The authors show that micro-interaction patterns can improve upon existing defect prediction models based on source code or history metrics. For example, the pattern "NumLowDOIEdit" representing the number of edit events with a low DOI value, i.e., editing a code element that one has not interacted with a lot before, has the highest power to predict a defect.

1.6.2 Awareness and Collaboration

"Awareness is the understanding of the activities of others, which provides a context for the own activity" [4]. Interaction data is being used to provide awareness to developers, mainly answering questions such as "who is working on what". Approaches for awareness vary depending on the granularity of the interaction data (from very fine-grained code edits to more coarse-grained file changes), the type of artifacts the awareness is provided for (such as project code or work items), and the kind of information visualizations being provided.

NOTE: This is the authors' version of the work.
Springer holds the copyright for the original publication, which can be accesses via it web site
http://www.springer.com/computer/swe/book/978-3-642-45134-8

To provide team awareness and avoid conflicts, FastDash [1] visualizes where people are interacting with files in a project. This approach collects two kinds of data: (a) active file actions that are based on developers' interactions with the Visual Studio IDE, such as opening, editing, or debugging files and (b) source repository actions, such as which files are checked out by whom. The interaction data is collected on a server and visualized in a dashboard, which presents the project files in a tree map and annotates files with which developers are currently interacting.

Seesoft [5] is a similar recommendation tool for creating awareness in software projects. It colors each line of code in the IDE according to the recency of its last change: the recently changed lines are colored in red, older lines in dark blue. The interaction used in this approach is limited to the changes that people made to the code in the source code repository.

More recently, Fritz et al. [6] suggested the Degree-of-knowledge model to recommend expert developers for parts of the code. This approach uses authorship and interaction data to characterize a developer's knowledge of the source code. The degree-of-knowledge model predicts for each code element a developer who should know most about it. In addition, Fritz et al. showed how this model could be used to recommend bug reports that might be of interest to a developer.

1.7 Challenges and Future Directions

The field of collecting and processing interaction data for the purpose of recommendation is relatively new. Despite recent advances, there are scientific and technical challenges as well as promising usage scenarios left for future research.

1.7.1 Challenges

Efficient, integrated, non-intrusive instrumentation. The first step in implementing recommendation systems that use interaction data is instrumenting the work environments of developers and (in particular cases) end users. To this end, a question about the efficiency and intrusiveness of data collection arise, i.e., how data can be collected without disturbing the user's workflow. Moreover, the integration of the context monitoring into heterogeneous tools and applications poses an additional engineering challenge on how can various workplaces (including heterogeneous tools, information, and activities) be instrumented and observed. This leads to the question whether such instrumentation can be *systematically* integrated into (or offered by) underlying frameworks such as graphical user interfaces, accessibility libraries, operating systems, middleware, and execution environments.

Representation of interaction data. The usefulness of interaction data depends on the specific scenario, for which it is used. In some cases, fine-grained interac-

NOTE: This is the authors' version of the work.
Springer holds the copyright for the original publication, which can be accesses via it web site
http://www.springer.com/computer/swe/book/978-3-642-45134-8

tion data and artifacts are needed. In other cases, higher-level interaction events and context information are more useful. This makes the general modeling and representation of interaction data for recommendation systems a difficult endeavor.

The representation of interaction data and its context has more complicated requirements than the representation of simple logs, e.g., a web server traffic log. Interaction data should be represented efficiently and allow for eventually unknown queries and processing. The following questions arise: How can we represent interaction data to enable reasoning, semantic interpretation, and querying? Which representation allow for a flexible and accurate comparison of similar contexts? What should be observed and what not? What should be rather processed?

Sessionization of interaction events. A central research issue for using interaction data in RSSEs and more generally building context aware systems is the sessionization of the event stream (see Sect. 1.5). Sessionization is a complex problem since people frequently switch their focus and intentions. Interruptions and new thoughts lead to context overlaps. Sessionization packages interaction data and context information that belongs together. The question is thus: How can we precisely sessionize interaction data? How can a context switch be detected? How can we automatically detect and classify users' intentions to well-defined types, e.g., based on the meaning of interaction events (such as testing, debugging, or releasing context)?

Context prediction and comparison. Raw interaction data includes a lot of noise because of the large amount of *potentially useful* information that can be collected. Interaction data should be processed and aggregated, its information ranked, and new knowledge about the context derived out of it. The following questions arise: How can we aggregate interaction data for different levels of granularity (different situations require different levels of details)? How can short-term context such as the current intention and long-term context such as the profile and preference of the developers be predicted based on observed interaction? How can aggregated context be compared, and decomposed if more details are needed?

Privacy protection. Recommendation systems based on interaction data collect numerous, possibly sensitive information about the user. This raises privacy concerns, since information can be abused, misinterpreted, or even sold for marketing agencies. For example the interaction data of a developer can be misused by the employer to measure and compare the productivity of the developers. The questions are: What are acceptable trade-offs for RSSE users? How can we protect users' privacy while collecting their sensitive information? How can we ensure the principle minimality, i.e., ensure to collect only the minimally required set of information? The more difficult question is: how can we ensure that anonymized interaction data cannot reveal more sensitive information the future e.g., if combined with other data collected about the user from different sources (e.g., multiple RSSEs)?

NOTE: This is the authors' version of the work.
Springer holds the copyright for the original publication, which can be accesses via it web site
http://www.springer.com/computer/swe/book/978-3-642-45134-8

1.7.2 Future Scenarios

Proactive knowledge capturing, sharing, and access. Interaction data includes useful knowledge, e.g., on how a problem has been solved by a developer [10]. If filtered and aggregated accurately such data will represent experience description, which can be populated in wikis or used to recommend solution alternatives when similar problems are encountered. It is useful, e.g., to capture information on where developers looked for help while having similar bugs, or what did they do to fix it [9]. Similarly, reuse scenarios such as component integration or API reuse require significant background knowledge [8]. In such scenarios useful information includes how other developers proceeded in the reuse, how they instantiated a particular API, where they looked for help, and where they started. Such experiences are typically lost or scattered across private documents.

Knowledge sharing can be made more precise and efficient by supporting the role of knowledge producers [9]. Future recommendation systems can actively capture the experiences of developers by observing interaction data and encouraging them to share certain information with certain team members [10], e.g., asking to share a web page which a developer extensively used to solve a certain problem.

Future recommendation systems can also automatically identify links between artifacts, e.g., source code created and documentation useful to understand it. For example, while implementing a change request, a developer might check the issue tracker, read the customer's email, browse a forum discussion, reuse a new library, and change several pieces of source code. The ticket, the discussion, the email, and the library can be linked and later traced to the changes and resulting versions. Linking changes to their context enables developers to trace these changes and understand them in the future [17]. These links simplify the information retrieval based on available information (e.g., the customer's email instead of the version number).

User involvement and continuous requirements engineering. In modern product development, the user feedback and the user acceptance of the product are essential for market success [25]. Current requirements engineering practices are characterized by a communication gap between users and developers [22]. The context that underlies the user feedback is either gathered asynchronously or submitted with the wrong level of detail.

Observing the interaction data of users can make user feedback a first order concern in software engineering. Software systems would observe how their users use certain features, their problem situations, their workplaces, and workflows. Such information facilitates continuous, semi-automatic, communication between users and developers. Problems or bugs will be reproduced and understood faster; wrong requirements corrected and elaborated remotely. This increases the quality of user input and the efficiency of requirements and maintenance processes. This would also enable users to bring their innovations and become a "collaborator" in the project [23], as their interaction data can be used to systematically evaluate particular software features (e.g., in a new release), how they are used, and why they are used in that way—promoting a deeper understanding of the user's needs.

NOTE: This is the authors' version of the work.
Springer holds the copyright for the original publication, which can be accesses via it web site
http://www.springer.com/computer/swe/book/978-3-642-45134-8

1.8 Conclusion

In software engineering interaction data captures the *interaction* of developers with *tools* to perform specific work and includes information about the *artifacts* being concerned by the interaction. Interaction data might also contain information about the *context* in which the interaction occurred (e.g., the task at hand, the intention in mind, or the problem being encountered). Current RSSEs using interaction data focus on increasing developer's productivity by, for instance, filtering irrelevant information or predicting reusable code, as well on creating awareness by, for instance, showing who is working on which artifact or recommending experts.

In this chapter, we discussed means to represent and collect interaction data for recommendation systems. Collecting this data typically requires installing monitors and sensors that listen to user interactions in the target applications and thereof create a log of interaction events. Furthermore, we discussed some of the major goals and challenges of processing interaction data, including the filtering of noise, the aggregation of events, the sessionization of event steams, and the inference of higher-level context. Although there have been considerable advances in the field in past years, there are still many open challenges for using interaction data in recommendation systems. These challenges include the efficient instrumentation and privacy concerns for interaction data. Potentially useful future scenarios include the extraction of knowledge and experience from the interaction data and the collection and processing of usage data software at runtime.

Acknowledgements We are grateful to Tobias Roehm, Zardosht Hodaie, and the reviewers for their constructive feedback on this chapter. We also thank Bernd Brügge and Bashar Nuseibeh for the comments on early versions of this work. The first author is supported by the EU research projects MUSES (grant FP7-318508).

References

1. Biehl, J.T., Czerwinski, M., Smith, G., Robertson, G.G.: FASTDash: A visual dashboard for fostering awareness in software teams. In: Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems, pp. 1313–1322 (2007). DOI 10.1145/1240624.1240823
2. Bruegge, B., Dutoit, A.: Object-Oriented Software Engineering. 3rd edn. Prentice Hall (2009)
3. DeLine, R., Czerwinski, M., Robertson, G.G.: Easing program comprehension by sharing navigation data. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 241–248 (2005). DOI 10.1109/VLHCC.2005.32
4. Dourish, P., Bellotti, V.: Awareness and coordination in shared workspaces. In: Proceedings of the ACM Conference on Computer Supported Cooperative Work, pp. 107–114 (1992). DOI 10.1145/143457.143468
5. Eick, S.G., Steffen, J.L., Sumner Jr., E.E.: Seesoft: A tool for visualizing line oriented software statistics. IEEE Transactions on Software Engineering **18**(11), 957–968 (1992). DOI 10.1109/32.177365

NOTE: This is the authors' version of the work.
Springer holds the copyright for the original publication, which can be accesses via it web site
http://www.springer.com/computer/swe/book/978-3-642-45134-8

6. Fritz, T., Ou, J., Murphy, G.C., Murphy-Hill, E.: A degree-of-knowledge model to capture source code familiarity. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, vol. 1, pp. 385–394 (2010). DOI 10.1145/1806799.1806856
7. Google Official Blog: Personalized search for everyone (2009). URL http://googleblog.blogspot.de/2009/12/personalized-search-for-everyone.html
8. Griss, M.L.: Software reuse: Objects and frameworks are not enough. Tech. Rep. HPL-95-03, Hewlett Packard Laboratories (1995)
9. Happel, H.J.: Social search and need-driven knowledge sharing in Wikis with Woogle. In: Proceedings of the International Symposium on Wikis and Open Collaboration, pp. 13:1–13:10 (2009). DOI 10.1145/1641309.1641329
10. Happel, H.J., Maalej, W.: Potentials and challenges of recommendation systems for software development. In: Proceedings of the International Workshop on Recommendation Systems for Software Engineering, pp. 11–15 (2008). DOI 10.1145/1454247.1454251
11. Herzig, K., Zeller, A.: Mining bug data: A practitioner's guide. In: Robillard, M., Maalej, W., Walker, R.J., Zimmermann, T. (eds.) Recommendation Systems in Software Engineering, Chap. **??**. Springer (2014)
12. Hill, W.C., Hollan, J.D., Wroblewski, D.A., McCandless, T.: Edit wear and read wear. In: Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems, pp. 3–9 (1992). DOI 10.1145/142750.142751
13. Kersten, M., Murphy, G.C.: Mylar: A degree-of-interest model for IDEs. In: Proceedings of the International Conference on Aspect-Oriented Software Deveopment, pp. 159–168 (2005). DOI 10.1145/1052898.1052912
14. Kersten, M., Murphy, G.C.: Using task context to improve programmer productivity. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 1–11 (2006). DOI 10.1145/1181775.1181777
15. Lanza, M., Hattori, L., Guzzi, A.: Supporting collaboration awareness with real-time visualization of development activity. In: Proceedings of the European Conference on Software Maintenance and Reengineering, pp. 202–211 (2010). DOI 10.1109/CSMR.2010.37
16. Lee, T., Nam, J., Han, D., Kim, S., In, H.P.: Micro interaction metrics for defect prediction. In: Proceedings of the European Software Engineering Conference/ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 311–321 (2011). DOI 10.1145/2025113.2025156
17. Maalej, W.: Task-first or context-first?: Tool integration revisited. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 344–355 (2009). DOI 10.1109/ASE.2009.36
18. Maalej, W.: Intention-Based Integration of Software Engineering Tools. Verlag Dr. Hut, München, Germany (2010)
19. Maalej, W., Happel, H.J.: A lightweight approach for knowledge sharing in distributed software teams. In: Proceedings of the International Conference on Practical Aspects of Knowledge Management, *Lecture Notes in Computer Science*, vol. 5345, pp. 14–25 (2008). DOI 10.1007/978-3-540-89447-6_4
20. Maalej, W., Happel, H.J.: From work to word: How do software developers describe their work? In: Proceedings of the International Working Conference on Mining Software Repositories, pp. 121–130 (2009). DOI 10.1109/MSR.2009.5069490
21. Maalej, W., Happel, H.J.: Can development work describe itself? In: Proceedings of the International Working Conference on Mining Software Repositories, pp. 191–200 (2010). DOI 10.1109/MSR.2010.5463344
22. Maalej, W., Happel, H.J., Rashid, A.: When users become collaborators: Towards continuous and context-aware user input. In: Companion to the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 981–990 (2009). DOI 10.1145/1639950.1640068
23. Maalej, W., Pagano, D.: On the socialness of software. In: Proceedings of the IEEE International Conference on Dependable, Autonomic and Secure Computing, pp. 864–871 (2011). DOI 10.1109/DASC.2011.146

NOTE: This is the authors' version of the work.
Springer holds the copyright for the original publication, which can be accesses via it web site
http://www.springer.com/computer/swe/book/978-3-642-45134-8

24. Maalej, W., Sahm, A.: Assisting engineers in switching artifacts by using task semantic and interaction history. In: Proceedings of the International Workshop on Recommendation Systems for Software Engineering, pp. 59–63 (2010). DOI 10.1145/1808920.1808935
25. McKeen, J.D., Guimaraes, T.: Successful strategies for user participation in system development. Journal of Management Information Systems **14**(2), 133–150 (1997)
26. Murphy-Hill, E., Jiresal, R., Murphy, G.C.: Improving software developers' fluency by recommending development environment commands. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 42:1–42:11 (2012). DOI 10.1145/2393596.2393645
27. Pagano, D., Juan, M.A., Bagnato, A., Roehm, T., Bruegge, B., Maalej, W.: FastFix: Monitoring control for remote software maintenance. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 1437–1438 (2012). DOI 10.1109/ICSE.2012.6227076
28. Parnin, C., Rugaber, S.: Resumption strategies for interrupted programming tasks. In: Proceedings of the IEEE International Conference on Program Comprehenension, pp. 80–89 (2009). DOI 10.1109/ICPC.2009.5090030
29. Rajlich, V.: Software Engineering: The Current Practice. CRC Press (2012)
30. Robbes, R., Lanza, M.: Characterizing and understanding development sessions. In: Proceedings of the IEEE International Conference on Program Comprehenension, pp. 155–166 (2007). DOI 10.1109/ICPC.2007.12
31. Robbes, R., Lanza, M.: SpyWare: A change-aware development toolset. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 847–850 (2008). DOI 10.1145/1368088.1368219
32. Robbes, R., Lanza, M.: Improving code completion with program history. Automated Software Engineering: An International Journal **17**(2), 181–212 (2010). DOI 10.1007/s10515-010-0064-x
33. Roehm, T., Gurbanova, N., Bruegge, B., Joubert, C., Maalej, W.: Monitoring user interactions for supporting failure reproduction. In: Proceedings of the IEEE International Conference on Program Comprehenension, pp. 73–82 (2013)
34. Roehm, T., Maalej, W.: Automatically detecting developer activities and problems in software development work. In: Proceedings of the ACM/IEEE International Conference on Software Engineering (2012). DOI 10.1109/ICSE.2012.6227104
35. Sawadsky, N., Murphy, G.C., Jiresal, R.: Reverb: Recommending code-related web pages. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 812–821 (2013). DOI 10.1109/ICSE.2013.6606627
36. Shen, J., Irvine, J., Bao, X., Goodman, M., Kolibaba, S., Tran, A., Carl, F., Kirschner, B., Stumpf, S., Dietterich, T.G.: Detecting and correcting user activity switches: Algorithms and interfaces. In: Proceedings of the International Conference on Intelligent User Interfaces, pp. 117–126 (2009). DOI 10.1145/1502650.1502670
37. Singer, J., Elves, R., Storey, M.A.D.: NavTracks: Supporting navigation in software maintenance. In: Proceedings of the IEEE International Conference on Software Maintenance, pp. 325–334 (2005). DOI 10.1109/ICSM.2005.66
38. Wexelblat, A., Maes, P.: Footprints: History-rich tools for information foraging. In: Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems, pp. 270–277 (1999). DOI 10.1145/302979.303060
39. Ying, A.T.T., Robillard, M.P.: The influence of the task on programmer behaviour. In: Proceedings of the IEEE International Conference on Program Comprehenension, pp. 31–40 (2011). DOI 10.1109/ICPC.2011.35
40. Zou, L., Godfrey, M.W.: An industrial case study of Coman's automated task detection algorithm: What worked, what didn't, and why. In: Proceedings of the IEEE International Conference on Software Maintenance, pp. 6–14 (2012). DOI 10.1109/ICSM.2012.6405247

NOTE: This is the authors' version of the work.
Springer holds the copyright for the original publication, which can be accesses via it web site
http://www.springer.com/computer/swe/book/978-3-642-45134-8