Supporting Program Comprehension by Automatic Bookmarks

Automatische Lesezeichen zur Unterstützung des Programmverstehens

— Master Thesis —

Moritz Weinig

University of Bremen
Computer Science

December 2020

Supervised by

Martin Schröer

Examined by

1. Prof. Dr. Rainer Koschke
   University of Bremen
2. Prof. Dr. Oliver Keszöcze
   FAU Erlangen-Nürnberg

Abstract

In order to fix or extend a program, software developers spend considerable amounts of their working time on just reading existing source code, attempting to find out what changes have to be made, at which locations. Researchers and the software engineering community presented many tools addressing this issue. A function most modern Integrated Development Environments (IDEs) offer is *bookmarks*. Such bookmarks can be toggled for individual lines in the source code. Often, they do not carry any contextual information apart from their location. Prior research showed that most software developers do not use bookmarks at all. This thesis proposes an automatic bookmark approach which has been implemented in a prototypic plug-in for the popular Eclipse IDE. It continuously monitors the developer's interaction with the IDE and dynamically marks possibly interesting locations in the source code. The plug-in uses a line-based degree-of-interest (DOI) model to decide which locations might be relevant to the developer. It has been evaluated in a remote study with eight participants. The study suggests that automatic bookmarks can support developers' program understanding, which encourages further research efforts on this topic.

Contents

List of Figures

List of Tables

1 Introduction

This document contains a master thesis in Computer Science with a focus on safety and software quality. The project has been supervised by the *Software Engineering Group* at the University of Bremen, which researches, among other topics, program comprehension.

This introductive chapter motivates the performed research project, outlines the thesis's structure, and defines the underlying research questions.

1.1 Motivation

Due to a publication by Fjeldstad and Hamlen in 1979, the software engineering community assumes maintenance developers spend approximately half of their effective working time understanding existing source code. A more recent study by Ko et al. (2006) found that developers spend 35% of their time on navigation within and between source files. Robert C. Martin, the initiator of the *Agile Manifesto* and author of *Clean Code*, claims it is even worse (Martin 2009):

> "Indeed, the ratio of time spent reading vs. writing is well over 10:1. We are *constantly* reading old code as part of the effort to write new code."

Assumptions like these have been, and still are the reason for various efforts in improving program comprehension.

In addition to proactive efforts performed by the code's original authors, such as documentation and compliance with existing coding conventions, numerous tools developed to help maintenance programmers exist. To avoid discontinuity, it makes sense to embed such tools into the environment where the developer also wants to edit the program's source code. Nowadays, for many software developers this is an *Integrated Development Environment* (IDE), including a powerful editor, build-automation and debugging tools, as well as tools for version control, and other recurrent tasks.

A function that most modern IDEs offer is *bookmarks*. These bookmarks are tuples of a source file, a line number in that file, and optionally other information such as a title. Bookmarks can be added, edited, and removed by the user. Usually, bookmarks are displayed by a symbol in the editor's margin. Some IDEs offer to

Figure 1.1: Bookmark in Eclipse

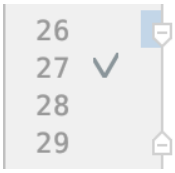


Figure 1.2: Bookmark in IntelliJ IDEA

view bookmark details such as a title in a table that summarizes all bookmarks in the opened project.

Prior research showed that most developers do not use those bookmarks at all (Guzzi et al. 2011; Murphy et al. 2006; Storey, Ryall et al. 2008)—even if they commonly estimated the use of bookmarks to be useful in an interview prior to the trial (Guzzi et al. 2011). Some developers consider bookmarks being not useful or cumbersome to create (ibid.).

This thesis aims to improve the elementary bookmark concept by automating the creation of bookmarks in order to support program comprehension. To pursue this goal, a tool has been implemented, which automatically adds bookmarks to code lines that might be interesting for the developer. Ideally, these *automatic bookmarks* should always offer the required information at the right time. The prototype developed for this thesis implements the following core features:

- Automatically bookmarking locations of potential importance or interest.
- Traceability of the developer's journey through the code.
- A configurable filter allowing the user to hide bookmarks of specific categories.

1.2 Thesis Structure

The definition of the research questions is followed by a chapter (Chapter 2) that sums up the project's background, including a literature review. Chapter 3 describes the design of the developed automatic bookmark tool, which is evaluated in Chapter 4. Finally, Chapter 5 discusses the findings and suggests future research topics.

1.3 Research Questions

The following specific research questions structure the project. The first two questions address the conceptual aspects of the above-mentioned bookmark approach. Two further questions aim to assess the tool's value for software developers who are in charge of maintenance tasks.

Implementation

The answer to the first question will sort out what user interaction events could be used to create automatic bookmarks in general. This will be derived from prior studies and literature to IDE bookmarks and program comprehension.

Research Question RQ1
Which interaction events are eligible for the creation of automatic bookmarks, and how is this influenced by the features of the corresponding code lines?

An implementation of the bookmark functionality resulting from RQ1 might lead to an entire pool of possible bookmarks. Prior research found that developers take different steps to understand a program, each step with different information needs (Maalej et al. 2014; Roehm et al. 2012; Sillito et al. 2008). The answer to the second research question will include in what context individual bookmarks have to be actually displayed to the developer in order to support the code analysis.

Research Question RQ2
How can be decided which particular automatic bookmarks will be created and how should they be displayed to the developer with regard to a specific comprehension task?

Evaluation

A prior survey by Guzzi et al. (2011) found most developers do not use bookmarks at all. Half of the surveyed did not know bookmarks even exist. For this reason, one of the key challenges will be to motivate developers to integrate the new tool into their settled workflows. Question RQ3 will investigate the developers' acceptance of automatic bookmarks.

Research Question RQ3

Do software developers show a greater acceptance towards the use of automatic bookmarks compared to classic bookmark approaches?

The last research question will validate whether the developed tool really supports the daily work of software developers.

Research Question RQ4

Can automatic bookmarks support program comprehension, and if so, to what extent?

Background 2

To propose an improved version of the rather neglected, classic bookmarks, it is crucial to understand today's bookmark approaches and related previous research. For this reason, this chapter gives an introduction to the state of the practice concerning bookmarks in popular IDEs as well as research on developers' bookmark usage and code comprehension in general. Moreover, the *Mimesis* framework by the University of Bremen's Software Engineering Group is introduced. It was developed as an Eclipse plug-in for recording developer interaction in studies on code comprehension and provided a basis for the developed automatic bookmark plug-in.

2.1 Bookmarks

Primitive bookmarks are offered by almost every modern IDE. For this thesis, a prototypical tool offering experimental automatic bookmarks will be implemented as a plug-in for the popular Eclipse IDE. For this reason, this section analyzes the built-in bookmarks of Eclipse.

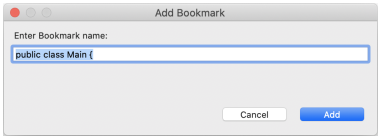


Figure 2.1: Dialog "Add Bookmark" in the Eclipse IDE

In Eclipse, the item Add Bookmark ... in the context menu of the editor's left margin can be used to add a bookmark to the line at the mouse cursor's position. If the user wants to add a bookmark, an input dialog opens, asking the user to insert a *bookmark name*—by default the code line's content is used as the name (Figure 2.1).

A bookmark is displayed by a stylized ribbon bookmark in the editor's margin, left of the line numbers (see Figure 1.1 on page 2), where they can be removed by Remove Bookmark ... in the context menu.

A hideable *bookmarks view* contains a sortable table displaying a *description* (name), the *resource* (file), *class path*, as well as the *location* in the file (line) for every bookmark. It is possible to change the bookmark's description in the table. Go to in the context menu of the bookmark view's table allows navigating to the bookmark in the editor, where the relevant file will be opened.

Bookmarks are associated with a line number. Eclipse recognizes line changes inside the IDE but not if lines were added or removed outside.

Some researchers and third-party developers searching for improvements to Eclipse's bookmark concept have developed plug-ins allowing to use or assess their approaches in Eclipse. All these approaches have in common that they solely extend the classic, manual bookmarking and do not include automatic bookmarks as proposed by this thesis.

Quick Bookmarks Marian Schedenig developed a third-party plug-in for Eclipse called *Quick Bookmarks* (Schedenig n.d.). It extends the Eclipse bookmarks by adding shortcuts to easily toggle bookmarks and to navigate between bookmarked lines.

Mesfavoris Another plug-in has been developed by Cédirc Chabanois (2018). *Mesfavoris* enables users to organize bookmarks in folders and share them via *Google Drive* among different computers. Moreover, it uses the *bitap algorithm* for a "fuzzy string search", helping the tool to match bookmarks with changed source files.

Pollicino Researchers from the Delft University of Technology and the University of Lugano have developed a plug-in offering "collective" bookmarks. As this plug-in has been developed and assessed specifically to support program comprehension, it will be described in detail in Section 2.2.

2.2 Bookmark Usage

Previous studies have focused on developer's usage of bookmarks and the improvement of the bookmark concept.

Bookmarks View in Eclipse

During a study on how Java developers are using Eclipse, Murphy et al. (2006) found only five out of 41 developers using the bookmarks view.

Task Annotations (Storey, Ryall et al. 2008)

In an empirical study exploring the role of task annotations in the management of (team) tasks, Storey, Ryall et al. (2008) found that 84% of the surveyed programmers never or rarely use bookmarks. One of the surveyed prefers `// TODO` annotations rather than bookmarks because of the better synchronization of textual comments.

Collective Code Bookmarks (Guzzi et al. 2011)

Guzzi et al. (2011) asked developers in an online survey why they do not make use of bookmarks. 88% of the respondents reported that they never or rarely use bookmarks. 50% of those developers did not even know bookmarks existed, 25% did not find them useful and 9% stated that creating bookmarks is cumbersome.

The authors also collected qualitative information. One developer who actually uses bookmarks explained, that he needs bookmarks when "digging into code." He also stated: "I must have a way to track all the jumps that I'm doing."

Based on the results of their survey, Guzzi et al. developed a tool called Pollicino which enables developers to share their insights with other team members. The Eclipse plug-in enables the user to "micro-document" findings as bookmarks that can be made accessible to others. The bookmarks are saved in an XML file.

According to Guzzi et al. bookmarks are a form of code documentation *outside* the code which leads to cleaner code. This approach is contrary to annotations like `// TODO` and the TagSEA system presented by Storey, Cheng et al. (2007) which enables the user to add tags to program elements, helping to navigate through the project. It is also contrary to *literate programming*, mainly propagated by Donald Knuth, where source code and documentation in natural language fuse together to one "work of literature" (Knuth 1984).

The Pollicino bookmarks are characterized by the following features:

- A marker in the editor's left margin, similar to the bookmarks deployed with Eclipse.
- A bookmark view displaying for all bookmarks in the workspace
  - a comment that may contain tags,
  - the respective resource,
  - the position in the resource, as line number,
  - the line's content,
  - the author of the bookmark,
  - as well as the date and the time of the bookmark's creation.
- Moreover, the bookmarks can be grouped and sorted.

To assess their tool, the authors performed a "pre-experimental" study consisting of a pretest questionnaire, an assignment (coding in Eclipse with the tool installed), and a posttest questionnaire followed by debriefing talks. In the pretest part, they asked the

participants questions about their habits in relation to code comprehension tasks. They learned that developers, in order to mark code locations, insert visually outstanding comments, or introduce lots of consecutive new lines and even compilation errors—even though the IDE offers bookmarks which were designed for this purpose.

Guzzi et al. finally conclude that Pollicino "can be effectively used to (micro-)document a developer's findings and that those can be used by others in her team." But they also admit that the new tool has not been effectively used during their study.

2.3 Program Comprehension

Below, a selection of research papers on program comprehension is presented to reflect the current state of research.

Seeking, Relating, and Collecting Information (Ko et al. 2006)

Ko et al. (2006) asked 31 developers to perform maintenance tasks in a Java system. Each experiment lasted 70 minutes. Every few minutes, the participants were interrupted by a program asking to solve a multiplication problem. By this means, the researchers wanted to simulate the interruptions commonly occurring in software engineering workplaces. The experiments were evaluated by analyzing screen-captured videos taken from the sessions.

Based on their findings, Ko et al. developed a model, which shall describe developers' program understanding. The model is summarized in Figure 2.2. It "describes program understanding as a process of *searching*, *relating*, and *collecting* relevant information, all by forming perceptions of relevance from cues in the programming environment" (ibid.). The researchers explain that developers start by picking any node, which can be any fragment of information related to the task. Possibly relevant nodes are related to depending nodes. If a node turns out to be irrelevant to the task or if more information is required, the developer visits more related nodes or even searches for entirely different nodes. All information necessary to the task is collected, for example, by memorizing or making a note.

Questions During a Program Change Task (Sillito et al. 2008)

In 2008 Sillito et al. published an article reporting on two qualitative studies they performed to find out what information programmers require and how developers find that information. Today, their

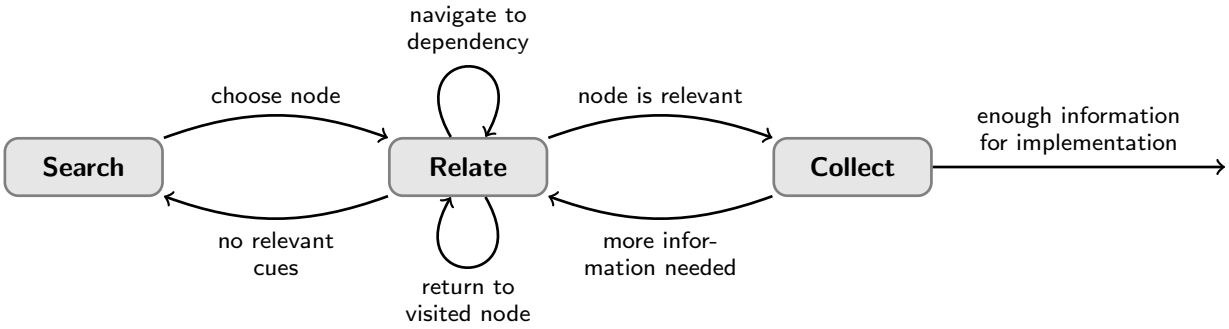


Figure 2.2: Model of program understanding proposed by Ko et al. (drawing inspired by Ko et al. (2006))

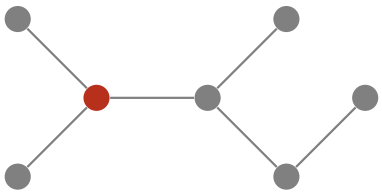
model belongs to the most widely accepted models on software developer's program comprehension.

During a laboratory-based investigation, the researchers asked nine participants to make code changes in an open-source project which was new to them. The participants had to work together in pairs because the discussion between the so-called "driver," editing the source code and the more experienced "observer," in each session, was used to learn about the participants' approaches. Each session ended with an informal interview.

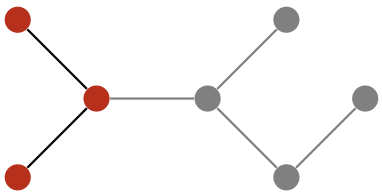
In an industry-based investigation, 16 developers working for the same company were observed while working on a real-life, non-trivial task chosen by themselves. During the session, the participants were supposed to think aloud. Like in the laboratory-based investigation, the participants were interviewed informally afterward.

To identify questions asked by developers, based on audio records from the sessions, first, a list of specific questions was made. The researchers then abstracted from the specific versions and determined 44 types of questions the participants asked during their observations. These questions were grouped into four categories which classify the questions by considering the code base as a graph:

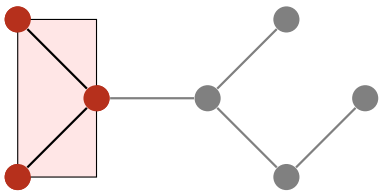
1. **Finding focus points:** Questions in this category ask for single elements in the graph (e.g. one method).
2. **Expanding focus points:** The second category contains questions asking for other entities directly linked to a given "focus point" (e.g. where a specific type is used).
3. **Understanding a subgraph:** This category contains questions asking for the behavior of entire subgraphs, i.e. how related entities in the graph work together.
4. **Questions over groups of subgraphs:** Finally, questions of the fourth category ask for the relations of subgraphs.



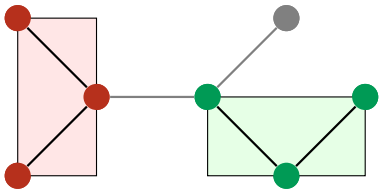
(a) Finding focus point



(b) Expanding focus point



(c) Understanding a subgraph



(d) Groups of subgraphs

Figure 2.3: The four categories defined by Sillito et al. in graphs consisting of linked software entities (inspired by Sillito et al. (2008))

In their paper, Sillito et al. do not claim these categories constitute a step-by-step instruction all developers follow in a strict gradual manner. Instead, the categories are based on the questions' scopes. Nevertheless, there might be scenarios where a developer starts by searching for any focus point, which is then expanded more and more in order to end up with a broad mental image of the code in question—especially if the programmer knew little about the code in advance.

The researchers also investigated the tool support for answering each of the 44 questions. They found that questions from the first two categories are fully covered by various tools, while most of the questions dealing with subgraphs are only partially covered by existing tools.

Foraging Diets (Piorkowski et al. 2013)

By analyzing nine screencasts collected in a previous study, Piorkowski et al. (2013) investigated *what* programmers are searching for during a debug task ("diets") and *how* they try to get this information ("foraging"). Their work is based on the *Information Foraging Theory* (IFT) (Pirolli and S. Card 1999) which uses a biologically inspired wording (Piorkowski et al. 2013):

> "In IFT, a *predator* (person seeking information) pursues *prey* (valuable sources of information) through a *topology* (collection of navigable paths through an information environment)".

Piorkowski et al. found that developers search for highly diverse information, even if they share the same goal (like fixing the same bug). Moreover, their participants spent only 24% of their time navigating between different locations and sources of information ("between-patch foraging"). Lastly, they found that developers do not necessarily consider the costs and benefits of their foraging (ibid.):

> "The participants' sometimes stubborn pursuit of particular information goals—tolerating very high costs even when their efforts showed only meager promise of delivering the needed dietary goal."

Comprehension of Professional Developers (Roehm et al. 2012)

Roehm et al. (2012) conducted a qualitative study with 28 participants who worked for software development companies. Students and researchers were explicitly excluded from the study to ensure the results reflect industry practice. Roehm et al. wanted to know

which comprehension strategies developers follow, which sources of information they use, and which information they miss. They also wanted to investigate developers' tool usage with regard to comprehension tasks.

As the researchers wanted to gain realistic results, they allowed the participants to choose a task from their real work instead of giving a predefined and constructed assignment to them. The developers were observed for 45 minutes while they were working on their tasks and commented on what they were doing ("think-aloud method"). The observations were followed by semi-structured interviews.

In many sessions, the authors observed recurring strategies, though these strategies varied among the developers. Roehm et al. also found that none of the observed developers used tools dedicated to program comprehension (such as visualization or metric tools). In their paper, the researchers speculate about possible reasons for this "gap" between academic research and industry practice (ibid.):

> "a) research results and their benefits being too abstract for industry,
> b) lack of knowledge about available tools among practitioners,
> c) fear of familiarization effort and lack of trust in new tools, or
> d) that using new tools requires too much training for practitioners"

Like Guzzi et al. in 2011, Roehm et al. also found that some developers do not know standard functions in their IDEs (like the Reference feature in Eclipse). The researchers suggest educating software developers to use their tools efficiently and to use research results for developing tools that provide filtered information based on the user's current activity. Subsection 2.4 will introduce an example for such a task-oriented Eclipse extension.

2.4 Mylar/Mylyn (Kersten and Murphy 2005)

Kersten and Murphy (2005) have presented an approach related to the automatic bookmark concept. They integrated their model into a task-focused Eclipse plug-in called *Mylar*, which is now known as *Mylyn* (Eclipse Foundation 2020).

The plug-in contains a degree-of-interest (DOI) model inspired by the work of S. K. Card and Nation (2002). Kersten and Murphy describe their DOI model as follows:

> "The Mylar model associates an interest value with each Java or AspectJ program element. When a program element is selected or edited, its DOI value increases. Over time, if the element is not selected or edited, its interest value decays."

Each selection of an element not just causes an increase of the associated DOI value but also a decrease of all other elements, which is meant by "interest value decay." Elements with a negative DOI are considered uninteresting, elements with a value ≤ -10 are removed from the model to save memory.

The researchers developed multiple views which give Eclipse users access to the DOI model. The views are shown in Figure 2.4. Most of the views are advanced versions of existing Eclipse views:

1. A **Package Explorer** with interest-based filtering, showing only the files that are relevant to the current task.
2. As the standard problem list usually shows a large number of compiler warnings, Mylar's **Problems List** uses the DOI model to highlight the currently relevant problems.
3. In contrast to its counterpart in the standard Eclipse interface, the **Outline** window in Mylar only shows the members related to the task.
4. Moreover, Mylar contains an AspectJ-related view using the DOI model, the **Pointcut Navigator**.

The authors evaluated their tool by asking developers to use Mylar for their daily work and conclude that the plug-in can help programmers reducing the time searching for locations. Chapter 3 will define a new DOI model inspired by the approach proposed by Kersten and Murphy. Instead of rating program elements, the automatic bookmark model rates lines in the source files, leading to a similar but alternative concept as the markings are not restricted to the structure of the source code.

2.5 Mimesis

The Software Engineering Group at the University of Bremen developed an Eclipse plug-in called *Mimesis* as part of a project performing fundamental research on program comprehension, funded by the German research foundation *DFG*. The tool allows recording user interaction events occurring in Eclipse. When the recording is stopped, the events are saved in an XML file. Table 2.1 lists the events recorded by Mimesis. The categories grouping the table are loosely based on Proksch et al. (2018). Mimesis' core logic has been reused in the tool developed for this thesis, which will be described in the next chapter.

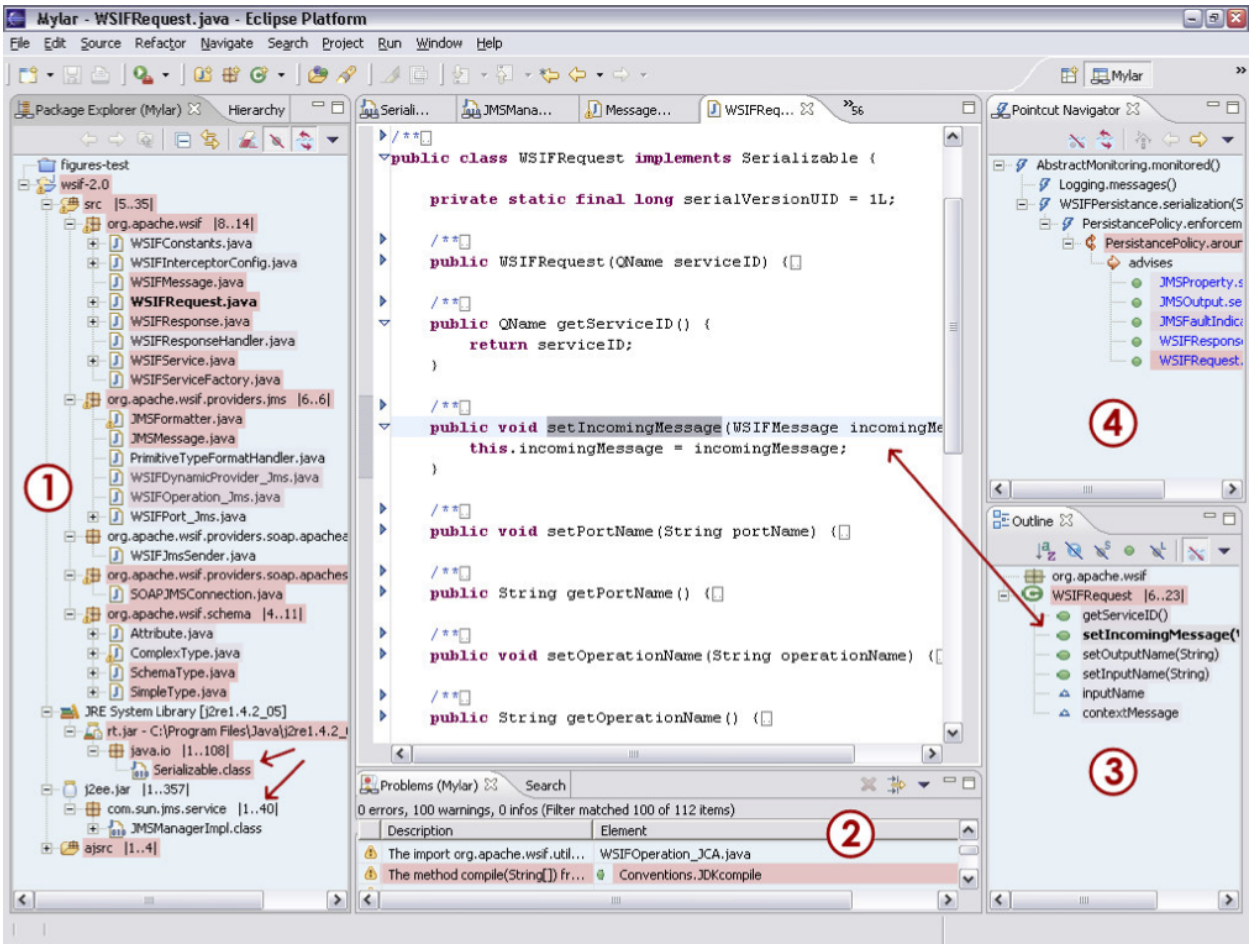


Figure 2.4: The Mylar interface (Kersten and Murphy 2005)

Figure 2.5: Visualization of interaction data collected by Mimesis

Table 2.1: Events recorded by Mimesis. The grayed-out event types do not denote events generated by Eclipse itself but by the plug-in which offers the recording of text and speech comments for evaluation purposes. These event types are definitely not eligible for an automatic bookmarks tool

Name	Trigger
`Event`	General event (base class)
	Activity
`Recording`	A recording has been started or stopped
`Scroll`	The user scrolled inside the editor
`TextSelection`	Text has been selected
	Build
`Launch`	Project is run or debugged
`Debug`	Debugging related event, i.e. a breakpoint set/hit, a manual break or a manual step inside of a debugged program
	Edit
`CodeChange`	Code in the IDE has been changed
`TextComment`	Developer added a comment
`CodeCompletion`	Code completion was used
`VoiceComment`	An audio comment has been added
	Environment
`Editor`	Event specific to an editor window
`View`	View is opened/closed, or focus has changed within the IDE
`Perspective`	Perspective is changed or opened
`Window`	Focus of the IDE has changed
	Navigation
`TreeViewer`	Elements in the navigation view have been collapsed or expanded
`TreeSelection`	Element in a tree view has been selected
`EditorMouse`	Mouse actions, like clicking or moving
`Search`	Search query has been conducted
`EditorTextCursor`	Text cursor has been moved to another position
	Solution
`File`	Event related to a single file
`Project`	Project has been loaded or removed
`Resource`	A file has been created, deleted or changed
`Save`	A file has been saved

3 Automatic Bookmarks

This chapter focuses on the development of the automatic bookmark model and its implementation.

3.1 Requirements

A prototypical tool offering experimental automatic bookmark functionality had to be implemented as a plug-in for the popular Eclipse IDE. Eclipse was chosen because it allowed reusing parts of the Mimesis system (see Section 2.5), developed by the Software Engineering Group. Technically it would have been possible to develop the plug-in also for other popular IDEs written in Java, such as IntelliJ and NetBeans, but probably that would have gone beyond the scope of this master thesis, regarding the required expenditure of time.

A reference use case helped in answering the first research question RQ1. It describes a generic comprehension task in an Eclipse environment the plug-in is intended to assist in. The use case diagram in Figure 3.1 integrates all aspects of the outlined approach.

Reference Use Case

In an existing software system, a developer wants to change or extend functionality, which may be part of fixing a bug. In the beginning, the developer does not know the location or other characteristics of the intended change. Maybe the developer is even working on the software system for the first time.

As assumed by Sillito et al. (2008) the developer starts foraging by more or less randomly picking a focus point. By expanding this focus point the search for key code locations is pursued. The developer browses the files and discontinuously reads code sections. Depending on system complexity and target environment, the program may also be executed. Maybe the developer also edits the code between executions by changing values, deactivating parts of the systems by commenting, or adding print statements for debugging purposes. These activities will be carried out until the entire task is completed.

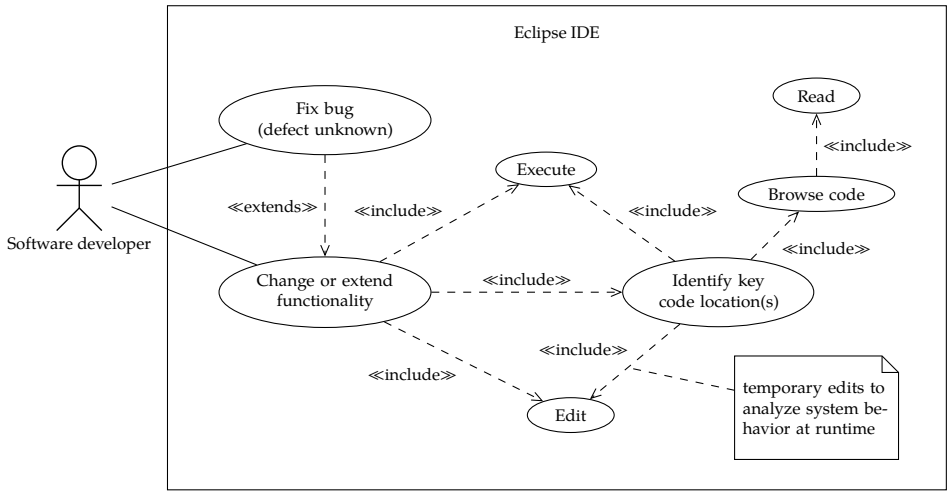


Figure 3.1: UML use case diagram depicting the reference use case the developed plug-in shall assist

One can assume that developers do not approach the key location in a strict target-oriented manner. Possibly there will be multiple visits to the same location during the comprehension process. At least the following two cases are imaginable:

a) While browsing, the essential code area is visited, though the developer does not recognize it yet. The developer will move on but finally return.
b) The developer finds the essential code area but without knowing exactly what changes have to be applied. The comprehension task will continue, but now in reference to the key location already identified. For example by descending into the call hierarchy of methods. The developer will return to the key location over and over again until the task is completed.

Meaningful Interaction

Piorkowski et al. (2013) found that developers stubbornly pursue very specific information goals. The reference use case assumes a developer wants to return to locations that have been visited before. For example, because a location combines multiple hints to very different program parts that have to be examined one after another in order to find the relevant information.

Such key locations should be emphasized by an automatic bookmark tool, which is why it has to detect recurrent visits to the same locations. Moreover, it has to find locations that stand out because of accumulative code changes and execution events occurring in the location context. A lot of interaction in the same area will probably characterize a location with a special meaning to the developer.

Event	Reason
~~`Event`~~	Too general
~~`Recording`~~	Mimesis event
~~`TextComment`~~	Mimesis event
~~`VoiceComment`~~	Mimesis event
~~`Project`~~	Out of scope
~~`Save`~~	Any changes have been recorded by other events already

Table 3.1: Event types recorded by the framework that can be ignored from the outset as in all likelihood they do not carry any information relevant to the automatic bookmarks designed in this thesis

Which of the events recorded by the Mimesis framework can be used to gather the information outlined above?

The events listed in Table 3.1 do not have to be taken into account. They have technical reasons or do not occur within the plug-in's scope. The table gives a reason for every ignorable event type.

From the remaining event pool, the following events have been selected for being processed by the prototypic automatic bookmark plug-in. They cover most aspects of the three core activities given by the reference use case: browse and read, edit, and partially execute (by analayzing debug events).

Viewport The code lines currently displayed to the Eclipse user can be detected by `Scroll` events as well as by view or editor specific event types. As Mimesis also offers access to the current `EventContext` at any time, a periodical time-discrete recording of the current viewport is possible as well. Such a recording would not just register occurring events but—without further processing—also detect periods without any interaction, which may indicate a phase of intense reading.

In addition to a line-based recording rather than an evaluation of program elements like entire methods, this approach would be another advantage over the one suggested by Kersten and Murphy (2005). As already mentioned in Subsection 2.4, Mylar increases its degree-of-interest values whenever the developer interacts with any program element. In Mylar, an interaction event is also necessary to trigger the "periodic" decay, which downgrades the DOI value of all other program elements. If the developer just looks at the currently displayed code lines for a while, DOI values are neither increased nor decreased.

Text selection `TextSelection` events can refine the viewport information providing the plug-in's model with information on interesting lines or even only parts of them.

Code changes Changes in files will be recorded by `CodeChange` events. Typing will lead to a single change event for every character. Whereas pasting copied text is represented by only one event instance.

Debug `Debug` events such as the creation of breakpoints.

Plug-in Interaction with the plug-in itself should be recorded as well to detect breaks in the actual comprehension task.

Conclusion The following list contains the events that will be processed by the plug-in. It answers RQ1 which asks for the events eligible for the creation of automatic bookmarks.

- Permanent monitoring of the viewport,
- combined with the recording of `TextSelection` events.
- Evaluation of `CodeChange` events to detect changes, including automated changes like code completion or code generation.
- `Debug` events.
- Recording of the user's interaction time with the plug-in.

3.2 Detecting Regions of Possible Interest

Approaching RQ2, an automatic-bookmark model had to be developed. This model had to define rules for detecting those regions in the browsed source code that might be relevant to the developer and therefore shall be bookmarked.

Data

The analysis leading to the automatic bookmark model was based on a set of Mimesis recordings collected by the Software Engineering Group prior to the work related to this thesis. Not all available datasets could be used as the recording tool still was subject to heavy changes during those early recordings, which resulted in some erroneous or incomplete recordings. The author of this thesis also participated in the outlined study. As this happened before the work at this thesis officially started, without much research on automatic bookmarks or code comprehension done, this dataset became part of the bookmark analysis.

For the recordings, academic software developers, students and university staff members, were asked to edit the source code of an example originally published by Oracle. The code implements a

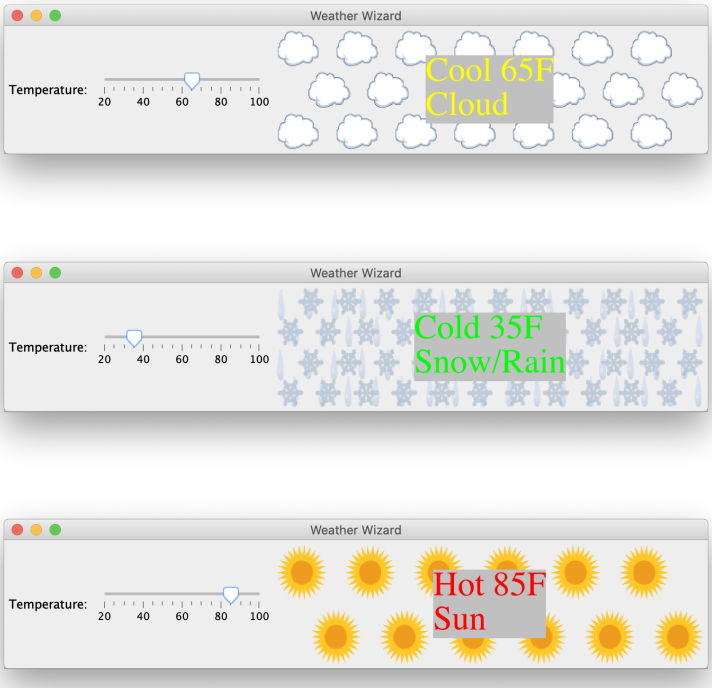


Figure 3.2: Different states of the Weather Wizard's UI

tiny Java Swing application called *Weather Wizard*. The code defines a window with a slider allowing the user to select a temperature. Changing the slider updates a text label and some images. Figure 3.2 shows some states of the Weather Wizard's frame. The entire source code is part of the same class file.

The participants' task was to change the slider's unit from degree Fahrenheit to degree Celcius. Table 3.2 contains for each participant the time spent on this task.

Analysis

Assuming that the user is interested in those code sections, where many events occurred or that the developer viewed often or for long periods, a new 'degree of interest' (DOI) has been defined. This DOI is inspired by the DOI by Kersten and Murphy (2005), but uses a line-based approach. In contrast to the Mylar approach it is not limited to program elements.

Moreover, the DOI does not only respond to actual events resulting from the developer's interaction with the IDE, but also integrates the editor's viewport—even when it is not changing. Therefor, the Mimesis framework has been extended by a `ViewportEvent`, which the automatic bookmarks logic generates in steps of 250 millisecond for the currently visible lines in the editor.

Table 3.2: Time each participant spent on the Weather Wizard task. The gray row contains the participant with the median time

Participant	Time [h:m:s]
e	0:15:30
f	0:16:42
b	0:16:46
d	0:18:26
c	0:36:50
a	0:38:28
g	2:16:39

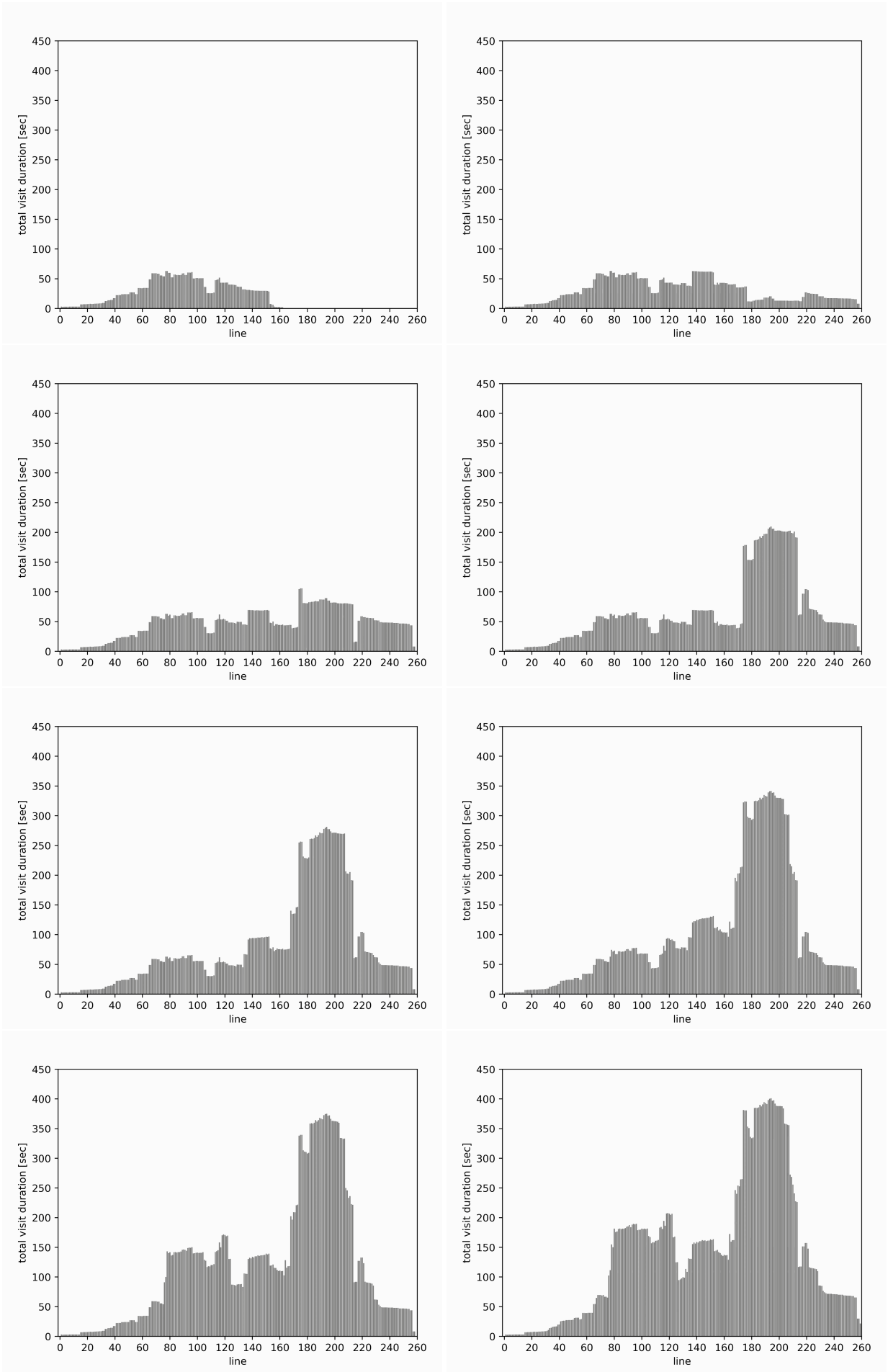


Figure 3.4: Exemplary development of line visit durations. The diagrams show the data from the "d" recording in two minute steps, starting two minutes after `WeatherWizard.java` has been opened for the first time

The software system developed for this thesis includes a simulator allowing to use XML recordings generated by Mimesis to simulate the behavior of the plug-in logic, or just parts of it. Instead of expecting events generated by a running Eclipse instance it feeds the events from the record file to the DOI model. This simulator helped assessing the final model prior to the experiments with developers (see Section 3.4). But it was already utilized in earlier steps of the development. Using the simulator, the DOI model was used to generate data for different visualizations of the Weather Wizard sessions that helped to define the plug-in's logic based on real developer behavior. The plots were created by Python scripts, based on the simulator's outputs.

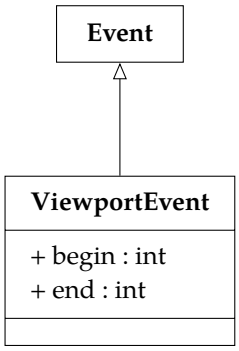


Figure 3.3: A `ViewportEvent` has been added to Mimesis allowing to record the currently displayed lines and process these information in the DOI model just like the events actually caused by the user

Time Lapse

First, the DOI's development over time generated from `ViewportEvents` and `TextSelectionEvents` has been explored to gain information on the developers' approaches. Figure 3.4 on page 20f. shows some frames of one recording.[1] Analyzing the visualizations, the following observations have been made:

Developers tend to first scroll down the entire document for getting an overview, without spending too much time at one certain position. Similar to the assumptions made by Sillito et al. (2008), this first inspection is followed by picking a 'focus point.' In the DOI plots, this is characterized by a rising 'peak' exceeding the gauge of the initial inspection. The developer might change the focus, which leads to other peaks in the diagram. Developers also alternately switch between major peaks, which indicates that the developer is working on different sections that are in some way linked to each other.

This is a pattern that, in fact, could be assisted by automatic bookmarks, providing easy access to the relevant sections.

Visits

To gather more information on how the peaks come about, another plot type was developed. It visualizes the visit durations for each line by giving the visits in each bar different colors. A visit is a series of DOI increases caused by a series of consecutive events. Between two visits lies a time gap during which the respective line was not visible in the editor. The gap must be larger than 250 milliseconds, the interval between the viewport captures.

[1] The recording of participant d has been chosen as an example for this document as it is the one with the median recording time.

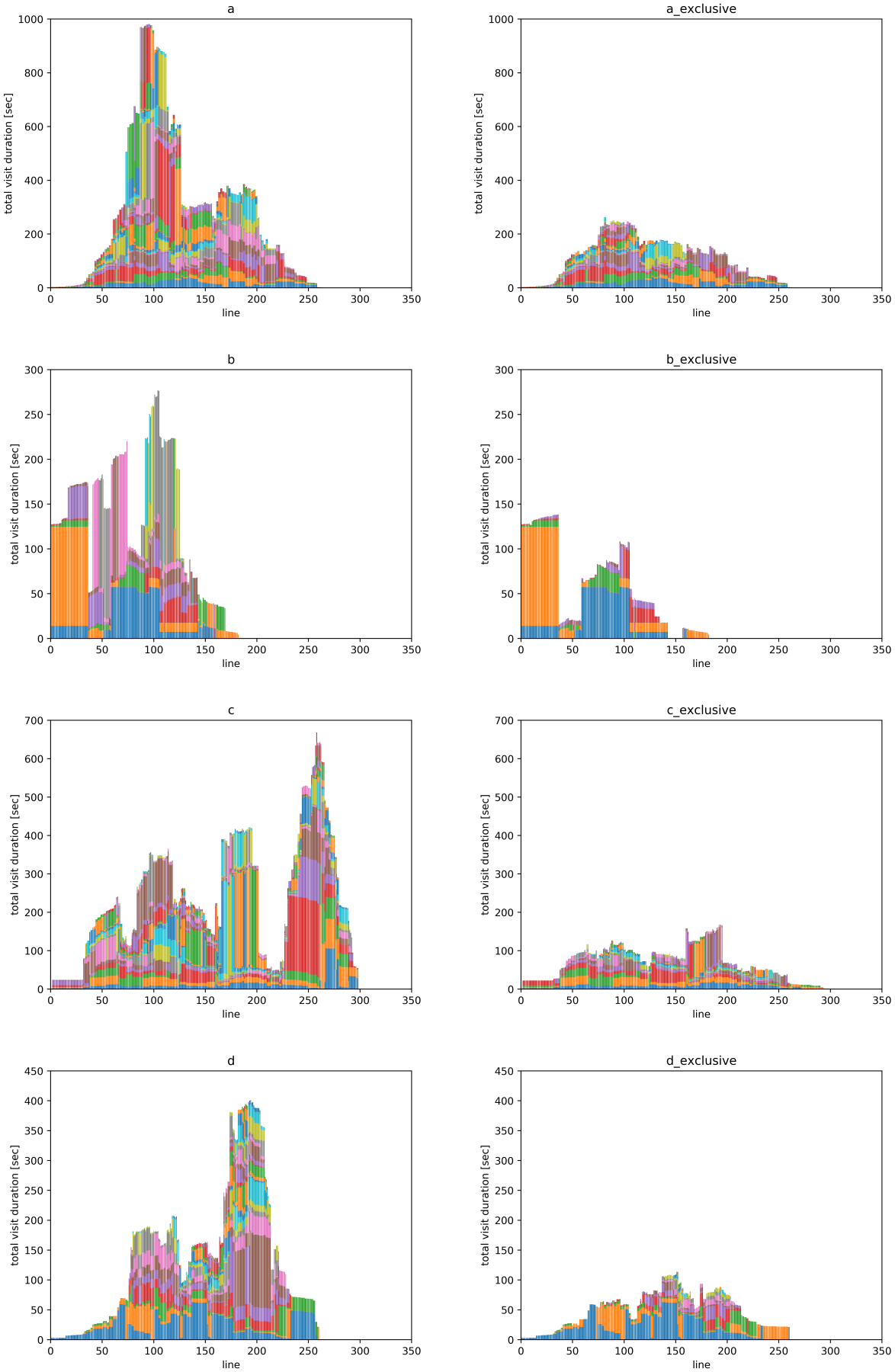
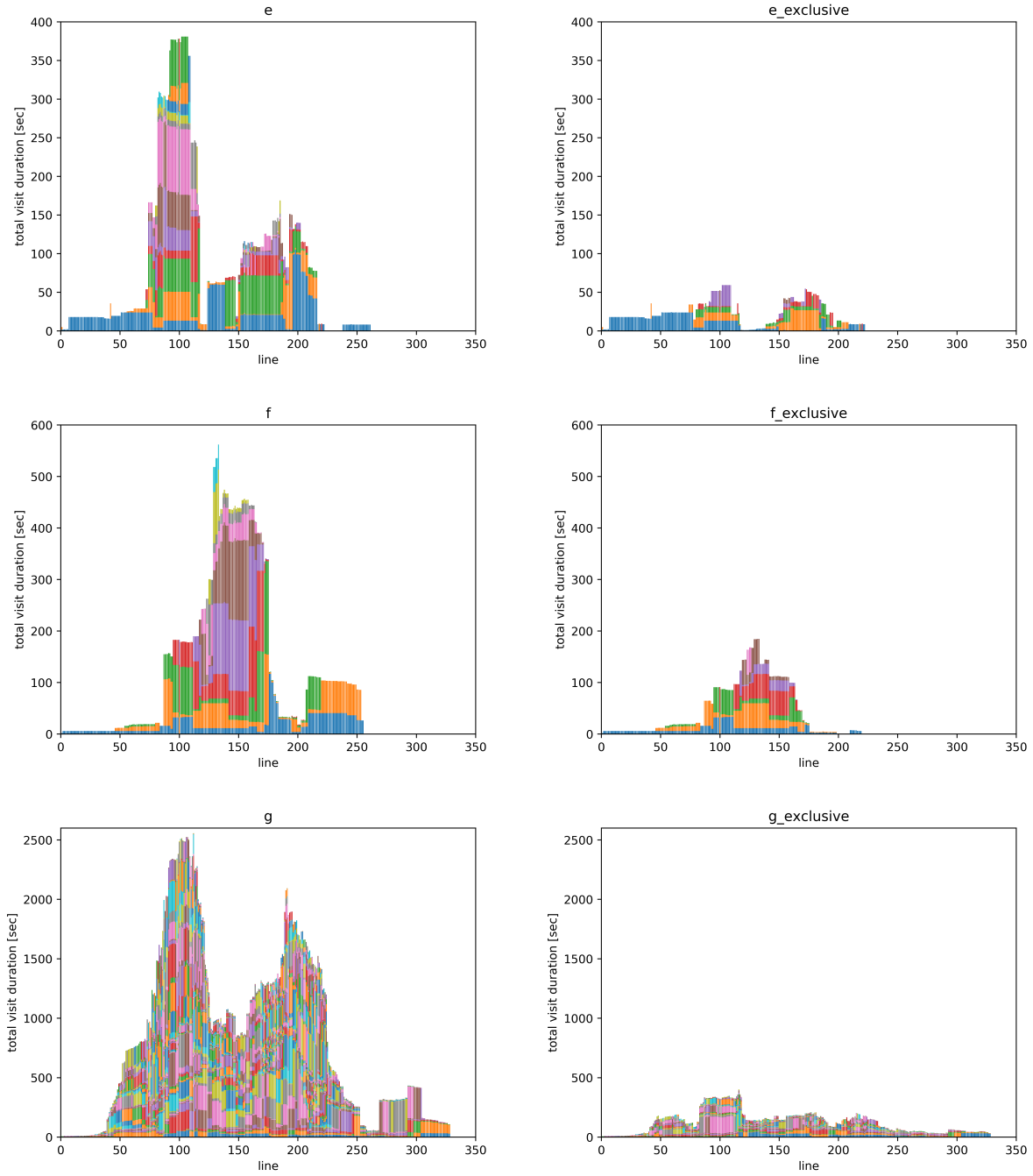
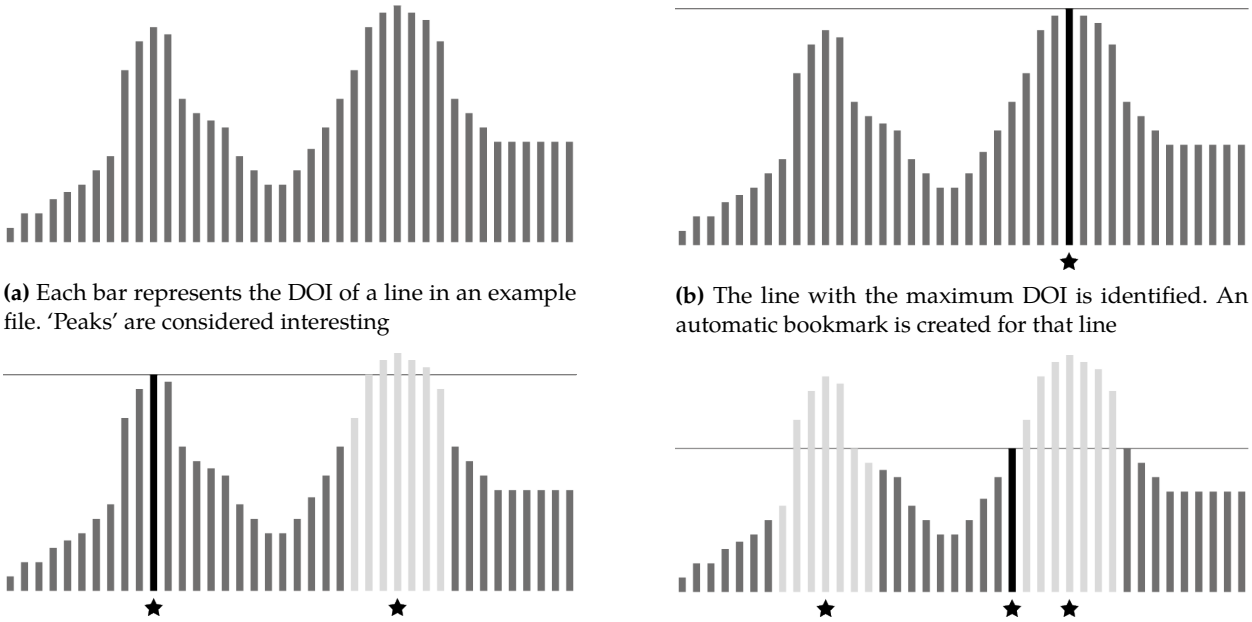


Figure 3.5: Visit durations per line in `WeatherWizard.java`. Different colors in one bar indicate independent visits of the respective line. In the diagrams in the right columns ("...-_exclusive"), visits during which the test person edited the line have been removed to reveal peaks that resulted from viewing only


e
e_exclusive
f
f_exclusive
g
g_exclusive
total visit duration [sec]
line



(a) Each bar represents the DOI of a line in an example file. 'Peaks' are considered interesting

(b) The line with the maximum DOI is identified. An automatic bookmark is created for that line

(c) During further iterations, areas around bookmarked maxima are not considered again

(d) If two lines have the same DOI, the one with the lower line number will be considered, first[2]

Figure 3.6: Bookmarking with the masking approach

An alternative version of this plot only contains the visits that did not include any edits. This allows finding peaks that did not just result from code changes (which, in most cases, require visiting the respective lines). Figure 3.5 on page 22f. contains the plots.

The plots show that developers do visit multiple locations multiple times—especially those they ended up editing. Moreover, as the largest peaks do not appear in the 'exclusive' plots in an identical form, they always contained visits used to edit.

Aggregating Regions

The analysis showed that bookmarking peaks in the DOI model might lead to a serviceable automatic bookmark tool. Hence, a method had to be found for aggregating regions with high DOI values. Two alternative methods were considered: masking and damping.

Masking

The first method searches for a file's maximum DOI and adds a bookmark for the respective line. Then, a region including r lines above the bookmark and r line below the bookmark's position is banned from the search of the following iterations.

[2] This is a design decision with no specific reasons. Further versions might analyze the contents of the code lines and choose a line based on its features.

The 'radius' r is given by the number of currently visible lines in the editor divided by two. This procedure is applied n times or until the entire file is masked by the algorithm, meaning that no more bookmarks, of the particular type, can be created. Algorithm 1 shows the pseudocode of this approach. Figure 3.6 gives a detailed example of this method.

Algorithm 1: Aggregation of DOI maxima with masking

```
foreach DOI-rated file do
    for i ← 1 to n by 1 do
        identify line with maximum DOI;
        create bookmark for the identified line;
        block region of radius r around bookmark position
          for maximum searches in further iterations;
```

Damping

The second approach works quite similar to the masking method. But instead of banning the bookmarked region from the maximum search of further iterations, the DOI values of the lines in the area will just be decreased by the region's median. Thus, already bookmarked regions might be considered for further bookmarks if they still exceed the remaining peaks. Actually, peaks with very large values will not be marked multiple times, but the fact that less than n bookmark were created will indicate towering peaks in the model. Algorithm 2 shows this method in pseudocode.

Algorithm 2: Aggregation of DOI maxima with damping

```
foreach DOI-rated file do
    for i ← 1 to n by 1 do
        identify line with maximum DOI;
        create bookmark for the identified line;
        decrease all values in region of radius r around
          bookmark position by the region's median;
```

3.3 Specification

Summing up the thoughts and assumptions from the previous sections, this list contains specifications for the automatic bookmark model implemented in the Eclipse plug-in:

- A DOI will be continuously computed for each line in the source code. The value is increased by 1 whenever a relevant event occurs.

- The DOI will be computed separatley in the following four categories:
  - Viewport, based on occurrences of `ViewportEvent` and `TextSelectionEvent`
  - Edit, based on occurrences of `CodeChangeEvent`
  - Debug, based on `DebugEvent` occurrences
  - Default, based on all events of the other categories together (`ViewportEvent`, `TextSelectionEvent`, `CodeChangeEvent` and `DebugEvent`)
- Automatic bookmarks will be created for the regions around the current DOI maxima in the source files. The regions will be selected by the two alternative methods described above, masking and damping. New maxima will cause the creation of new bookmarks and the destruction of obsolete ones to comply with the number of maximum bookmarks per file and type n.

Figure 3.7: The outputs of the bookmark simulations have been translated into source code listings with colored text marks for the bookmark regions

Table 3.3: Numbers of insertions (+) and deletions (-) in the Weather Wizard class file for each participant

	Part.	+	-	Sum
1	f	11	3	14
2	a	15	5	20
3	d	17	17	34
4	e	25	17	42
5	b	27	19	46
6	c	54	16	70
7	g	157	49	206

3.4 Checking the Automatic Bookmark Model

Using the simulator already mentioned in Subsection 3.2, the implemented model has been pre-evaluated prior to the final study described in Chapter 4. Again, the existing recordings a to g were used. The simulation was performed with the maximum number of bookmarks for each file and category set to one, five, and ten bookmarks ($n = \{1, 5, 10\}$). Also, for each participant and each n-parametrization, both the mask and the damp mode were simulated.

Only the bookmarks existing at the end of the simulated session were reviewed—with regard to the final code version. Of course, the simulator can retrace the bookmark history. However, even though the recordings contain all code change events, by the time the simulation was done, no tool existed in the Mimesis framework allowing to restore a source code version from the event history. Without the source code version of the time a bookmark was created, it is difficult to decide whether the bookmark makes sense or not.

First, the numbers of generated bookmarks in each simulation were examined and checked for plausibility. Table 3.4 shows the data for all simulations. For all recordings, the $n = 1$ configuration led to one bookmark—except in the Debug category as only the work of participant c resulted in a Debug bookmark. As he only toggled one holding point, in all configurations an equal bookmark was created. The damp mode always created fewer or as many bookmarks as the mask mode with the same record and n, but

Table 3.4: Numbers of generated bookmarks in the final states of the simulations for each participant and configuration

(a) Default

	$n = 1$		$n = 5$		$n = 10$	
	mask	damp	mask	damp	mask	damp
a	1	1	5	5	10	6
b	1	1	5	5	10	5
c	1	1	5	5	10	7
d	1	1	5	2	10	2
e	1	1	5	5	10	8
f	1	1	5	2	10	2
g	1	1	5	5	10	5

(b) Viewport

	$n = 1$		$n = 5$		$n = 10$	
	mask	damp	mask	damp	mask	damp
a	1	1	5	4	10	4
b	1	1	5	4	10	4
c	1	1	5	5	10	7
d	1	1	5	5	10	5
e	1	1	5	5	10	10
f	1	1	5	2	10	2
g	1	1	5	5	10	10

(c) Edit

	$n = 1$		$n = 5$		$n = 10$	
	mask	damp	mask	damp	mask	damp
a	1	1	5	1	6	1
b	1	1	5	1	10	1
c	1	1	5	1	10	1
d	1	1	5	1	6	1
e	1	1	5	1	10	1
f	1	1	5	1	3	1
g	1	1	5	2	10	2

(d) Debug

	$n = 1$		$n = 5$		$n = 10$	
	mask	damp	mask	damp	mask	damp
a	0	0	0	0	0	0
b	0	0	0	0	0	0
c	1	1	1	1	1	1
d	0	0	0	0	0	0
e	0	0	0	0	0	0
f	0	0	0	0	0	0
g	0	0	0	0	0	0

never more. As expected, checking random samples indicated that the $n = 1$ sets were subsets of the $n = 5$ sets, which were again subsets of the $n = 10$ sets (Table 3.5). Moreover, the damp sets were subsets of the mask sets. The implementation does not allow multiple bookmarks of the same type in the same line. This explains why the damp mode often created fewer bookmarks than the mask mode.

The Edit simulations for $n = 10$ and mask mode, for example, showed another irregularity. Three of the simulations created far less than ten bookmarks. To investigate such abnormal behavior, a Python script was written to transform the simulator's output into LaTeX-based source code listings, with each bookmark region colored differently (see Figure 3.7). This process resulted in 132 PDF files, each containing the bookmarks for a single configuration (recording/participant, n, type, mask/damp mode). In connection with the outputs of the Unix tools `diff` and `diffstat`, it turns out, that those participants a, d, and f only introduced a few changes actually, which is the reason why the algorithm was not able to generate 10 bookmarks. For example, participant f only changed three existing lines and just added two methods. Table 3.3 contains a sorted list of the change sums in ascending order. Not surprisingly, a, d and f cover the first three ranks.

Table 3.5: Generated bookmarks for two randomly chosen participants. Each entry denotes the line number associated with the respective bookmark

(a) Participant a

	Default		Viewport		Edit		Debug	
n	mask	damp	mask	damp	mask	damp	mask	damp
1	92	92	95	95	92	92		
	81	81	81	81	112			
	112	112	105	105	149			
	123	123	119	119	189			
5	189	189	187		202			
	171	171	69		123			
	69		129					
	135		149					
	149		171					
10	199		197					

(b) Participant c

	Default		Viewport		Edit		Debug	
n	mask	damp	mask	damp	mask	damp	mask	damp
1	261	261	258	258	263	263	187	187
	97	97	174	174	45			
	114	114	193	193	97			
	179	179	240	240	123			
5	242	242	275	275	179			
	200	200	96	96	159			
	278	278	114	114	140			
	65		65		197			
	131		131		245			
10	160		148		288			

In the same way, the created toolset could be used to check the bookmarks generated by the damping method. Eventually, the analysis led to the assumption that the implementation of the automatic bookmark model works according to its specification. Section 3.5 will present the user interface of the plug-in giving Eclipse users access to the automatic bookmark model. The development of the plug-in allowed to finally assess the model not just in a simulation but in experiments with real developers.

3.5 User Interface

Even a perfect automatic bookmark model would be useless without an interface enabling developers to make use of the collected data. This section gives a brief introduction to the graphical user interface of the developed plug-in. The plug-in not just adds automatic bookmarks as markers and highlights to the code editor but also offers a view with different tabs containing detailed information. Figure 3.11 provides an overview of the plug-in's components visible to the user.

Regarding the implementation of the user interface, this thesis does not go into specifics. For technical details, please consult the source code on the attached CD-ROM. Most elements contain *Javadoc* comments.

Bookmarks

Editor

In the Editor, automatic bookmarks are displayed by a marker next to the line numbers, just as built-in markers, like 'classic' bookmarks.[3] Besides, the entire region belonging to the bookmark is highlighted by a colored text marker, which is applied to the source code in the respective lines. This marker type also adds colored bars next to the scroll bar allowing to find all bookmarked regions at one glance, without scrolling through the document.

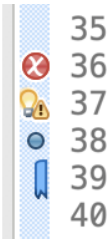


Figure 3.8: Different markers in Eclipse. From top to bottom: Error, warning, breakpoint and (built-in) bookmark

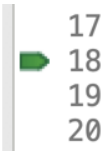


Figure 3.9: An automatic bookmark

View

A table in the plug-in's view lists all automatic bookmarks.[4] It can be sorted with respect to each of the columns. The table is made up of the following columns:

- **Rank:**[5] The rank based on the DOI for the respective file and the bookmark type ($1 \cdots n$).
- **DOI:** The DOI at the time, the bookmark was created.
- **Type:** The bookmark's type (Viewport, Edit, etc.).
- **Ressource:** The (source) file containing the bookmark.
- **Line:** The line the bookmark is associated with.

[3] Flatt and Maison (2011) wrote a short but helpful introduction to the implementation of custom Eclipse markers.

[4] The view can be opened by navigating to Window › Show View › Other… › Automatic Bookmarks and selecting the *Automatic Bookmarks* view.

[5] This column has been introduced to replace the DOI column in the experiments. Thus, the participants were able to sort the bookmarks by significance without having told what 'DOI' means. See for details.

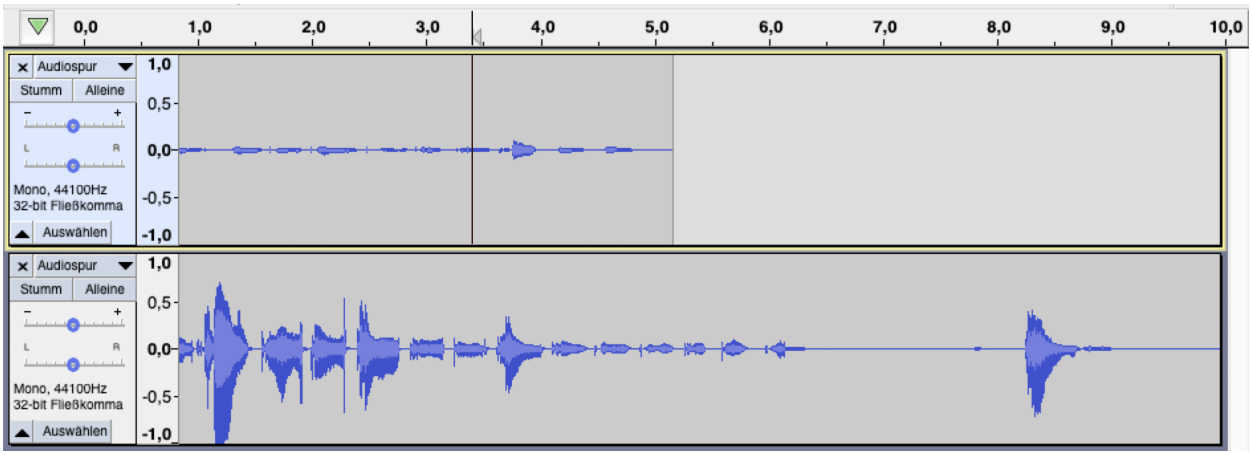


Figure 3.10: Audio tracks in the free audio editor *Audacity*. The visualization of the user's navigation in the automatic bookmark plug-in has been inspired by the typical track visualization of audio editing software

- **Element:** The program element containing the bookmark (for example, a class, method, or constant).
- **Content:** The line's content.

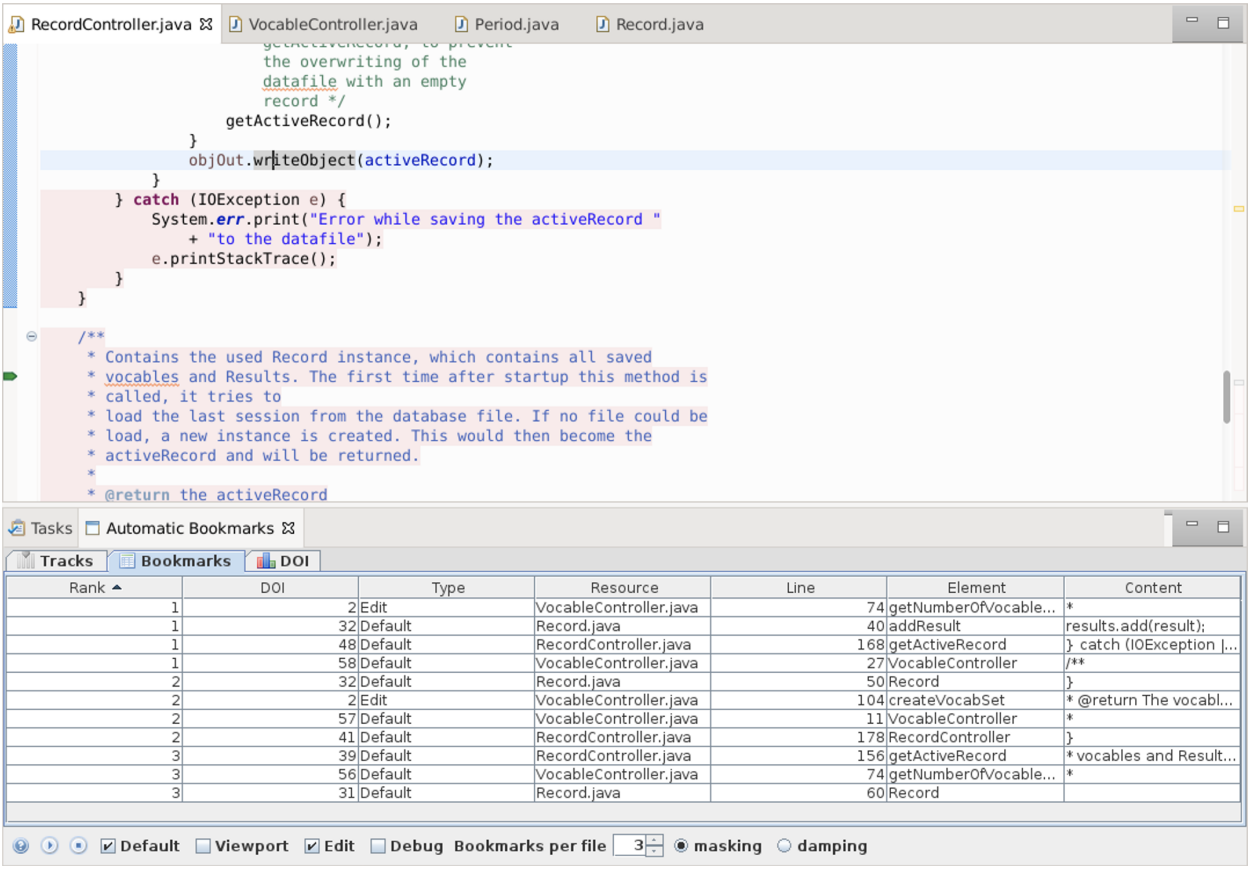
A configuration panel at the view's bottom allows the user to configure the plug-in. By unchecking checkboxes, entire bookmark categories can be hidden. Also, the number of bookmarks per file (n) can be increased or decreased, and the user can decide whether the plug-in shall apply the mask or the damp method. The panel is not part of the tabbed view, and hence it is visible and editable whenever the view is visible, regardless of the selected tab.

Tracks

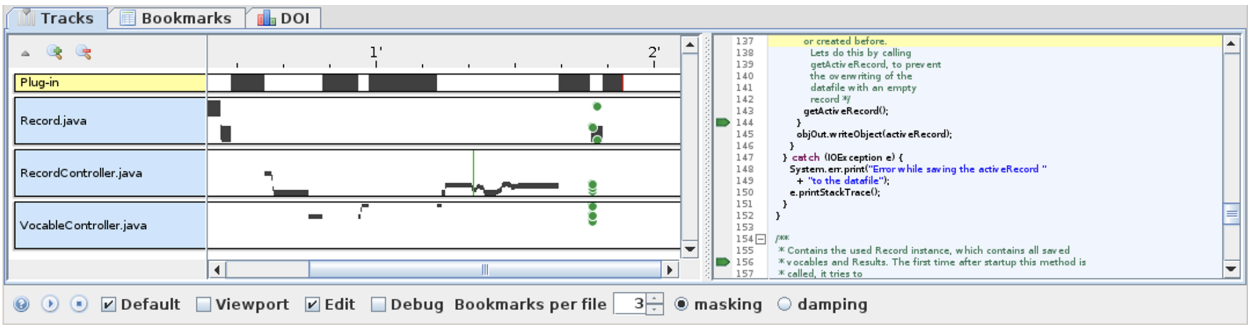
The 'tracks' visualization is inspired by the interface commonly found in audio editing software, featuring a stack of audio tracks that can be composed (see Figure 3.10 for an example). In terms of automatic bookmarks, each 'track' represents a source file. Instead of a sound signal, the track shows the visible code range over time. Thus, users can retrace their journeys through the project. Hovering on the tracks will load the excerpt visible at the selected time to a preview window next to the track panel.

DOI

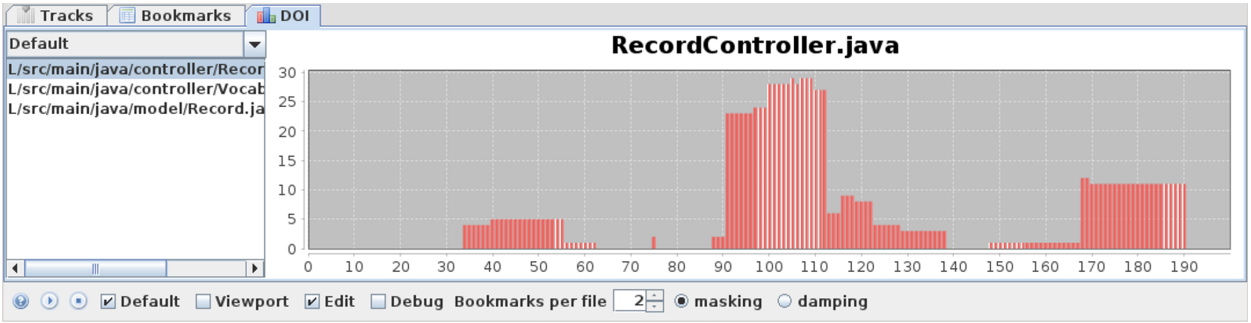
The user interface also allows inspecting the raw DOI data in a bar chart with the line numbers on the x-axis and the corresponding DOI on the y-axis.



(a) Automatic bookmarks appear as green markers in the editor's left margin. A reddish shadow in the source code marks the region belonging to the bookmark. A sortable table in the Automatic Bookmark view lists all automatic bookmarks currently visible in the project



(b) The 'Tracks' tab shows the user's scroll behavior. The gray areas represent the source code excerpt visible in the editor at the respective time. Green dots indicate the creation of an automatic bookmark (only the current bookmarks are shown)



(c) Another tab shows the DOI raw data

Figure 3.11: The plug-in's user interface: Automatic bookmarks are displayed in the editor; a view provides more detailed information in three tabs, with a configuration panel offering controls for filtering, etc., visible no matter what tab is selected

4 Evaluation

In an attempt to finally answer the research questions, a series of controlled experiments with software developers have been conducted. This chapter explains the method and discusses the results.

4.1 Method

Environment

To spare the participants an error-prone installation of the plug-in but to still allow remote participation, a terminal server was set up. The terminal server ran on a Linux VServer providing an 8 vCore CPU and 32 GB RAM guaranteed with *Ubuntu 18.04.5 LTS* installed. The server was hosted in Germany.

Each participant was given access to an individual user account on the terminal server via *Xrdp* (Neutrinolabs 2020), an open-source Linux implementation of Microsoft's *Remote Desktop Protocol* (RDP). Windows users did not have to install a client as the application *Remote Desktop Connection* is already included in standard Windows installations. Mac users could, for example, install the official *Microsoft Remote Desktop* app from the Apple App Store. RDP applications for Linux are also available. Actually, at least one of the participants used *rdesktop* on Linux. After he corrected the initially poor resolution manually, it worked as fine as the Microsoft client.

For security reasons, the server only accepted local RDP connections, which is why the participants were required to establish an SSH tunnel before they were able to start the RDP session. Windows 10 users can easily do this via Windows' *cmd* command-line interpreter, using the very same syntax like on Unix-like systems.

To ease the connection process, Windows 10 users were provided with two files: A batch file helping the user to establish the SSH connection to the Linux server, and an RDP file. Opening the latter starts the Windows application Remote Desktop Connection with the hostname and user name already configured. Thus, both in the command-line interpreter and the RDP tool, the participants only had to enter the bare minimum of data and commands manually

Table 4.1: Limits of the Linux server according to `ulimit -a`

Parameter	Value
core file size (blocks)	0
data seg size	∞
scheduling priority	0
file size (blocks)	∞
pending signals	1 545 097
max locked memory	16 384 kB
max memory size	∞
open files	1024
pipe size (512 bytes)	8
POSIX message queues	819 200 B
real-time priority	0
stack size	8 192 kB
cpu time	∞
max user processes	62 987
virtual memory	∞
file locks	∞

(such as the password and some confirmations). See Chapter A for the instructions sent to the participants.

The RDP tool adapts to the screen size of the host device and presented the participants with a well-formatted Ubuntu desktop in a suitable resolution. For a graphical user interface the lightweight *Xfce* desktop environment was installed on the Ubuntu server.

Once logged into the terminal server, the participants were asked to click on a symbolic link on the desktop, which started a survey in the web browser.[1] The survey was provided by the free web app *LimeSurvey* (LimeSurvey 2020) locally installed on the machine. During the survey, the participants were invited to start the *Eclipse IDE for Java Developers* version 2020-09 (4.17.0) with the automatic bookmark plug-in already installed and its view opened. Java was provided by Oracle's *Java SE Development Kit 11.0.8* (JDK 11.0.8).

Task

Source code

Figure 4.1: The vocabulary trainer

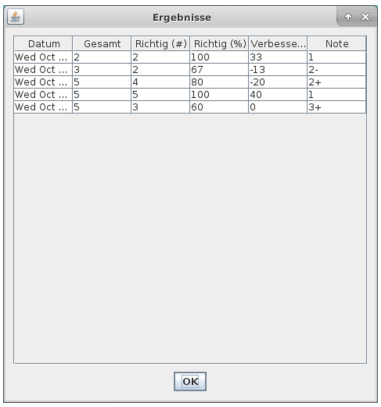


Figure 4.2: Requested change in the user interface: The participants were asked to add a column displaying the improvement (in percent) to the dialog displayed after training a set of vocables (fifth column)

The participants were confronted with the source code of an unfamiliar software system written in Java. The code implements a simple vocabulary trainer. It was written by the author and fellow students during a class on technical documentation in the computer science master program several years ago. The program served as an example system for discussing different aspects of documentation.[2]

The complexity and seriousness of the software can be located somewhere between very simple single-class examples on the one hand and professionally developed software systems on the other hand. Asking the participants to edit this system forced them to assess the bookmark plug-in in a realistic environment without overstraining them.

The source code was entirely documented with *Javadoc* and formatted using *checkstyle*, thus meeting the vast majority of *Sun's* coding conventions. The repository initially also contained *JUnit* test cases.[3] All external documentation had been removed from the

[1] Initially, *Firefox* was installed but due to memory issues it has been replaced by the lightweight *Falkon* browser from the third session on. Section 4.2 discusses the problems more detailed.

[2] Later, the group had to swap projects with another group in order to extend an unknown application using the given documentation of the original developers.

[3] The third participant actually executed the tests and experienced failing test cases. Apparently, the test cases were designed to run in a continuous integration environment, without a human editing the tool's database as well. Hence, to avoid further irritation, most test classes were removed for the remaining sessions.

repository. An architectural description for example would have made the exercise too easy.

On the start of Eclipse, the class `Main` containing the program's `main` method was opened to offer an entry point to the participants.

Assignment

The participants were asked to extend the statistics dialog presented at the end of each vocabulary training. They should add a column containing the improvement (in percent, positive or negative) to the prior training 'session.' Figure 4.2 shows the user interface of an example solution. To comply with the architecture, changes in three Java source files and one properties file (string resources) were necessary. However, many different and much simpler solutions exist. A detailed description of this task was provided by the survey tool after completing the pretest questionnaire (see Chapter B).

Setting

The questionnaire opened by the participants after their log-in to the terminal server contained eight pages—with both forward and backward navigation allowed. The entire questionnaire can be found in the appendix (Chapter B). The list below gives a summary of the questionnaire.

1. In the beginning, the participants were welcomed and informed about what to expect. They were also asked to sign an informed consent before they were able to carry on.
2. The first survey page asked some demographic questions (gender, age, education, occupation).
3. On the third page, the participants were asked to share information on their practical experience in programming and related areas.
4. Another page asked about the usage of bookmarks in IDEs. These questions were very similar to some questions asked by Guzzi et al. (2011). Because Guzzi et al. found that many developers do not even know bookmarks exist, a very short introduction into the 'classic' bookmark function was given at the top of the page, including two images of a bookmark in Eclipse and IntelliJ IDEA.
5. The fifth page introduced the automatic bookmark plug-in to the participants. This page did not contain any questions.
6. After the introduction of the plug-in going to be assessed, the programming task was explained. This page also did not contain questions. Instead, at the bottom of the page, the

participants were invited to minimize the browser window and launch Eclipse for starting the coding.
7. After finishing the assignment, in this last step, the participants were invited to share their opinions on the plug-in and its influence on their performance.
8. A final page thanked the participants and explained how to disconnect from the server.

The demographic questions and the questions asking about programming experience were in parts taken from the questionnaire given to the participants of the initial Mimesis study outlined in Section 3.2.

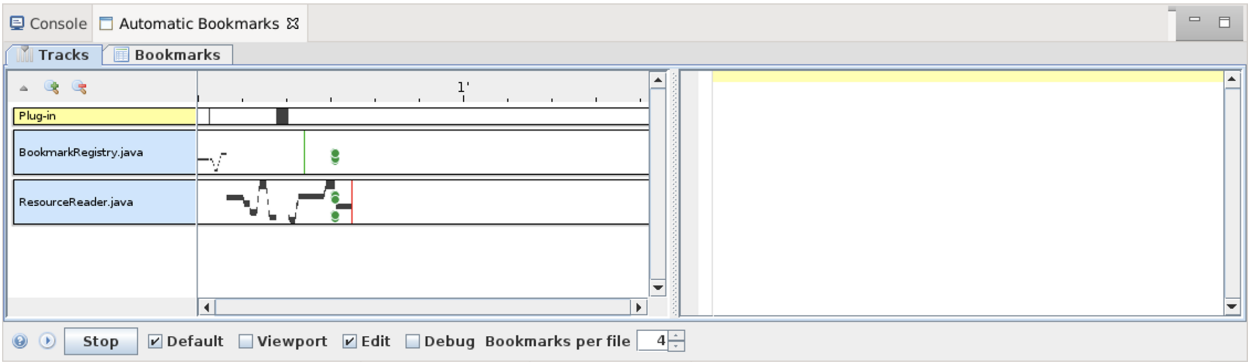
To avoid a long and possibly complex, written introduction to the plug-in, it was decided to cut down its functionality. The goal was to test the plug-in concept, not the participants' patience. For this reason, all references to the DOI model had been removed. Thus, neither a DOI tab showing the raw metric data in diagrams nor a DOI column in the table could be found during the study. To allow the users to sort the bookmarks by significance without the DOI value, each bookmark was given a 'rank,' unique for each file and type. The radio buttons for changing between mask mode and damp mode were also removed—the mask mode was applied all the time. These adjustments made it possible to describe the plug-in in just a few sentences. Figure 4.3 shows the state of the plug-in in the study.

The tab opened after starting Eclipse ('Tracks' or 'Bookmarks') can be set via a file in the user's home directory.[4] The user accounts on the Linux server were alternatingly configured with one of both tabs to ensure that both tabs have been visible to participants during the study, even if they would not change the visible tab.

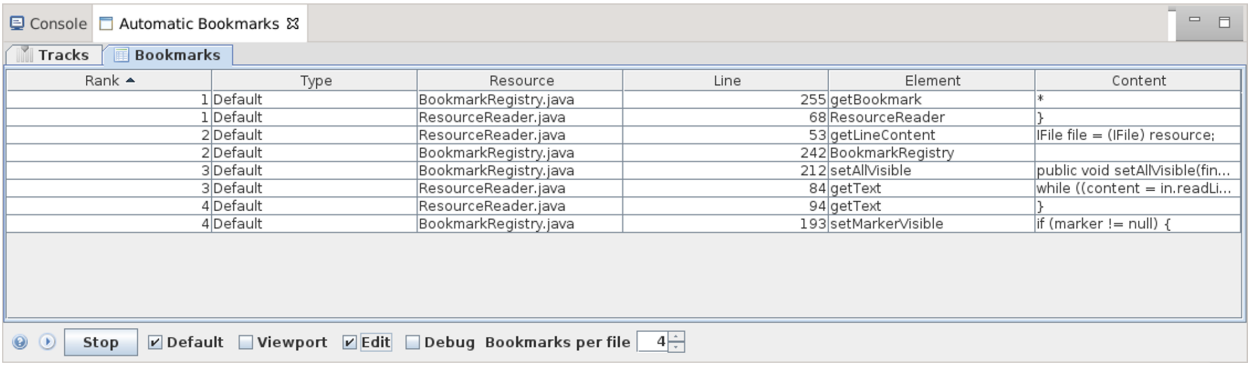
The setup contained an equal launch configuration for all participants allowing them to execute the vocabulary trainer by just clicking on the launch button in the Eclipse menu bar. Also, a sample vocable set was already imported enabling the participants to test the software without having to enter vocables first.

The participants were told to press the Stop button in the plug-in view (and confirming a dialog) when they were finished with the assignment. On pressing this button, the plug-in was stopped and the Mimesis recording was saved to the local Git repository of the edited vocabulary trainer project. Pausing the recording by pressing Stop with a delayed restart was not possible, as stopping results in a dirty repository that Mimesis cannot deal with. Later, each recording—along with the corresponding code changes—was

[4] The plug-in searches for a file named `ab_tab.txt`, containing an integer denoting the requested tab index.



(a) The tracks tab



(b) The bookmark (table) tab

Figure 4.3: State of the plug-in user interface in the experiment: All references to the underlying DOI model as well as advanced controls (such as those for switching from masking to damping) have been removed

pushed manually to separate branches (one for each participant) on the Git server.

The recordings also include events related to the plug-in to retrace the participants' usage of the plug-in.

4.2 Results

Eight software developers participated in the study (in the following named A to H). Most of them managed to start the RDP session without any difficulty[5] and the survey software in the web browser worked trouble-free in most cases—only one participant reported a time-out error when he returned from Eclipse to the browser, which forced him to answer the pretest questions again.

[5] One participant had difficulty with entering the password as he could not distinguish the capital *i* from the lower case *l* in the sans-serif font used for the instructions. Another participant reported that the remote desktop was very small. In fact, at least three developers established the RDP session from a Linux machine, which was hardly explained in the provided instructions. The paper did not explain how Linux users can increase the screen resolution manually (the Microsoft tools for Windows and Mac adapt automatically).

Unfortunately, the first two participants faced a severe Java memory issue. In both cases, Eclipse announced its inability to create another native thread.[6] In the first session, the problem occurred directly after launching Eclipse. Rebooting the server solved the problem, but forced the affected participant to answer the pretest part of the questionnaire again. The second participant was unable to launch the vocabulary trainer inside the running Eclipse environment. He solved the assignment without executing the edited program. Nevertheless, he submitted a correct solution.

Unfortunately, modifying the size of Java's memory allocation pool, the thread stack size or the maximum number of processes per user did not have a lasting effect.[7] Instead, replacing *Firefox* as the web browser for the survey with the *Falkon* browser finally did the trick and allowed to conduct the study as intended. Though, during most of the experiments, the author was contactable via telephone or instant messaging services for possible technical assistance—which did not include any form of observation. After changing the browser, Eclipse worked properly in all following sessions. Two participants just did not stop the recording themselves. After the sessions, the author had to log into the user account to save the recording.

One participant reported that the plug-in view disappeared when he made use of the debugger because Eclipse changed from the default Java perspective to the debug environment. According to his answers in the questionnaire, the participant had no experience in using Eclipse. Via online chat, he was guided back to the Java perspective where he was able to stop the recording, eventually.

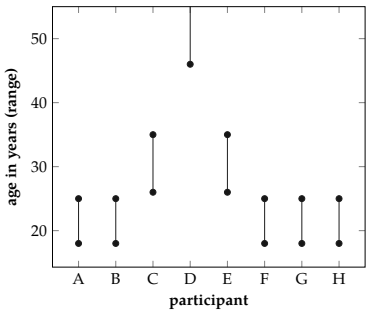


Figure 4.4: Age distribution of the developers who participated in the study

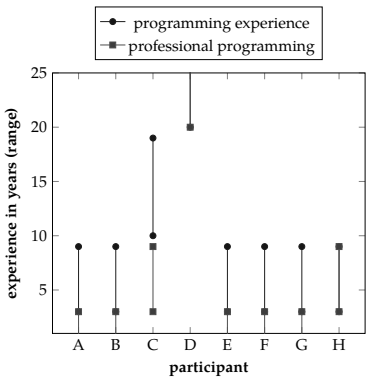


Figure 4.5: Programming experience of the participants

Demographic Characteristics and Experience

Five participants were younger than 26, two of them were under 36 and one older than 45 years (Figure 4.4). The oldest participant was a self-employed software developer with a master's degree (diploma) and more than 20 years of programming experience. Two participants were undergraduate students, another two were graduate students, and two participants had a bachelor's degree and were employed in industry. The only female participant holds a master's degree and was occupied at the university. Most of the participants had less than ten years of programming experience (see Figure 4.5).

[6] The exception message was: `java.lang.OutOfMemoryError: unable to create native thread: possibly out of memory or process/resource limits reached`

[7] The initial and maximum heap size of Eclipse were set to `-Xms1G` and `-Xmx2G`. Table 4.1 contains the system's limits.

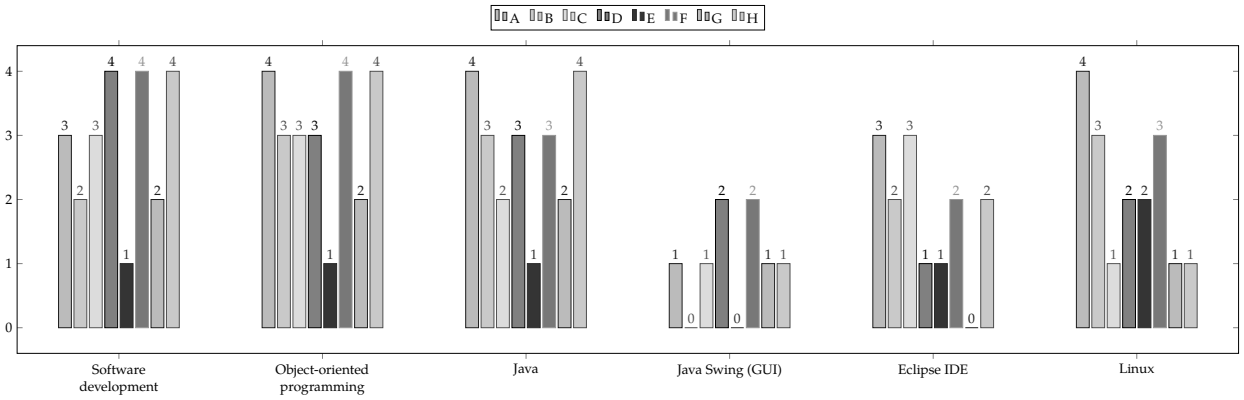


Figure 4.6: The participants' self-assessed experience in areas related to the task (0: unexperienced, 4: very experienced)

Figure 4.6 shows the participants' experience in areas that were relevant to the assignment. Most participants were at least medium experienced software developers, but their experience in UI development with Java Swing was rather poor, as was the participants' experience with the Eclipse IDE. Actually, only two participants use Eclipse frequently. In contrast, six developers use *IntelliJ IDEA* on a regular basis. Other answers were *Android Studio*—which is based on IntelliJ—, *NetBeans*, *Visual Studio* as well as *RStudio*.

Bookmark Usage

According to the answers given in the survey, only one of the participants uses bookmarks often, one rarely, and the remaining six never. The developer who likes to set bookmarks stated he uses them to mark code locations for a short time. Figure 4.7 shows the reasons why the other developers do not use bookmarks. Confirming the result of Guzzi et al. (2011), four of them declared, they did not even know bookmarks existed. One participant used the text field and explained that he does not know why bookmarks shall be useful, at all, as IDEs offer shortcuts for jumping to class and method definitions (translation): "If classes/methods are kept short, for what do people need bookmarks?"

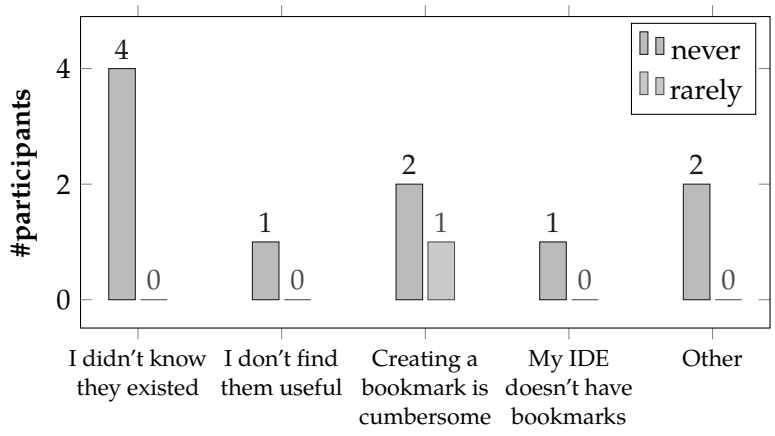
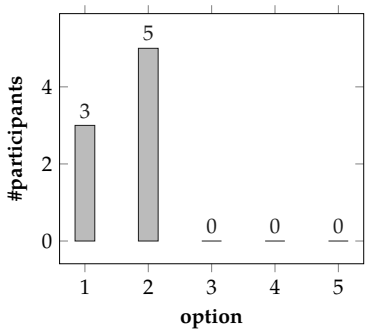
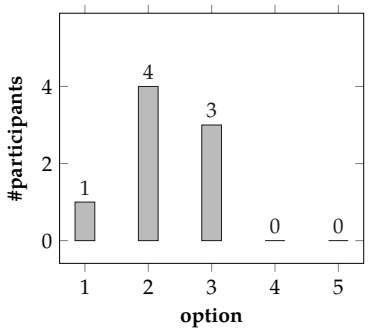


Figure 4.7: Reasons why the participants nerver or rarely use boomarks



(a) How intensivly did you use the automatic bookmarks? (1: little, 5: very much)



(b) To what extent did the automatic bookmarks support you in fulfilling the assignment? (1: not at all, 5: very much)

Figure 4.8: The participants statements concering their own plug-in usage and its support. The plots show the number of participants per answer

Posttest Results

Six participants stated, they were able to solve the assignment. Testing the solutions confirmed these statements. However, differences in the quality of the solution exist. One participant was unsure and one was unable to submit a working solution. The one who was unsure was the second participant who could not execute the program due to a memory issue. The participant who was unable to solve the assignment was the one with the most programming experience. He managed to change the user interface but stated he had problems to formulate the required calculation.

Being asked about their usage of the automatic bookmarks, the participants answered, they did not use them very much (Figure 4.8). Though, three participants thought, the plug-in gave them medium support.

The answers to the open question asking for explanations for the usage, or rather non-usage, were inductively coded in order to analyze them. The coding can be found in Chapter C.

One participant stated, he did not fully understand, how the plug-in works. Prior to the assignment, the participant stated he has low experience in (object-oriented) software development and Java. He also was the only participant who said to have no experience at all in using Eclipse. Two participants stated, they were not used to (classic) bookmarks. One of them explained this is why she is used to memorizing the locations relevant to her task. The other one wrote, he is unsure how to use bookmarks, because he never used them before.

Both undergraduate students and another participant answered the assignment was not complicated enough, which is why they felt no need for any tool support. Two participants thought the tool might be useful when working on larger software projects. At least three participants recalled, that the automatic bookmark plug-in helped them to find a relevant location. One of them explained, he had difficulty identifying the required bookmark from the table, but “wild clicking” succeeded. Another one stated, he once forgot a class name and could find the required class in the table faster than in the package browser. Two participants found that not all of the generated bookmarks were relevant to them. One suggested, a five-minute phone conversation, he had during his session, might be the reason, why the class `MainFrame` constantly was the first entry in the table, even though it was not relevant to him, at all.

Two developers complained about missing structure in the plug-in. According to his answer to the question asked later, under which circumstances he would take automatic bookmarks into account, one of them likes to search for program elements in the tree view

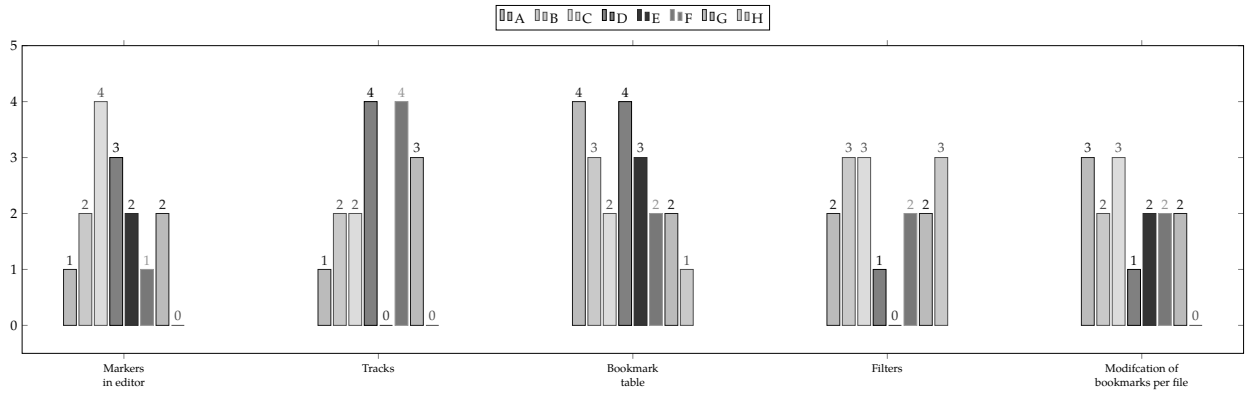


Figure 4.9: The participants' assessment of the plug-in's components (0: not helpful at all, 4: very helpful)

showing the package/directory structure of the project. As he is used to memorizing positions in the tree rather than filenames, he had problems to identify the required bookmarks in the table. The last participant, an experienced Java developer, submitted an entire list of pros and cons. Regarding the plug-in's design, he found the table has many columns, is very long, and—in contrast to source code—it does not offer any "optical clues." Moreover, this developer did not like the selection of the entire bookmark region when he clicked on entries in the table. He also disliked the tracks as they moved,[8] which made him nervous.

Figure 4.10 shows the number of `PlugInEvents` recorded in the sessions. The plug-in should record a `PlugInEvent` in the following situations:

- `TAB`: A tab change (Tracks ↔ Bookmarks).
- `GOTO`: A jump to a location in the code by clicking on a bookmark (row) in the table.
- `SORT`: A change of the tables sorting.
- `CONFIG`: A configuration change (show/hide categroy or bookmarks per file).

The first `PlugInEvent` occurrence of type `CONFIG` in the recording has programmatic reasons. All the following events should be caused by the user. However, the data suggests a defect in the recording of the plug-in-related events. While most participants caused a low to medium number of `PlugInEvents`, it seems unlikely that participant E actually caused almost one thousand events and participant G caused none, although the plug-in itself should at least produce one event. The reason for this behavior could not be determined. Hence, it does not make sense to make any statements on the participants' plug-in usage, based on the recordings.

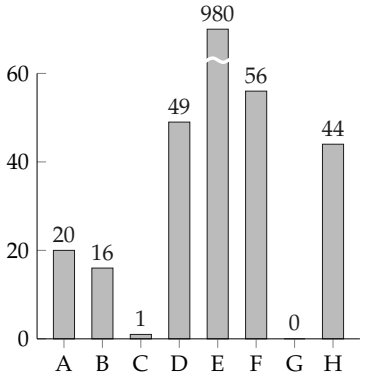


Figure 4.10: Number of plug-in event nodes in the recordings, indicating the interaction with the plug-in

[8] It is possible to prevent the tracks from moving by changing the position of the horizontal scroll bar. If the scroll bar is at the very right, the panel constantly moves with the write cursor.

The questionnaire asked how useful the participants find the different aspects of the plug-in. Figure 4.9 shows the answers. The most popular component was the bookmark table.

In the end, the developers were asked if they would use automatic bookmarks if they were available for their favorite IDE. Four participants answered they would. The developer who submitted the list of pros and cons declined, and referred to his prior statements. Three developers were unsure. Two of them would give it a try in more complex projects. One developer would need some more time to assess the concept. The developer who frequently searches the Eclipse tree view would appreciate a similar visualization for the automatic bookmarks.

The final text field for miscellaneous notes was used twice. One entry was a piece of session-related information already discussed. The other developer missed an option in the track panel, allowing him to jump to code locations (only clicking on table entries revealed the respective location in the editor).

5 Conclusion

This thesis presented a novel approach aiming to enhance the program understanding of software developers by automatically marking code locations of possible interest. The selection of these locations is based on the developers' interaction with the IDE. The concept uses the metaphor of bookmarks, as bookmarks name a feature that is—like the literature review at the beginning showed—widely neglected by software developers, which calls for a 'rethink' of the concept. In order to assess the new approach, a prototypic Eclipse plug-in implementing the automatic bookmarks has been developed. Even though the plug-in is not mature enough for being used in a developer's every-day work, it allowed assessing the concept in a controlled laboratory-like environment. The previous section has given a descriptive overview of the results of these experiments. This chapter will discuss the results with respect to the initial research questions. It will also characterize the study's threats to validity and, finally, will suggest future research topics related to this work.

5.1 Discussion

In Section 1.3 four research questions have been presented. RQ1 and RQ2 dealt with the creation of bookmarks and their presentation to the user. RQ3 and RQ4 focused on the evaluation and asked for the automatic bookmarks' acceptance compared to the neglected classic bookmarking and their benefit concerning program comprehension.

Creation and Presentation

The first two research questions RQ1 and RQ2 asked for the events eligible for the creation of automatic bookmarks and how this information can be processed to generate automatic bookmarks displayed to the user.

Section 3.1 discussed a selection of event types recorded by the Mimesis framework that can be used to detect locations possibly interesting to the developer. This choice allowed implementing a prototype of an automatic bookmark plug-in. Undoubtedly, the selection could be easily complemented by further event types,

such as tree selections or certain mouse events, for example indicating reading patterns with the cursor used as a pointer. Other events, such as launches, could improve the model but would need additional changes in the Mimesis code in order to associate a location with event types that normally do not belong to a specific location in the code. In the case of launch events, for example, it might be interesting to know which was the last location that has been edited or visited before the developer decided to do a test run.

RQ1 also asked, how the creation is influenced by the features of the corresponding code lines. The current model does not analyze the content of code lines. Of course, this would be possible and should be considered for future versions. Owing to the viewport-based approach, without such a content analysis, sometimes the model creates bookmarks for empty lines. As lines above and below the bookmarked line belong to each bookmark, in such cases the relevant part might be included in that region. Nevertheless, a method allowing to rate lines differently depending on their contents would further improve the concept and might lead to a more precise bookmarking.

Approaching RQ2, in Section 3.2 a method has been presented allowing to generate automatic bookmarks based on peaks in a DOI model. Different from previous approaches, the developed DOI model rates individual lines instead of being restricted to abstract program elements. The plug-in constantly updates the model which makes outdated bookmarks disappear and new markers appear. The developer can further adjust the bookmarking by making use of the filter options presented in Section 3.5.

Acceptance

As stated at the end of the previous chapter, only one of the participants said, he uses bookmarks regularly. However, in the survey's posttest part, four developers who use bookmarks rarely or never answered they would use automatic bookmarks if they were available for their favorite IDE. Another two developers would consider using such a plug-in. One of them would need some more time to test the tool. The other asked for a tree-like listing of the bookmarks, which reminds of the Mylar approach which offers task-specific filtering of the Eclipse tree views.

The fact that four developers stated, they did not know bookmarks existed makes it difficult to answer RQ3. Three of them said, they would use or would consider using bookmarks. As they did not know bookmarks existed, it is not possible to compare their statement regarding the automatic bookmarks to their

non-existent usage of classic bookmarks. To answer the research question it would be necessary to know if they will start using classic bookmarks, now they know this function exists.

On the other hand, only three developers, knowing bookmarks existed prior to the study, would or might use the automatic bookmarks. Thus, based on the statements of the eight participants, RQ3 cannot be answered. However, there is some evidence suggesting that developers do show a greater acceptance towards the use of automatic bookmarks than they do towards the classic bookmarks.

Benefit

RQ4 asked whether automatic bookmarks can support program comprehension. The study was not designed to measure any performance. Three developers even stated the codebase of the vocabulary trainer was too small to require any advanced tool assistance. Thus, this evaluation must rely on the answers given by the participants. Three of them stated, the automatic bookmarks gave them medium support. Four thought the plug-in helped a little bit. The participant who stated he did not fully understand how the automatic bookmarks worked answered the plug-in did not assist him at all. Asked to elaborate on their statements, three participants wrote, the automatic bookmarks helped them to find locations they wanted to revisit. Though, participants complained about irrelevant bookmarks and flaws in the interface like a crowded table, unneeded text selections when jumping to bookmarks, as well as the moving tracks.

To sum up, the results suggest that automatic bookmarks can support comprehension tasks. However, it is difficult to make any statements on a tool's benefit if the participants who were supposed to test it did not really try it—none of the participants stated to have used the automatic bookmarks more than a little bit. Roehm et al. (2012) presented some ideas about why developers might not make use of program comprehension tools (see Section 2.3). Apparently, at least one of the participants found the bookmarks too abstract and did not understand the concept. Roehm et al. also suggested a fear of training and familiarization efforts as well as a lack of trust in the new tools. These are problems most tool developers probably face in the beginning.

5.2 Threats to Validity

All of the participants were personally known to the author. Three of them were undergraduate students from the same research group dealing with Mimesis or connected topics. For this reason, there might be a social desirability bias in the answers to the posttest questions asking for the usability of the plug-in. On the other hand, a close relationship with the researcher might also motivate participants to submit a proper solution or give decent feedback.

Another threat is given by the fact that the sample contains many young developers with rather small experience in professional, non-academic software development. Young developers might be more open-minded about new tools, but the lack of experience might also have biased the results.

Furthermore, the remote participation caused different conditions in each session. Of course, the accounts on the Linux server were set up equally. But establishing the RDP connection was different depending on the host system. Hence, for some participants getting started was more cumbersome than for others. Also, the RDP window adapted to the host's screen size (on Windows and Mac), or had a low resolution on Linux (if not configured manually), which led to different experiences of the plug-in.

Before they launched Eclipse, the participants were interviewed on their usage of bookmarks. Moreover, the new concept was introduced as automatic *bookmarks*. Both may have influenced the participants' reception of the assessed plug-in. Some of the statements indicate that participants did not understand the automatic bookmark as an own concept, only loosely based on the classic, neglected bookmarks, but as some kind of extension of these bookmarks they were not used to work with. Probably it would have been better to ask questions about bookmark usage in the questionnaire's posttest part or to 'hide' them in a series of similar questions on other IDE features, so the participant does not recognize the bookmark focus.

Lastly, the software system in the study was not an industry-scale project. Neither was the change task a real-world problem. As stated above, some participants even thought this was the reason why they did not make use of the automatic bookmarks a lot.

5.3 Future Research

The experiments showed that automatic bookmarks have the potential to support developers' program understanding. And even more important, the study suggests that developers would consider using a well-designed automatic bookmarks tool. These results motivate further research efforts on this topic.

The following list gives some ideas for possible conceptual improvements in the automatic bookmark model:

- DOI increases with respect to the features of the respective code lines (empty lines, certain program elements, definitions, etc.)
- Also considering information that is not just based on user interaction, such as dependencies, calls, etc.
- Proactive handling of line differences (insertions and deletions): What should happen to the DOI values of deleted lines, and should a new line really 'inherit' the DOI value of the previous content at this location?
- Emphasizing locations that have been visited many times (not just summing up the visit durations)
- The 'radius' of the bookmark regions is based on the current size of the editor window, which may change during the task: How should this affect the size of the bookmark regions? Should the region size really depend on the window dimensions?

As a side result, this thesis showed the potential of remote participation in software engineering studies. A more stable terminal server allowing parallel participation combined with an improved connection concept such as a browser-based solution or scripts for fully automated connection on each platform could provide access to massive sample sizes with participants distributed around the entire globe.

But also an evaluation in real-world software engineering workplaces is necessary to give profound evidence for a tool's benefits. A study over a longer period of time would allow participants to familiarize themselves with the new plug-in which might give the tool a better chance to demonstrate its effects on the developers' work.

Of course, such a study would also require improvements in the plug-in. For example, currently, it does not support any kind of break in the comprehension task; neither 'idle times' caused by a sudden phone call or a developer's urge to fill up the coffee mug, nor larger breaks like the end of the working day or even an entire weekend. Thus, the tool would both need to stop increasing

DOI values during idle times (whether automatically or just by pressing a pause button) and had to restore sessions when the IDE is launched again. Both could be easily realized as Mimesis already offers an interface for parsing the XML recording file containing all relevant events. However, the system might need enhanced data structures as, currently, neither Mimesis nor the automatic bookmark plug-in is designed to record sessions lasting more than a few hours maximum. For a realistic study also converting the plug-in to be used in other popular IDEs should be considered as the participants' answers indicate that developers are not necessarily used to work with Eclipse.

Lastly, for any future research of the automatic bookmarks, a new name should be considered for the concept—at least in all communication that might influence experiments. A promising approach like the automatic bookmarks should not suffer from the bad reputation of outdated IDE features.

Bibliography

[S. K. Card and Nation 2002] Card, Stuart K. and David Nation (2002): "Degree-of-Interest Trees: A Component of an Attention-Reactive User Interface". In: *Proceedings of the Working Conference on Advanced Visual Interfaces*. AVI '02. Association for Computing Machinery, pp. 231–245.

[Chabanois 2018] Chabanois, Cédric (2018): Mesfavoris: A bookmarks Eclipse plugin. URL: `https://github.com/cchabanois/mesfavoris` (visited on 27/11/2019).

[Eclipse Foundation 2020] Eclipse Foundation (2020): Eclipse Mylyn. URL: `https://projects.eclipse.org/projects/mylyn` (visited on 30/11/2020).

[Fjeldstad and Hamlen 1979] Fjeldstad, R. K. and W. T. Hamlen (1979): "Application Program Maintenance Study. Report to our Respondents". In: *Proceedings of the GUIDE 48*.

[Flatt and Maison 2011] Flatt, Andy and Mickael Maison (2011): Best practices for developing Eclipse plugins. URL: `https://www.ibm.com/developerworks/opensource/tutorials/os-eclipse-plugin-guide/index.html` (visited on 18/02/2020).

[Guzzi et al. 2011] Guzzi, Anja, Lile Hattori, Michele Lanza, Martin Pinzger and Arie van Deursen (2011): "Collective Code Bookmarks for Program Comprehension". In: *2011 19th IEEE International Conference on Program Comprehension*, pp. 101–110.

[Kersten and Murphy 2005] Kersten, Mik and Gail C. Murphy (2005): "Mylar: A degree-of-interest model for IDEs". In: *Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pp. 159–168.

[Knuth 1984] Knuth, Donald E. (1984): "Literate programming". In: *Comput. J*, pp. 97–111.

[Ko et al. 2006] Ko, Amy, Brad Myers, Michael Coblenz and Htet Aung (2006): "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks". In: *IEEE Transactions on Software Engineering* 32, pp. 971–987.

[LimeSurvey 2020] LimeSurvey GmbH (2020): LimeSurvey Community Edition. URL: `https://community.limesurvey.org` (visited on 08/11/2020).

[Maalej et al. 2014] Maalej, Walid, Rebecca Tiarks, Tobias Roehm and Rainer Koschke (2014): "On the Comprehension of Program

Comprehension". In: *ACM Transactions on Embedded Computing Systems* 23 (4), 31:1–31:37.

[Martin 2009] Martin, Robert C. (2009): *Clean Code. A Handbook of Agile Software Craftsmanship*. Upper Saddle River: Prentice Hall.

[Murphy et al. 2006] Murphy, Gail C., Mik Kersten and Leah Findlater (2006): "How are Java software developers using the Elipse IDE?" In: *IEEE Software* 23 (4).

[Neutrinolabs 2020] Neutrinolabs (2020): xrdp - an open source RDP server. URL: `https://github.com/neutrinolabs/xrdp` (visited on 21/10/2020).

[Piorkowski et al. 2013] Piorkowski, David J., Scott D. Fleming, Irwin Kwan, Margaret M. Burnett, Christopher Scaffidi, Rachel K.E. Bellamy and Joshua Jordahl (2013): "The Whats and Hows of Programmers' Foraging Diets". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 3063–3072.

[Pirolli and S. Card 1999] Pirolli, Peter and Stuart Card (1999): "Information foraging". In: *Psychological Review* 106 (4).

[Proksch et al. 2018] Proksch, Sebastian, Sven Amann and Sarah Nadi (2018): "Enriched Event Streams: A General Dataset For Empirical Studies On In-IDE Activities Of Software Developers". In: *Proceedings of the 15th International Conference on Mining Software Repositories*, pp. 62–65.

[Roehm et al. 2012] Roehm, Tobias, Rebecca Tiarks, Rainer Koschke and Walid Maalej (2012): "How Do Professional Developers Comprehend Software?" In: *Proceedings of the 34th International Conference on Software Engineering*, pp. 255–265.

[Schedenig n.d.] Schedenig, Marian (n.d.): Quick Bookmarks (Eclipse Plug-in). URL: `http://marian.schedenig.name/projects/quickbookmarks/` (visited on 27/11/2019).

[Sillito et al. 2008] Sillito, Jonathan, Gail C. Murphy and Kris De Volder (2008): "Asking and Answering Questions during a Programming Change Task". In: *IEEE Transactions on Software Engineering* 34 (4), pp. 434–451.

[Storey, Cheng et al. 2007] Storey, Margaret-Anne, Li-Te Cheng, Janice Singer, Michael Muller, Del Myers and Jody Ryall (2007): "How Programmers can Turn Comments into Waypoints for Code Navigation". In: *IEEE International Conference on Software Maintenance, ICSM*, pp. 265–274.

[Storey, Ryall et al. 2008] Storey, Margaret-Anne, Jody Ryall, R. Ian Bull, Del Myers and Janice Singer (2008): "TODO or To Bug: Exploring How Task Annotations Play a Role in the Work Practices of Software Developers". In: *2008 ACM/IEEE 30th International Conference on Software Engineering*.

Instructions A

Anleitung

zur Teilnahme an der Evaluation zur Masterarbeit
„Supporting Program Comprehension by Automatic Bookmarks“

Zur Teilnahme an der Studie erhalten Sie über eine Remotedesktopverbindung Zugang zu einem Linux-Sever mit einer vorbereiteten Eclipse-Installation und einem Fragebogen.

Sobald die Verbindung zum Terminal-Server hergestellt wurde, öffnen Sie bitte die Verknüpfung **Befragung** auf dem übertragenen Desktop. Es werden Ihnen einige Fragen zu Ihnen und Ihrer Erfahrung gestellt. Anschließend werden Sie gebeten, unter Zuhilfenahme eines neuen Plug-ins eine Programmieraufgabe zu lösen, die im Verlauf der Befragung erläutert wird. Abschließend werden Sie noch um Ihre Einschätzung gebeten.

Im Folgenden wird der Verbindungsaufgabu mit dem Sever unter Windows 10 erläutert. Hinweise für die Teilnahme über ein anderes Betriebssystem finden sich am Ende des Dokuments.

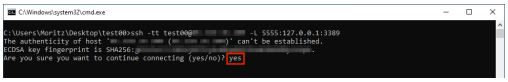
> Bei Problemen oder wenn Sie Hilfe benötigen, zögern Sie bitte nicht, eine E-Mail an Moritz Weinig zu senden (<email>).

Windows 10

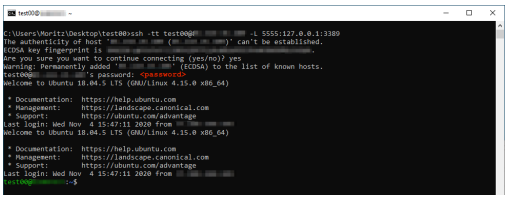
Bitte speichern Sie das angehängte ZIP-Archiv auf Ihrem Rechner. Extrahieren Sie das Archiv anschließend (*Rechtsklick* > *Alle extrahieren...*). Die beiden im Paket enthaltenen Dateien unterstützen Sie beim Verbindungsaufbau.

1. SSH-Tunnel aufbauen

Da der Server aus Sicherheitsgründen nur lokale Verbindungen akzeptiert, muss zunächst ein SSH-Tunnel aufgebaut werden. Führen Sie dazu die Batchdatei **<username>_A** durch einen Doppelklick aus.



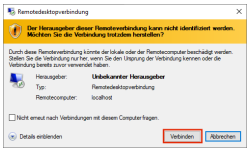
Bestätigen Sie auf Nachfrage durch die Eingabe yes, dass Sie mit der Verbindung mit dem unbekannten Rechner fortfahren möchten und geben Sie anschließend das Passwort <password> ein.

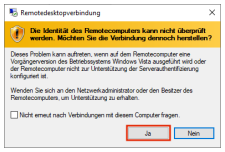


Datenverkehr, der sich an den Port 5555 Ihres Rechners richtet, wird nun auf den Port 3389 des Servers umgeleitet, ebenso andersherum. **Bitte schließen Sie dieses Fenster erst, wenn Sie die gesamte Teilnahme abgeschlossen haben!**

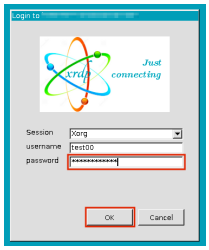
2. Remotedesktopverbindung herstellen

Öffnen Sie nun die Datei <**username**>**_B**, um die Remotedesktopverbindung herzustellen. Sie werden nun darauf hingewiesen, dass der Herausgeber der Verbindung sowie die Identität des Remotecomputers nicht überprüft werden können. Bitte klicken Sie Verbinden bzw. Ja, um dennoch fortzufahren.





Geben Sie anschließend das Passwort <password> ein und klicken Sie OK.



Andere Betriebssysteme

Unter Betriebssystemen (oder älteren Windows-Versionen) muss die Verbindung manuell hergestellt werden.

1. SSH-Tunnel aufbauen:
   - `ssh <username>@<address> -L 5555:127.0.0.1:3389`
   - Passwort <password>
2. Remotedesktopverbindung herstellen:
   - Computername `localhost:5555`
   - Benutzername <username> und Passwort <password>

Hinweise:

- Benutzer älterer **Windows**-Versionen benötigen einen extra Client zur Herstellung der SSH-Verbindung (z. B. *PuTTY*). Für die Remotedesktopverbindung kann das gleichnamige, in Windows enthaltene Tool verwendet werden.
- **Mac**-Nutzer können u. a. den Remotedesktop-Client *Microsoft Remote Desktop* aus dem Apple App Store laden.
- Unter **Linux** kann z. B. die Anwendung rdesktop installiert werden. Nach dem Aufbau des SSH-Tunnels wird die Sitzung durch folgenden Befehl gestartet:

```
rdesktop -u <username> -p <password> -z localhost:5555
```

Questionnaire B

Asterisked questions were mandatory to proceed the questionnaire. If not stated differently, only one option could be selected in multiple-choice questions. Questions without options were open questions with text fields. Each section has been displayed on a seperate page.

Welcome

Liebe Teilnehmerin, lieber Teilnehmer,

vielen Dank für das Interesse an dieser Untersuchung, die im Rahmen meiner Masterarbeit im Studiengang Informatik an der Universität Bremen stattfindet. Nach einigen Fragen zu Ihrer Person und Ihrer Programmiererfahrung, bitte ich Sie, eine Programmieraufgabe zu bearbeiten. Diese wird später erläutert. Abschließend werden Sie zu Ihrer Einschätzungen bezüglich der Aufgabe befragt. Bitte lesen Sie sich alle Texte in diese Befragung aufmerksam durch. Sie wurden so kurz wie möglich gestaltet.

Moritz Weinig

In dieser Umfrage sind 25 Fragen enthalten.

Informed Consent

Im Rahmen einer Masterarbeit werden Maßnahmen untersucht, die das Verstehen von unbekanntem Quelltext unterstützen sollen. Dazu ist es erforderlich, während der gestellten Programmieraufgabe Ihre Interaktion mit der Entwicklungsumgebung aufzuzeichnen. Zudem werden Daten zu Ihrer Person, relevanter (Vor-) Erfahrung sowie Ihre persönliche Einschätzungen erhoben. Für die weitere wissenschaftliche Auswertung der Aufzeichnungen werden alle Angaben, die zu Ihrer Identifizierung führen könnten, verändert. Sie können Ihr Einverständnis jederzeit widerrufen.

- ▶ Ich stimme den oben genannten Bedingungen zu.*

Demographic Data

Question D1

Was ist Ihr Geschlecht?*

- männlich
- weiblich
- divers
- keine Angabe

Question D2

Wie alt sind Sie?*

- unter 18 Jahren
- 18-25 Jahre
- 26-35 Jahre
- 36-45 Jahre
- über 46 Jahre

Question D3

Was ist Ihr höchster Bildungsabschluss?*

- Abitur
- Bachelor
- Master/Diplom
- Promotion/PhD

Question D4

Wie sind Sie derzeit beschäftigt?*

- Studium
- Angestellt
- Selbstständig

Question D5

Only if question D4 was answered 'Angestellt'.

In welchem Bereich sind Sie tätig?*

- Industie/Wirtschaft
- Forschung/Hochschule

Expierence

Question E1

Wie schätzen Sie Ihre Erfahrung in folgenden Bereichen ein?*

Matrix question with options 0 to 4 for each item.

0: keine Erfahrung, 4: sehr erfahren

- Erfahrung in der Softwareentwicklung (allgemein)
- Objektorientierte Programmierung
- Java
- Java Swing (GUI)
- Eclipse IDE
- Linux

Question E2

Welche integrierten Entwicklungsumgebungen (IDEs) verwenden Sie häufig?

Multiple answers allowed.

- Eclipse IDE
- NetBeans IDE
- IntelliJ IDEA
- Android Studio
- CLion
- Visual Studio
- Apple Xcode
- Sonstiges: *Text field*

Question E3

Wie lange programmieren Sie schon?*

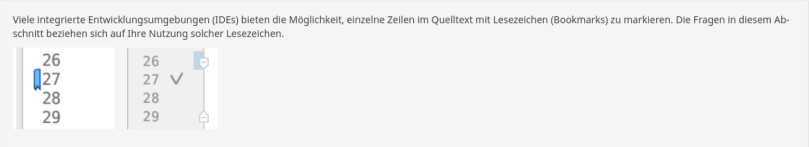
- weniger als drei Jahre
- 3-9 Jahre
- 10-19 Jahre
- 20 Jahre oder mehr

Question E4

Wie lange wirken Sie schon an größeren Softwareprojekten mit (z. B. in einem Unternehmen)?*

- weniger als drei Jahre
- 3-9 Jahre
- 10-19 Jahre
- 20 Jahre oder mehr

Bookmarks



Question B1

Wie häufig nutzen Sie Lesezeichen (Bookmarks) während der Arbeit mit einer integrierten Entwicklungsumgebung (IDE) wie z. B. Eclipse?*

- Häufig
- Gelegentlich
- Sehr selten
- Gar nicht

Question B2

Only if question B1 was answered with 'Häufig' or 'Gelegentlich'.

In welchen Situationen verwenden Sie Lesezeichen?*

Question B3

Only if question B1 was answered with 'Sehr selten' or 'Gar nicht'.

Weshalb verwenden Sie Lesezeichen sehr selten bzw. gar nicht?*

Multiple answers allowed.

- Ich wusste nicht, dass sie existieren.
- Ich finde sie nicht hilfreich.
- Das Setzen von Lesezeichen ist mühsam.
- Meine IDE hat gar keine Lesezeichen.
- Sonstiges: *Text field*

Plug-in

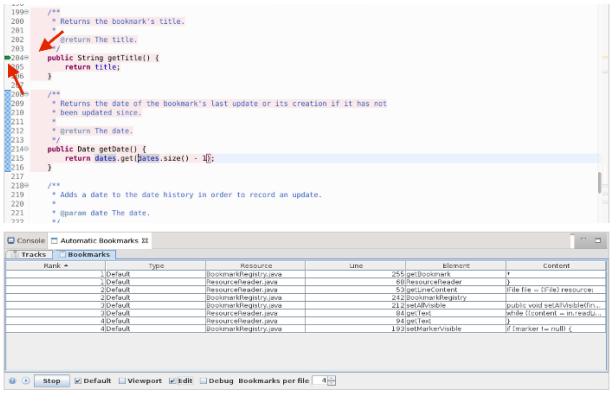
LimeSurvey

50%

Im weiteren Verlauf dieser Befragung werden Sie gebeten, ein Plug-in für die Eclipse IDE zu testen. Basierend auf Ihrer Interaktion mit der Entwicklungsumgebung bestimmt das Plug-in Stellen im Quelltext, die für Sie relevant sein könnten. Diese Bereiche werden als „automatische Lesezeichen" im Editor dargestellt und in einer tabellarischen Übersicht aufgelistet.

Automatische Lesezeichen

Ihre Arbeit mit dem Quelltext wird laufend analysiert, um potentiell wichtige Stellen für Sie automatisch hervorzuheben. Die Bereiche mit dem höchsten Ranking in einer Datei werden mit Lesezeichen markiert. Sie können dabei entscheiden, wieviele Lesezeichen maximal pro Datei eingeblendet werden sollen.

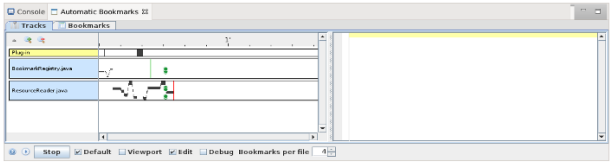


Die Lesezeichen werden für verschiedenen Kategorien erzeugt, die Sie ebenfalls einzeln ein- und ausblenden können:

- ***Viewport*** – Hier fließen die Betrachtung des Codes sowie Ihre Markierungen ein.
- ***Edit*** – Hier werden allein Änderungen am Code berücksichtigt.
- ***Debug*** – Hier werden nur Debug-Events berücksichtigt (z. B. das Setzen von Haltepunkten).
- ***Default*** – Alle Events der vorgenannten Kategorien zusammen.

Tracks

Zusätzlich bietet das Plug-in unter dem Reiter „Tracks" eine Visualisierung Ihrer Navigation durch den Quelltext.

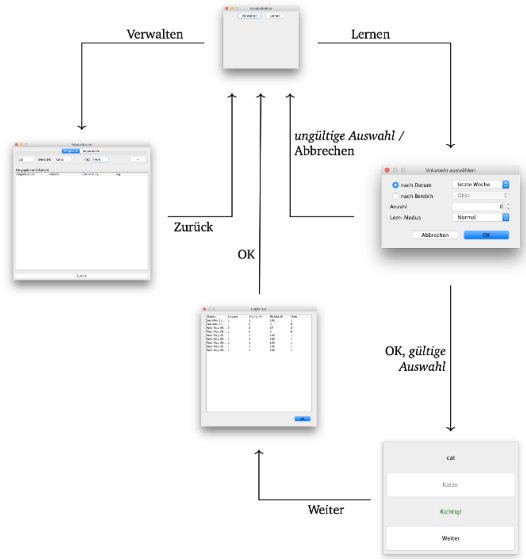


Assignment

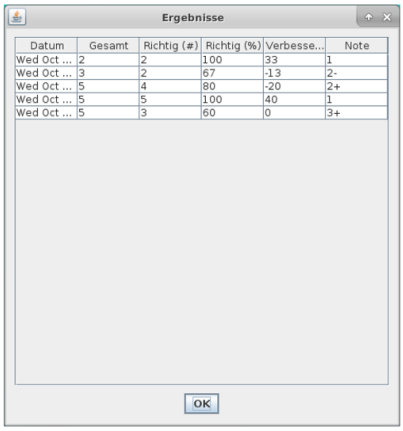
LimeSurvey

66%

Um das Plug-in unter realistischen Bedingungen testen zu können, liegt auf diesem Rechner ein Softwareprojekt bereit, das von Ihnen erweitert werden soll. Es handelt sich dabei um den Quelltext eines einfachen Vokabeltrainers. Folgende Navigationsmöglichkeiten stehen im Programm zur Verfügung:



Am Ende einer Vokabelabfrage wird eine Auswertung aller vergangenen Durchläufe angezeigt.
Ihre Aufgabe besteht darin, zu der Auswertung eine Spalte mit der Verbesserung in Prozent gegenüber dem vorherigen Durchgang hinzuzufügen. Die folgende Abbildung zeigt beispielhaft die geforderte Änderung in der GUI (der jüngste Datensatz steht ganz oben und der älteste unten):



Datum	Gesamt	Richtig (#)	Richtig (%)	Verbesse...	Note
Wed Oct ...	2	2	100	33	1
Wed Oct ...	3	2	67	-13	2-
Wed Oct ...	5	4	80	-20	2+
Wed Oct ...	5	5	100	40	1
Wed Oct ...	5	3	60	0	3+

Die Änderung muss sich dabei nur auf zukünftige Durchläufe auswirken. Zum jetzigen Zeitpunkt bereits vorhandene Datensätze müssen nicht verändert werden!

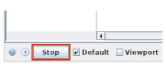
Bitte lösen Sie die Aufgabe, so gut es Ihnen möglich ist. Ziel dieser Studie ist die Evaluation des vorgestellten Plug-ins. Es gibt mehr als eine richtige Lösung und auch unvollständige Lösungen können im Rahmen dieser Untersuchung ausgewertet werden.

Sie können das Programm über Eclipse zum Testen beliebig oft ausführen oder auch debuggen.

Zu Eclipse wechseln

Bitte minimieren Sie nun diesen Browser **(nicht schließen!)** und öffnen Sie über die gleichnamige Schaltfläche auf dem Desktop Eclipse. Nach dem Programmstart informiert Sie ein Dialog darüber, dass die Bearbeitung gestartet wurde. Bestätigen Sie und beginnen Sie bitte mit der Aufgabe. Gerne dürfen Sie während der Bearbeitung zu dieser Befragung zurückkehren, um z. B. die Anleitung oder die Vorstellung des Plug-ins noch einmal zu lesen.

Wenn Sie fertig sind, speichern Sie bitte alle bearbeiteten Dateien. Klicken Sie anschließend auf den Button ***Stop*** und bestätigen Sie die Nachfrage im angezeigten Dialog. Danach kehren Sie bitte zu dieser Befragung zurück.



Sollte die „Automatic Bookmarks"-Ansicht im Laufe der Bearbeitung verschwinden (z. B. durch versehentliches „wegklicken"), können Sie diese über ***Window > Show View > Other... > Automatic Bookmarks*** wieder öffnen.

Assesment

Question A1

Konnten Sie die gestellte Aufgabe lösen?*

- Ja
- Nein
- Bin unsicher

Question A2

Only if question A1 was answered with 'Nein'.

Was hat Ihnen Schwierigkeiten bereitet?*

Question A3

Only if question A1 was answered with 'Bin unsicher'.

Warum?*

Question A4

Wie intensiv haben Sie von den automatischen Lesezeichen Gebrauch gemacht?*

1: wenig, 5: sehr stark

Question A5

Inwieweit haben die automatischen Lesezeichen Sie Ihrer Meinung nach bei der Bearbeitung der Aufgabe unterstützt?*

1: gar nicht, 5: sehr stark

Question A6

Warum?*

Question A7

Wie hilfreich schätzen Sie die einzelnen Aspekte des Plug-ins ein?*

Matrix question with options 0 to 4 for each item.

0: gar nicht hilfreich, 4: sehr hilfreich

- Markierungen im Editor
- Tracks (Darstellung Ihres Scroll- bzw. Navigationsverhaltens)
- Tabellarische Auflistung
- Filterfunktion (Ein-/Ausblenden)
- Ändern der Lesezeichen-Anzahl

Question A8

Können Sie sich vorstellen, automatische Lesezeichen zu verwenden, wenn sie als Plug-in für Ihre IDE verfügbar wären?*

- Ja
- Nein
- Vielleicht

Question A9

Only if question A8 was answered with 'Nein'.

Warum nicht?*

Question A10

Only if question A8 was answered with 'Vielleicht'.

Unter welchen Bedingungen würden Sie die Verwendung von automatischen Lesezeichen in Betracht ziehen?*

Question A11

Gibt es einen Punkt, nach dem nicht gefragt wurde, den Sie aber für relevant halten? Haben Sie Anmerkungen?

End

Vielen Dank für Ihre Teilnahme!

Sie können die Verbindung zum Terminal-Server nun trennen. Wenn Sie das Microsoft-Tool Remotedesktopverbindung verwenden, fahren Sie dazu mit der Maus an den oberen Bildschirmrand, um dessen Kontrollelemente einzublenden. Sie SSH-Verbidnung können Sie anschließend durch Eingabe des Befehls exit in der Eingabeaufforderung beenden.

Coding C

Reasons For Plug-in Usage and Non-Usage

Categories

- I did not understand, how the plug-in works
- The assignment was not complicated
- I can image, the plug-in would assist in larger projects
- I am not used to use (classic) bookmarks
- Automatic bookmarks (or some of them) were not relevant
- The plug-in helped me to find a relevant location
- Missing structure
- Idea for improvement

Answers

Participant A

Die Aufgabe war fuer mich nicht so kompliziert, dass ich die Lesezeichen benoetigt habe, um die relevanten Codestellen wiederzufinden, und es war fuer mich komplizierter, das richtige Lesezeichen anhand des Dateinamens und der Zeilennummer zu identifizieren, als die Datei in der Baumstruktur herauszusuchen und sie direkt zu oeffnen, wobei die relevante Zeile in diesem Falle dann meistens noch im Fokus war, weil sie die letzte Zeile war, mit der ich interagiert habe.

Participant B

Da ich die Lesezeichen nur in sehr geringem Maße genutzt habe, konnte ich dadurch auch nicht wirklich unterstützt werden (vielleicht wäre ich es aber, wenn ich sie genutzt hätte.)
Bei Aufgaben wie diesen sind die für die Aufgabe relevanten Stellen klein genug, dass ich nicht unbedingt ein externes Tool (wie die Autobookmarks) brauche, um mir die Stellen zu merken. Zum Beispiel habe ich mir hier ja nur ResultsDialogPanel, Result und ResultsTableModel merken müssen, wobei das schon durch die Tabs der offenen Dateien in Eclipse (quasi als Datei-Bookmarks) gegeben ist und wegen der geringen Anzahl an Zeilen in diesen Klassen kein externes Tool erfordert.
Bei größeren Projekten könnte ich mir schon eher vorstellen, dass ich die Autobookmarks nutzen würde, weil es dann zu viele Codestellen werden, als dass man sich diese merken oder mit Tabs im Vordergrund behalten kann.

Participant C

Die Lesezeichen haben dabei geholfen, bestimmte Stellen innerhalb einer Datei wiederzufinden. Da die Dateien selbst jedoch recht übersichtlich war, waren die Lesezeichen in den Dateien nicht entscheidend für die Lösung der Aufgabe. Bei komplexeren Dateien helfen diese vielleicht mehr.
Ich habe vorher nie Lesezeichen in Eclipse (oder einer anderen IDE) verwendet und bin mir unsicher, wie diese korrekt zu verwenden sind.

Participant D

Nicht alle automatisch generierten Lesezeichenwaren für mich relevant.

Participant E

Genutzt um in andere Datei zu springen.
Bin nicht gewoehnt Lesezeichen zu nutzen, deswegen habe ich mir aus der Erinnerung gemerkt, welche Dateien wichtig sind

Participant F

Normalerweise hangle ich mich durch Code, indem ich die IDE Funktionen zum Suchen von Deklaration und Aufrufen verwende (oft Strg+Mausklick).
Es war ungewohnt in der Liste der Bookmarks das richtige zu identifizieren. Wildes Durchklicken hat jedoch auch gut geklappt.

Participant G

Ich müsste mich etwas mehr einlesen wie ich sie sinnvoll verwenden kann. (Breakpoints sind ja keine automatischen lesezeichen, weil man sie manuell setzt oder?)
ich glaube ich hab noch nicht so 100% verstanden wie sie funktionieren.

Participant H

Was war gut:
+ Als ich einen Klassennamen vergessen hatte, konnte ich die Klasse in der Tabelle schneller wiederfinden im Package Browser
+ Filter wechseln funktioniert schnell und intuitiv

Wo gibt es Verbesserungspotenzial:
- "Main Frame", die Klasse, die nie geändert wurde war konstant ganz oben in der Tabelle. In mitten der Bearbeitung musste ich ein 5m Telefonat "dazwischenschieben". Möglicherweise war die Main Frame Klasse da geöffnet? Zeiten in denen der Computer nicht aktiv genutzt wird und die IDE auf einer "uninteressanten" Datei verharrt sind im Entwickleralltag jedoch häufig.
- Die Tabelle gibt optisch keine Anhaltspunkte: Suchen im Quellcode ist einfacher als Suchen in der Tabelle, da der Quellecode eine optische Struktur hat.
- Die Tabelle ist sehr lang.
- Die Tabelle hat sehr viele Spalten.
- Beim Anklicken eines Lesezeichens wurden häufig große Teile des Quellcodes markiert. Gar keine Markierung wäre mir lieber.
- Während ich mir die Tracks angucke möchte ich nicht, dass diese sich weiter bewegen. In der Zeit interagiere ich ja nicht mit dem Quellcode.
Die Bewegung während der Betrachtung macht mich nervös und hat mich davon abgehalten das Feature zu benutzen.

CD-ROM D

A CD-ROM containing source code, results and plots is attached.
The following tree lists the files and directories on the disc.

- /
  - analysis
    - timelapse
      - *Videos showing animated plots of the DOI developments (like in Figure 3.4).*
    - visits
      - *Plots showing the visits (like in Figure 3.5).*
  - answers
    - *The answers of each participant.*
  - assignment
    - master
      - *The source code given to the participants.*
    - solutions
      - *The solutions of each participant.*
    - task
      - *A reference solution by the author.*
  - checking
    - pdf
      - *Listings with colored bookmark areas (like in Figure 3.7).*
    - raw
      - *The results of the simulations as described in Section 3.4*
  - de.unibremen.informatik.autobookmarks_1.0.0.jar
    - *The plug-in version installed for the experiments.*
  - sources
    - automatic-bookmarks
      - full
        - *Source code of the automatic bookmark plug-in including all developed features (as presented in Section 3.5)*
      - master
        - *Source code of the automatic bookmark plug-in as used for the experiments.*
    - mimesis
      - *The edited Mimesis sources.*[1]
  - thesis.pdf
    - *Digital version of this document.*