UNIVERSITÉ DE MONTRÉAL

IMPROVING PROGRAM COMPREHENSION AND RECOMMENDATION SYSTEMS USING DEVELOPERS' CONTEXT

ZÉPHYRIN SOH
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
DÉCEMBRE 2015

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée:

IMPROVING PROGRAM COMPREHENSION AND RECOMMENDATION SYSTEMS
USING DEVELOPERS' CONTEXT

présentée par: SOH Zéphyrin
en vue de l'obtention du diplôme de: Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de:

M. MULLINS John, Ph. D., président
M. GUÉHÉNEUC Yann-Gaël, Doctorat, membre et directeur de recherche
M. ANTONIOL Giuliano, Ph. D., membre et codirecteur de recherche
M. KHOMH Foutse, Ph. D., membre et codirecteur de recherche
M. DESMARAIS Michel C., Ph. D., membre
M. ROBBES Romain, Ph. D., membre externe

To my family. . .

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the participation of numerous people who gave me the chance to work and discuss with them.

Dr. Yann-Gaël Guéhéneuc, you have been a special tutor both in academia and extra-academia. I am specially and sincerely grateful for your guidance, advice, and patience. During my Ph.D., I always felt confident when I discussing with you and we always came out with nice ideas. Your practical and theoretical knowledge provided me wonderful insights to achieve this thesis. Your "NPV", "why", "so what", "threat" comments helped me to improve my writing skills, and the ":-)" ones gave me a smily environment to work in :-).

I would like to express my gratitude to my co-supervisors, Dr. Giuliano Antoniol and Dr. Foutse Khomh. This thesis would not have been possible without your suitable comments and suggestions. Your advices always resulted in improvements of my work.

I would also like to thank Dr. Bram Adams for his useful comments and discussions. I greatly appreciate your availability to provide suggestions towards further exploration.

I would like to thank Pierre-Antoine and Thomas with whom I designed and ran an experiment. It was a great pleasure to work with you during your internship.

I also collaborated, discussed and shared ideas with wonderful peoples. Many thanks to the members of Ptidej Team, Soccer, MCIS, SWAT Labs. We shared together more than research, but also life.

I dedicate this work to my family, especially my lovely wife - Laure, my children Freddy and Chris for their love, unconditional support, and understanding. You are the ones who gave me the courage and strength to accomplish this dissertation.

A special thank to my mother and my father. You are the persons who give me love and what I needed to become what I am.

My thanks goes to AUF (Agence Universitaire de la Francophonie) which provided a financial support for a part of my Ph.D. study.

PUBLICATIONS

Part of the work reported in this dissertation has been published in the following venues:

International peer reviewed journal article(s):

1. **Zéphyrin Soh**, Foutse Khomh, Yann-Gaël Guéhéneuc and Giuliano Antoniol. *Noise in Interaction Traces Data and its Impact on Developers' Behaviour Studies and on the Accuracy of Recommendation Systems*. Empirical Software Engineering Journal, Springer (**EMSE**, **2015**). (**under review**).
2. Zohreh Sharafi, **Zéphyrin Soh**, Yann-Gaël Guéhéneuc. *A systematic literature review on the usage of eye-tracking in software engineering.* Journal of Information and Software Technology, Elsevier (**IST**, **2015**).

International peer reviewed conference proceedings:

1. **Zéphyrin Soh**, Thomas Drioul, Pierre-Antoine Rappe, Foutse Khomh, Yann-Gaël Guéhéneuc and Naji Habra. *Noise in Interaction Traces Data and their Impact on Previous Research Studies*. In Proceedings of the 9th ACM/IEEE International Symposium of Empirical Software Engineering and Measurement (**ESEM**), Beijing, China (**October 2015**).
2. **Zéphyrin Soh**, Foutse Khomh, Yann-Gaël Guéhéneuc and Giuliano Antoniol; *Towards Understanding How Developers Spend Their Effort During Maintenance Activities*. In Proceedings of the 20th IEEE Working Conference on Reverse Engineering (**WCRE**), Koblenz, Germany (**October 2013**).
3. **Zéphyrin Soh**, Foutse Khomh, Yann-Gaël Guéhéneuc, Giuliano Antoniol and Bram Adams; *On the Effect of Exploration Focus on Maintenance Tasks*. In Proceedings of the 20th IEEE Working Conference on Reverse Engineering (**WCRE**), Koblenz, Germany (**October 2013**).
4. **Zéphyrin Soh**, Zohreh Sharafi, Bertrand Van Den Plas, Gerardo Cepeda, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. *Professional Status and Expertise for UML Class Diagram Comprehension: An Empirical Study*. In Proceedings of the 20th IEEE International Conference on Program Comprehension (**ICPC**), Passau, Germany (**June 2012**).

5. Zohreh Sharafi, **Zéphyrin Soh**, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. *Women and Men: Different but Equal: On the Impact of Identifier Style on Source Code Reading*. In Proceedings of the 20th IEEE International Conference on Program Comprehension (**ICPC**), Passau, Germany (**June 2012**).

Symposiums and posters:

1. **Zéphyrin Soh**. *Context and Vision: Studying Two Factors Impacting Program Comprehension*. In Proceedings of the 19th IEEE International Conference on Program Comprehension (**ICPC**), Ph.D. Student Symposium, Kingston, Ontario, Canada.(**June 2011**).

2. **Zéphyrin Soh**. *On the Relation Between Code Smells and Maintenance Effort*. 3rd Poly-MORSE Open Workshop (**PLOW**), Montréal, Canada (**2014**).

RÉSUMÉ

Avec les besoins croissants des utilisateurs ainsi que l'évolution de la technologie, les programmes informatiques deviennent de plus en plus complexes. Ces programmes doivent évolués et être maintenus afin de corriger les bogues et–ou s'adapter aux nouveaux besoins. Lors de la maintenance des programmes, la compréhension des programmes est une activité indispensable et préliminaire à tout changement sur le programme. Environ 50% du temps de maintenance d'un programme est consacré à leur compréhension. Plusieurs facteurs affectent cette compréhension. L'étude de ces facteurs permet non seulement de les comprendre et d'identifier les problèmes que peuvent rencontrer les développeurs mais aussi et surtout de concevoir des outils permettant de faciliter leur travail et d'améliorer ainsi leur productivité.

Pour comprendre le programme et effectuer des tâches de maintenance, les développeurs doivent naviguer entre les éléments du programme (par exemple les fichiers) dans l'objectif de trouver le(s) élément(s) à modifier pour réaliser leurs tâches. L'exploration du programme est donc essentielle. Notre thèse est que l'effort de maintenance n'est pas seulement lié à la complexité de la tâche, mais aussi à la façon dont les développeurs explorent un programme.

Pour soutenir cette thèse, nous utilisons les données d'interaction qui contiennent les activités effectuées par les développeurs durant la maintenance. D'abord, nous étudions les bruits dans les données d'interaction et montrons qu'environ 6% du temps de réalisation d'une tâche ne sont pas contenus dans ces données et qu'elles contiennent environ 28% d'activités faussement considérées comme modifications effectuées sur le programme. Nous proposons une approche de nettoyage de ces bruits qui atteind une précision et un rappel jusqu'à 93% et 81%, respectivement. Ensuite, nous montrons que ces bruits impactent les résultats de certains travaux précédents et que le nettoyage des bruits améliore la précision et le rappel des systèmes de récommendation jusqu'à environ 51% et 57%, respectivement. Nous montrons également utilisant les données d'interaction que l'effort de maintenance n'est pas seulement lié à la complexité de la tâche de maintenance mais aussi à la façon dont les développeurs explorent les programmes. Ainsi, nous validons notre thèse et concluons que nous pouvons nettoyer des données d'interaction, les utiliser pour calculer l'effort de maintenance et comprendre l'impact de l'exploration du programme sur l'effort. Nous recommendons aux chercheurs de nettoyer les traces d'interaction et de les utiliser lorsqu'elles sont disponibles pour calculer l'effort de maintenance. Nous mettons à la disposition des fournisseurs d'outils de monitoring et de recommendation des résultats leur permettant d'améliorer leurs outils en vue d'aider les développeurs à plus de productivité.

ABSTRACT

Software systems must be maintained and evolved to fix bugs and adapted to new technologies and requirements. Maintaining and evolving systems require to understand the systems (*e.g.,* program comprehension) prior to any modification. Program comprehension consumes about half of developers' time during software maintenance. Many factors impact program comprehension (*e.g.,* developers, systems, tasks). The study of these factors aims (1) to identify and understand problems faced by developers during maintenance task and (2) to build tools to support developers and improve their productivity.

To evolve software, developers must always explore the program, *i.e.,* navigate through its entities. The purpose of this navigation is to find the subset of entities relevant to the task. Our thesis is that the maintenance effort is not only correlated to the complexity of the implementation of the task, but developers' exploration of a program also impacts the maintenance effort.

To validate our thesis, we consider interaction traces data, which are developers' activities logs collected during maintenance task. We study noise in interaction traces data and show that these data miss about 6% of the time spent to perform the task and contain on average about 28% of activities wrongly considered as modification of the source code. We propose an approach to clean interaction traces. Our approach achieves up to 93% precision and 81% recall. Using our cleaning approach, we show that some previous works that use interaction traces have been impacted by noise and we improve the recommendation of entities by up to 51% precision and 57% recall. We also show that the effort is not only correlated to the complexity of the implementation of the task but impacted by the developers' exploration strategies. Thus, we validate our thesis and conclude that we can clean interaction traces data and use them to assess developers' effort, understand how developers spend their effort, and assess the impact of their program exploration on their effort.

Researchers and practitioners should be aware of the noise in interaction traces. We recommend researchers to clean interaction traces prior to their usage and use them to compute the effort of maintenance tasks instead of estimating from bug report. We provide practitioners with an approach to improve the collection of interaction traces and the accuracy of recommendation systems. The use of our approach to improve these tools will likely improve the developers' productivity.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS AND ABBREVIATIONS

IDE	Integrated Development Environment
SCMS	Source Code Management System
SVN	Subversion
ITS	Issue Tracker System
IT	Interaction Trace
RIT	Raw Interaction Trace
CIT	Cleaned Interaction Trace
ES	Exploration Strategy
UE	Unreferenced Exploration
RE	Referenced Exploration
LOC	Line Of Code
UML	Unified Modeling Language

CHAPTER 1 INTRODUCTION

"... because of the lack of physical constraints, software systems can quickly become extremely complex, difficult to understand, and expensive to change."
Ian Sommerville

1.1 Context

Software engineering is the engineering discipline that study and provide the methods, techniques, and tools to solve various problems with software systems. Software systems are widely used in our daily life. They are maintained and evolved throughout their life (Lehman et Belady, 1985; Sommerville, 2011). Software systems are changed to fix bugs and adapt to new technologies and requirements. Thus, software maintenance and evolution is an essential activity in software engineering.

Maintaining and evolving software systems involve three main steps: understanding the existing software, modifying the existing software, and revalidating the modified software (Boehm, 1976b). Understanding the existing software (also known as program comprehension) is an important activity prior to any change (Singer *et al.*, 1997). Program comprehension consumes about half of developers' time during software maintenance (Fjeldstad et Hamlen, 1983). It is the activity of understanding how a software system (or part thereof) works (Maalej *et al.*, 2014b). It consists of exploring source code, finding, and understanding the subset of program entities relevant to the maintenance task (Ko *et al.*, 2006; Sillito *et al.*, 2008; Singer *et al.*, 1997).

Previous work studied how developers understand systems and perform maintenance and evolution tasks. Program comprehension theories report the processes taking place in the software developers' minds (Storey, 2006). They study how developers use prior knowledge about programming, and information present in the system to form their mental model. However, these program comprehension theories are difficult to assess in practice. In complement to these theories and because the cognitive processes taking place in developers' minds are triggered by visual attention (Duchowski, 2007), researchers have used eye-tracking technology to study program comprehension (Sharafi *et al.*, 2015). The use of eye-trackers is limited for real-world development environment. Eye-trackers are usually used in experimental environment with a stimulus that is an image of a snippet code or other artifact under study. Yet, developers usually used modern IDE to maintain systems containing many source code files.

To overcome the limitations of the theories above and the use of eye-tracking, other previous works conducted explanatory studies to assess the external behaviors of developers and understand how they explore the program and search for relevant entities. These other works used several techniques to collect data in order to assess how developers understand program: the video capture of the screen, the think-aloud and protocol analysis. It is difficult to automatically analyse data collected through these techniques. Moreover, these data are generally collected in experimental settings and may not reflect developers' real working conditions.

On the contrary to the use of the techniques stated above (*i.e.,* video capture, think-aloud, protocol analysis) the interaction traces data are collected when developers perform task in a real-world development environment. These data are collected by monitoring tools installed in the developers' IDE. The interaction traces data contains the interaction of developers with the program entities through the IDE. Thus, they contain fined-grained information about developers' activities. Therefore, we think that interaction traces are valuable source of information to study how developers understand program and spend their effort.

1.2 Problem and Thesis Statement

On the one hand, developers spend a certain effort exploring the program, finding relevant entities, understanding and making changes to the program. Previous explanatory studies have shown that program exploration is time consuming (Ko *et al.*, 2006; Robillard *et al.*, 2004) and developers usually navigate between and within related program entities (Ko *et al.*, 2006).

On the other hand, interaction traces have been used to study and link developers' editing patterns to the type of maintenance task (Ying et Robillard, 2011), to correlate the effort spend by the developers during maintenance task with the developers' experience (Robbes et Röthlisberger, 2013), to reduce developers' information overhead and evaluate developers' productivity (Kersten et Murphy, 2006), to relate the work fragmentation to developers' productivity (Sanchez *et al.*, 2015), to predict changes (Bantelay *et al.*, 2013) and their impact (Zanjani *et al.*, 2014), and to improve tools for code completion (Robbes et Lanza, 2010) and recommendations of program entities (Kersten et Murphy, 2005; Lee et Kang, 2013; Lee *et al.*, 2015). Most of these studies depend on the accuracy of the information mined from interaction traces data. To the best of our knowledge, there is no work that aim to validate the quality of interaction traces.

Our thesis in this dissertation is:

We can collect and use accurate interaction traces to assess developers' effort, understand how developers' spent their effort, and assess their program exploration strategies during maintenance and evolution activities.

1.3 Contributions

To answer our thesis, we perform the following research works and make the following contributions.

1.3.1 Investigate the Quality of Interaction Traces Data

Motivations

Interaction traces (ITs) are developers' logs collected when developers maintain or evolve software systems. ITs are used in research and practice to study developers' behaviour (Ying et Robillard, 2011; Zhang *et al.*, 2012), their productivity (Kersten et Murphy, 2006; Sanchez *et al.*, 2015), and recommend relevant program entities (Lee *et al.*, 2015). However, the use of ITs is based on several assumptions, which may or may not be true. Therefore, our first research work concerns the investigation of the quality of interaction traces data.

Goal and Relevance

We assess to which extent the assumptions made when using ITs are correct and study whether ITs contain noise due to incorrect assumptions. Noise could let us to make the wrong conclusions on developers' efforts and strategies when validating our thesis. Knowing if ITs contain noise may also show that previous analyses of ITs are biased and prevent both researchers and practitioners to assist developers effectively. Providing an approach to clean noise in ITs may increase the capability of the software engineering community to better assist software development.

Approach

To study noise in ITs, we conduct a controlled experiment collecting both Mylyn ITs and video-screen captures with 15 participants who performed four bug-fixing activities. We assess the noise in ITs by comparing Mylyn ITs and the ITs obtained from the video captures.

Results

We observed that Mylyn ITs can miss on average about 6% of the time spent by participants performing tasks and contain on average about 28% of events labelled as edit events that are not real modification of source code (*i.e.,* false edit-events). We then propose two approaches to clean noise in ITs. The first approach uses the time spent on edit activity to automatically classify an edit activity as the change performed on source code. This approach achieves from 18% to 93% precision and form 64% to 81% recall. The second approach uses the machine learning classification to predict the changes performed on source code and achieves form 31% to 98% precision and from 77% to 90% recall. According to our thesis, we argue that **we can collect and use ITs to assess developers' effort (*i.e.,* time spent by developers).**

Our results are published in the following conference:

- **Zéphyrin Soh**, Thomas Drioul, Pierre-Antoine Rappe, Foutse Khomh, Yann-Gaël Guéhéneuc and Naji Habra. *Noise in Interaction Traces Data and their Impact on Previous Research Studies*. In Proceedings of the 9th ACM/IEEE International Symposium of Empirical Software Engineering and Measurement (**ESEM**), Beijing, China (**October 2015**).

The extension of this work is under review in the following journal:

- **Zéphyrin Soh**, Foutse Khomh, Yann-Gaël Guéhéneuc and Giuliano Antoniol. *Noise in Interaction Traces Data and its Impact on Developers' Behaviour Studies and on the Accuracy of Recommendation Systems*. Empirical Software Engineering Journal, Springer (**EMSE, 2015**). (**under review**)

Implication

The implication of the presence of noise in ITs is that we cannot accept some of the assumptions made when using ITs, and that previous studies would have not made these assumptions. Thus, we first revisit previous studies to assess how noise impact their findings.

1.3.2 Investigate the Impact of Noise on Previous Studies

Motivations

The noise observed in ITs indicate that previous works that made assumptions on ITs are likely to have some inaccuracies. These noise are important knowledge and may receive the attention of researchers and practitioners only if they impact the previous works *i.e.,* the uses of ITs with and without noise provide different results.

Goal and Relevance

We aim to investigate the impact of noise on previous works. The absence of the impact of noise on previous works may indicate that cleaning noise in ITs is an extra processing step that do not provide any benefit. However, if noise impact previous works, it shows that noise must be cleaned prior to their usage.

Approach

We select two previous works representative of two families of studies: (1) studies that used ITs to understand developers' behaviours during development and maintenance activities and (2) studies that leveraged IT data to recommend program entities to developers performing development and maintenance tasks. We choose these two categories of studies for two reasons. First, they are studies that investigate the aspects pertaining to the assumptions made on ITs. Second, we have access to the data and–or the scripts/source code used in these studies and we can use them for fair comparison of their results using RITs and CITs. We use the work of (Ying et Robillard, 2011) on developers' editing styles and the work of (Lee *et al.*, 2015) on recommendation systems. We revisit these work using ITs with and without noise and compare the obtained results.

Results

Cleaning noise reveals 41% of misclassification of ITs editing styles. We also show that cleaning noise in ITs can improve the precision and recall of recommendation systems by up to 51% and 57%, respectively. Therefore, we conclude that **noise impact previous studies and researchers must clean noise in ITs prior to their usage.**

Our results are under review together with the previous contribution (Section 1.3.1) in the following journal:

- **Zéphyrin Soh**, Foutse Khomh, Yann-Gaël Guéhéneuc and Giuliano Antoniol. *Noise in Interaction Traces Data and its Impact on Developers' Behaviour Studies and on the Accuracy of Recommendation Systems*. Empirical Software Engineering Journal, Springer (**EMSE**, **2015**). (**under review**)

Implication

The impact of noise in ITs on previous studies show that noise would have impacted the validation of our thesis if we made the same assumptions as previous works. Thus, to realise our next contributions, we use both raw ITs (RITs in the remaining) and cleaned ITs (CITs in the remaining) to assess the sensibility of our findings to the accuracy of the ITs data.

1.3.3 Understand How Developers Spend their Effort

Motivations

Many works aim at estimating the effort needed from developers to fix a bug (Nadeem Ahsan *et al.*, 2009; Giger *et al.*, 2010; Hewett et Kijsanayothin, 2009; Song *et al.*, 2006; Weiss *et al.*, 2007). It generally consists in building models using both bug report and source code metrics. However, these models measure the effort to fix a bug without considering how developers spent their effort. Yet, developers spend an amount of effort trying to understand source code prior to making changes (Singer *et al.*, 1997), and some metrics (*e.g.,* LOC) used to predict the effort under-estimate the amount of effort required (Shihab *et al.*, 2013).

Goal and Relevance

We aim to understanding how developers spend their effort. By knowing how developers' spend their effort, software organisations could take the necessary steps to improve the efficiency of their developers, for example, by providing them with adequate development tools.

Approach

We mine ITs and patches to study how developers spent their effort. We consider ITs collected by developers while performing a task. We also collect patches that developers provide as the result of the task resolution. The patches include the files involving the lines of code added, deleted, and modified. Then, we first match ITs to patches to define the patch that is the result of an IT. Secondly, we analyse whether developers spent their effort building complex implementations of the task and–or if they spent effort using program entities that are not changed during the implementation solution of the task (*i.e.,* additional files). We analyse both RITs and CITs to assess the sensibility of our results to the noise in ITs.

Results

We found that the effort spent by developers when performing a task is not strongly correlated to the complexity of the implementation of the task. They use on average about 62% of additional files and, the more developers explore additional files, the more they spent effort. Our conclusion is that **we can use interaction traces to understand how developers spend their effort during maintenance and evolution tasks**.

Our results are published in the following conference:

- **Zéphyrin Soh**, Foutse Khomh, Yann-Gaël Guéhéneuc and Giuliano Antoniol; *Towards Understanding How Developers Spend Their Effort During Maintenance Activities*. In Proceedings of the 20th IEEE Working Conference on Reverse Engineering (**WCRE**), Koblenz, Germany (**October 2013**).

Implication

Because developers' effort is not correlated to the complexity of the implementation and their effort is also spent on additional files, we conclude that a part of the developers' effort is spent during their exploration of the program. This conclusion calls for an assessment of developers' program exploration strategies and their impact on the effort.

1.3.4 Assess the Impact of Exploration Strategies on Effort

Motivations

In addition to our conclusion about how developers spend their effort, others reported in previous works suggested the need of understanding how developers explore program and studying how program exploration strategies impact the developers' effort. In fact, Robillard *et al.* (2004) reported an exploratory study on how developers investigate source code. They observed that *methodical investigation* of source code (by successful subjects) is more effective than *opportunistic approach* (by unsuccessful subjects). However, they designed the study by separating the program investigation phase from the program change phase, while developers usually interleaves these activities (Ko *et al.*, 2006). Ko *et al.* (2006) conducted an exploratory study to investigate the strategies followed by the developers to perform maintenance task. They observed that developers interleaved three activities and they model the program understanding as the process of *searching*, *relating*, and *collecting* relevant information. Their study reveals that developers spent a considerable amount of time navigating within and between source files.

Goal and Relevance

We investigate the exploration strategies (ES) followed by developers during maintenance tasks and assess the impact of these ES on the effort spent by developers on the tasks. We assess whether developers frequently revisit one (or a set) of program entities (referenced exploration — RE) or visit program entities with almost the same frequency (unreferenced exploration — UE) when performing a maintenance task. The study of exploration strategies can help (1) to improve our knowledge on developers' comprehension process, (2) characterise developers' expertise, *e.g.,* how experienced developers explore a program can differ from the way inexperienced ones explore a program and how the strategy of experienced developers can be used to help inexperienced ones; and (3) to find techniques and tools to reduce the developers' search effort and guide them when exploring a program.

Approach

We conduct a user study to assess whether both referenced and unreferenced explorations occur in practice. The user study involved nine participants to whom we asked to manually classify a sample of ITs as RE or UE. The manual classification shows the moderate agreement between subjects when distinguishing RE and UE. We then define an approach to automatically classify an IT as referenced or unreferenced exploration. We finally compare the effort spent when following UE with the effort spent when following RE. We analyse both RITs and CITs to assess the sensitivity of our results to the noise in ITs.

Results

We observed that developers mostly follow UE and that UE is on average 12.30% less effort consuming than RE. Our results show that **we can use interaction traces to assess developers' exploration strategies and study how these strategies impact developers' effort.**

Our results are published in the following conference:

- **Zéphyrin Soh**, Foutse Khomh, Yann-Gaël Guéhéneuc, Giuliano Antoniol and Bram Adams; *On the Effect of Exploration Focus on Maintenance Tasks*. In Proceedings of the 20th IEEE Working Conference on Reverse Engineering (**WCRE**), Koblenz, Germany (**October 2013**).

Implication

Our results shows the need for characterizing exploration strategies in the future. In particular, we plan to analyse the relations between exploration strategies and developers' expertise to confirm or not the intuition that novice and expert developers may tend to follow different exploration strategies. Toward this investigation, our pilot study using eye-tracker to assess the understanding of UML class diagram shows that experts are more accurate than novices while novices spend around 33% less time than experts.

Our results on the pilot study are published in the following conference:

- **Zéphyrin Soh**, Zohreh Sharafi, Bertrand Van Den Plas, Gerardo Cepeda, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. *Professional Status and Expertise for UML Class Diagram Comprehension: An Empirical Study.* In Proceedings of the 20th IEEE International Conference on Program Comprehension (**ICPC**), Passau, Germany (**June 2012**).

Table 1.1 summarise our contributions and introduce the organisation of the dissertation. In particular, while our findings support our thesis, *i.e.,* **we can collect and use accurate interaction traces to assess developers' effort, understand how developers' spent their effort, and assess their program exploration strategies during maintenance and evolution activities**, we report how our contributions could help to improve the state of the practice of software development in both research and practical perspectives.

1.4 Organization

Our dissertation is organized as follow:

Chapter 2 provides the background concepts necessary to follow the dissertation. We first describe the developers working environment followed by the detailed information on interaction traces and the use of interaction traces.

Chapter 3 presents previous work related to our dissertation. We summarise works that address program comprehension, developers' behaviour and the use of interaction traces data.

Chapter 4 aims to study noise in interaction traces data. We first describe the experiment conducted to investigate if there is noise in interaction traces data. Then we propose two approaches to clean noise in interaction traces data.

Chapter 5 reports the study of the impact of noise on previous studies. We describe the two previous works that we revisited and show how much noise impact the identification of developers' editing styles and improve the accuracy of recommendation systems.

Chapter 6 relates developers' effort to the complexity of the implementation of the task and assess the factors that contribute to the effort spent during maintenance task.

Chapter 7 investigates the impact of program exploration on maintenance tasks. We conduct a user study of define two exploration strategies. Then, we automatically classify interaction traces as referenced or unreferenced and study how developers spent time following different exploration strategies.

Chapter 8 concludes the dissertation and outlines future works.

Table 1.1 Summary of the Contributions

Publications	**Sections**	**Contributions**		
		Contributions	**Researchers**	**Practitioners**
ESEM 2015	Chapter 4	We propose a methodology to validate the assumptions made by researchers on ITs data.	Do not make these assumptions anymore, at least for Mylyn ITs.	The opinion from the industry confirms that noise is prevalent in ITs.
		We propose the approaches to clean ITs data.	Researchers must use our approaches to clean ITs data prior to their usage.	The industry express the need for cleaning ITs data during the collection of ITs.
EMSE 2015	Chapter 5	We show that noise in ITs bias the results reported in previous works.		
WCRE 2013	Chapter 6	We improve the knowledge on how developers spent their effort.	Do not use the complexity of the patches as a proxy to the developers' effort.	Tools (*e.g.,* Jira) needs improvement to avoid self reported effort.
			Need to characterise entities that are not changed to resolve the tasks, but that are useful to understand entities to change. Need new feature location approaches that consider additional useful entities and need appropriate methods to validate feature location approaches.	Need feature location and recommendation tools that may suggest both entities to change and those needed to understand the entities to change.
WCRE 2013	Chapter 7	We improve the knowledge on developers' program comprehension process.	Need to characterise developers' exploration strategies and identify the ”good“ strategy that may help to guide developers during program exploration.	Need tools to reduce the developers' search effort and guide them when exploring a program.

CHAPTER 2 BACKGROUND

2.1 Developers' Working Environments

Our thesis focus on the developers' working environments where the system to maintain and evolve is hosted in a source code management system such as Subversion[1] (SVN) or Git[2]. Figure 2.1 summarise the developers' working environment.

Workspace: To maintain and evolve software systems, developers usually use an Integrated Development Environment (IDE) such as Eclipse[3]. Developers must have the system in their workspace (*e.g.,* by cloning the system from the source code management system).

ITS (Issue Tracker System): The change requests that aim to improve or correct the system during the life cycle of the system are reported in the ITS such as Bugzilla[4]. For the sake of simplicity, we consider the change requests reported in the ITS as the maintenance tasks (or simply tasks) as they may be the correction, adaptation, or evolution tasks.

SCMS (Source Code Management System): Once the developers have the system in their workspace, they can work on a task reported in the ITS. After resolving the task, developers provide the solution as a patch or they can commit into the SCMS if they have the commiter right.

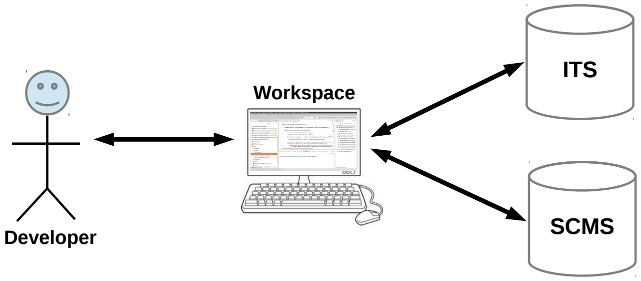


Figure 2.1 The Developers' Working Ecosystem

2.2 Interaction Traces

When resolving a task, developers must find where and how to change relevant program entities (Ko *et al.*, 2006; Sillito *et al.*, 2008). They usually interact with program entities through the IDE

[1] https://subversion.apache.org/ (visited 20/10/2015)
[2] https://git-scm.com/ (visited 20/10/2015)
[3] http://www.eclipse.org/ (visited 20/10/2015)
[4] https://www.bugzilla.org/ (visited 20/10/2015)

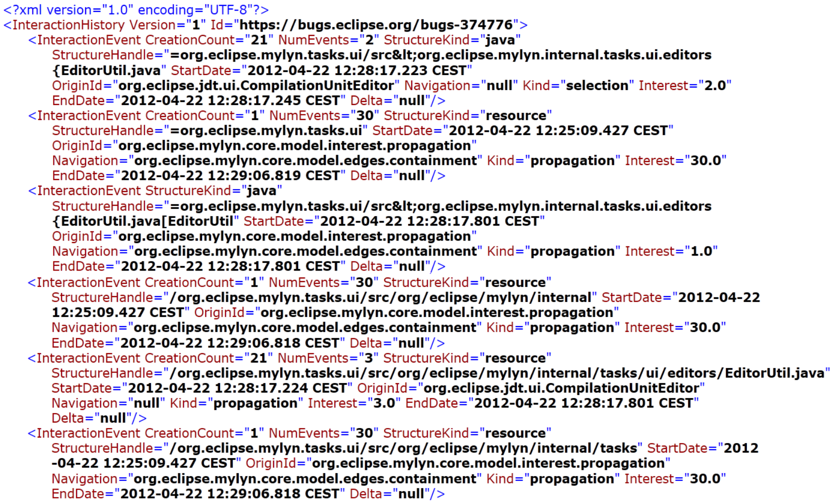
and perform several kind of activities, such as opening a file, navigating static relations, searching program entities, changing a code. The study of the developers' activities reveals the need for tools and methods to assist developers when searching information, to reduce developers information overhead (Kersten et Murphy, 2006), and to collect relevant information (Ko *et al.*, 2006). These developers' activities are observed from the video capture of the developers' screen or through a monitoring tool that collect developers' activities.

2.2.1 Monitoring Tools

Developers' activities are usually collected by monitoring tools, such as Mimec (Layman *et al.*, 2008) or Mylyn (Kersten et Murphy, 2005). Mylyn is an Eclipse plugin that captures developers' interactions with program entities when performing a task. Figure 2.2 shows an example of Mylyn interaction trace. Each developers' action on a program entity is recorded as an *interaction event* (or simply an event). There are eight types of *events* in Mylyn[5]: *Attention*, *Command*, *Edit*, *Manipulation*, *Prediction*, *Preference*, *Propagation*, and *Selection*. The list of interaction events form an interaction trace. An interaction trace is therefore a sequence of interaction events that describe accesses and operations performed on program entities (Kersten et Murphy, 2006). Interaction traces logs are stored in an XML format. Each interaction traces log is identified by a unique *ID* and contains the descriptions of events (*i.e., InteractionEvent*) recorded by Mylyn. The description of each event includes: a starting date (*i.e., StartDate*), an end date (*i.e., EndDate*), a type (*i.e., Kind*), the identifier of the UI component that tracks the event (*i.e., OriginId*), and the program entity involved in the event (*i.e., StructureHandle*). Mylyn also records events that are not directly triggered by developers. However, we consider only interaction events explicitly triggered by developers: *Selection*, *Edit*, *Command*, and *Preference*.

Developers' interaction traces (ITs) have been used to study developers' preferred views in an IDE (Murphy *et al.*, 2006), and edit styles (Ying et Robillard, 2011; Zhang *et al.*, 2012) as well as work fragmentation (Sanchez *et al.*, 2015), maintenance effort (Robbes et Röthlisberger, 2013) and to predict changes (Bantelay *et al.*, 2013), and their impact (Zanjani *et al.*, 2014). Developers' ITs have also been used to provide tool support through code completion (Robbes et Lanza, 2010) and recommendations of program entities (Kersten et Murphy, 2005; Lee et Kang, 2013; Lee *et al.*, 2015). Therefore, developers' ITs are valuable source of information to understand developers' behaviour and improve their productivity.

[5] http://wiki.eclipse.org/Mylyn_Integrator_Reference (visited 20/10/2015)



```
<?xml version="1.0" encoding="UTF-8"?>
<InteractionHistory Version="1" Id="https://bugs.eclipse.org/bugs-374776">
    <InteractionEvent CreationCount="21" NumEvents="2" StructureKind="java"
        StructureHandle="=org.eclipse.mylyn.tasks.ui/src&lt;org.eclipse.mylyn.internal.tasks.ui.editors
        {EditorUtil.java" StartDate="2012-04-22 12:28:17.223 CEST"
        OriginId="org.eclipse.jdt.ui.CompilationUnitEditor" Navigation="null" Kind="selection" Interest="2.0"
        EndDate="2012-04-22 12:28:17.245 CEST" Delta="null"/>
    <InteractionEvent CreationCount="1" NumEvents="30" StructureKind="resource"
        StructureHandle="=org.eclipse.mylyn.tasks.ui" StartDate="2012-04-22 12:25:09.427 CEST"
        OriginId="org.eclipse.mylyn.core.model.interest.propagation"
        Navigation="org.eclipse.mylyn.core.model.edges.containment" Kind="propagation" Interest="30.0"
        EndDate="2012-04-22 12:29:06.819 CEST" Delta="null"/>
    <InteractionEvent StructureKind="java"
        StructureHandle="=org.eclipse.mylyn.tasks.ui/src&lt;org.eclipse.mylyn.internal.tasks.ui.editors
        {EditorUtil.java[EditorUtil" StartDate="2012-04-22 12:28:17.801 CEST"
        OriginId="org.eclipse.mylyn.core.model.interest.propagation"
        Navigation="org.eclipse.mylyn.core.model.edges.containment" Kind="propagation" Interest="1.0"
        EndDate="2012-04-22 12:28:17.801 CEST" Delta="null"/>
    <InteractionEvent CreationCount="1" NumEvents="30" StructureKind="resource"
        StructureHandle="/org.eclipse.mylyn.tasks.ui/src/org/eclipse/mylyn/internal" StartDate="2012-04-22
        12:25:09.427 CEST" OriginId="org.eclipse.mylyn.core.model.interest.propagation"
        Navigation="org.eclipse.mylyn.core.model.edges.containment" Kind="propagation" Interest="30.0"
        EndDate="2012-04-22 12:29:06.818 CEST" Delta="null"/>
    <InteractionEvent CreationCount="21" NumEvents="3" StructureKind="resource"
        StructureHandle="/org.eclipse.mylyn.tasks.ui/src/org/eclipse/mylyn/internal/tasks/ui/editors/EditorUtil.java"
        StartDate="2012-04-22 12:28:17.224 CEST" OriginId="org.eclipse.jdt.ui.CompilationUnitEditor"
        Navigation="null" Kind="propagation" Interest="3.0" EndDate="2012-04-22 12:28:17.801 CEST"
        Delta="null"/>
    <InteractionEvent CreationCount="1" NumEvents="30" StructureKind="resource"
        StructureHandle="/org.eclipse.mylyn.tasks.ui/src/org/eclipse/mylyn/internal/tasks" StartDate="2012
        -04-22 12:25:09.427 CEST" OriginId="org.eclipse.mylyn.core.model.interest.propagation"
        Navigation="org.eclipse.mylyn.core.model.edges.containment" Kind="propagation" Interest="30.0"
        EndDate="2012-04-22 12:29:06.818 CEST" Delta="null"/>
```

Figure 2.2 An Example of Mylyn Interaction Trace

2.2.2 Collecting and Processing Interaction Traces

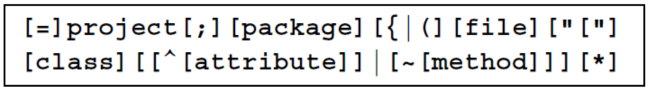
We use the interactions traces collected by the Eclipse Mylyn plug-in. Interaction traces are associated to a task as attachments. After performing a task, developers attach their traces to the appropriate task in the ITS. An attachment is identified by the tag “attachment” in the XML bug report file. Eclipse Bugzilla is an example of ITS where developers attach interaction traces.

To obtain interactions traces, we first download the bug reports and parse them to extract the interactions traces IDs. Then, we download the bug reports’ attachments with the name “mylyn-context.zip", which is the default name given by Mylyn to interaction traces.

We download the interaction traces using their IDs and we parse them to extract useful information such as the program entities explored/used by a developer, the actions performed and the time spend on these entities. A program entity can be a resource (XML, MANIFEST.MF, properties, HTML files, etc.) or a Java program entity (*i.e.,* project, package, file, class, attribute, or method). A Java *StructureHandle* is structured in multiple parts. We use a regular expression to identify all its parts. Regular expressions were already used by Bettenburg *et al.* (2008) to identify parts of stack traces contained in bug reports. Figure 2.3 shows the structure of a Java *StructureHandle*.

2.3 Patches and Relevance of Program Entities

After performing a task, developers commit their changes in the SCMS or provide them as a patch in a code review system (*e.g.,* Gerrit) or an ITS. The changes provided as a commit or patch can be used to know how a task was finally addressed *i.e.,* the entities used and the modifications



```
[=]project[;][package][{|(][file]["["]
[class][[^[attribute]]|[~[method]]][*]
```

Figure 2.3 The structure of a Java *StructureHandle*

performed on these entities.

We consider the results submitted as a patch. A patch is associated to a bug report as an attachment. To collect patches, we download bug reports' attachments with the attribute "ispatch" (of the tag "attachment") equal to one, that identify the patch.

According to Lee et Kang (2011), a significantly relevant entity is *a program entity that a developer needs to change in order to accomplish the task.* Therefore, the entities in the patch are *significantly relevant entities* because they are changed to perform a task. The entities in the interaction traces are those that are explored, but not changed. We call them *additional entities* because these entities can be important for the understanding of the program and the completion of the task (*i.e., useful files*). These *additional entities* can also be just *accidental entities i.e.,* developers accidentally explored these entities when they were looking for significantly relevant and–or useful ones.

CHAPTER 3 LITERATURE REVIEW

Our thesis relies on two main families of work that are reported in this chapter: (1) Understanding program comprehension and (2) Mining interaction traces data. They are related to our thesis in the sense that we aim to use interaction traces data to understand how developers' explore programs and spend their effort.

3.1 Understanding Program Comprehension

The first step toward helping developers during maintenance and evolution tasks has been to understand how they comprehend programs. Two directions emerged: the study of the cognitive process taking place in the developers' minds (*i.e.,* theories of program comprehension) and the investigation of developers' behavior during program exploration (*i.e.,* observable activities performed by the developers).

3.1.1 Theories of Program Comprehension

Cognitive theories of program comprehension define how software engineers understand a program. These theories explain the process that take place in developers' minds and the knowledge that they use to perform a given task. There are four existing cognitive theories proposed for program comprehension.

The top-down model (Brooks, 1983) defines a cognitive process as reconstruction of domain knowledge and their mapping into source code, through several intermediate domains. In the top-down process, a software engineer forms a primary hypothesis about a program. Then, she uses the domain knowledge to refine that hypothesis until a specific hypothesis. Finally, she maps the refined hypothesis to source code by verifying the absence or the presence of corresponding beacons. A beacon is a recognizable, familiar feature in the code that acts as an indicator of the presence of particular structure or operation.

The bottom-up model (Pennington, 1987; Shneiderman et Mayer, 1979) assumes that the code statements are used to form a program model (control-flow abstraction), which is used to build the situation model by using domain concepts. A software engineer reads the code statements. She mentally groups these statements into higher level abstractions. Then, she uses domain concept to understand grouped statements.

The opportunistic/systematic model (Littman *et al.*, 1986) considers comprehension as a process in which any available cues can be exploited. A software engineer reads the code in details. She traces through the control-flow and data-flow abstraction. She matches code to her knowledge base for global understanding of the system. The as-needed approach is used for specific understanding by focusing only on the code related to a given task. Letovsky (1986) complements Littman *et al.* model (Littman *et al.*, 1986) and defines the comprehension as a knowledge-based process containing the knowledge base, the mental model, and the assimilation process. The comprehension process resides on inquiries concept, that consists of asking the question, conjecturing an answer, and searching through the code and documentation to verify the answer (Storey, 2006).

The integrated model (von Mayrhauser et Vans, 1995) relates top-down, bottom-up, and opportunistic approaches to define several levels of abstraction and how to switch between the three previous models. The mental representation of the program is build regarding the domain, program, and situation models.

These program comprehension theories are difficult to assess in practice because they focused on the process in the developers' minds. Thus, the externalized behavior of developers when comprehending programs have been studied.

3.1.2 Program Exploration

Wang *et al.* (2013) identified *actions*, *phases* and *patterns* occurred when developers perform feature location tasks. Wang *et al.* observed that feature location process involve *search*, *extend*, *validation*, and *document* phases. Developers search for entrance points, extends the relevant program element, validate the relevance of program element and document analysis results. In addition to feature location phases, Wang *et al.* identified *information retrieval-based*, *execution-based* and *exploration-based* search patterns that developers follow when performing the search of entrance point. They also observed that developers follow *execution-based* and *exploration-based* extension patterns when extending entrance point. Wang *et al.* also studied the effectiveness of identified phases/patterns and observed that the knowledge of these phases/patterns help developers in the completion of the task. The work by Wang *et al.* is limited to the feature location activities which are only part of maintenance task. Moreover, they performed an exploratory study involving only six feature location tasks.

Robillard *et al.* (2004) used maintenance tasks and assessed the accuracy of the task resolution. They reported that *methodical investigation* of source code (by successful subjects) is more effective than *opportunistic approach* (by unsuccessful subjects). Successful subjects seem to answer specific questions using focused search (structural guided searches *e.g.,* keywords and cross-reference searches) and create a detailed and complete plan prior to the change while unsuccessful subjects exhibit more guessing using broad searches without a specific discovery purpose (search based on intuition, scrolling/browsing) and did not have a plan prior to the change. Moreover, successful subjects do not reinvestigate source code as frequently as unsuccessful subjects, and unsuccessful subjects seem to have their code modification in one place. Robillard *et al.* focused on comparing successful and unsuccessful developers through a controlled explanatory study. Their study were performed in the lab setting with only five developers performing identical task on one system, and they devided the maintenance task into two distinct phases (*i.e.,* program investigation and program change phases).

Regardless the ability of developers to solve the task and the constraints imposed by Robillard *et al*. (2004), Ko *et al*. (2006) described developers' behavior as a process of *searching*, *relating*, and *collecting* relevant information. Developers usually explore program to find relevant elements. They understand relevant element by relating to other elements and when they found an element relevant, they collect in order to implement solution. The model described by Ko *et al.* revealed the need of (1) environment to provide cues that may help developers to judge the relevance of information and (2) way to collect relevant information. While Ko *et al.* provided a more general model compared to Robillard *et al.*, they provided an unfamiliar program to the developers, without documentation nor comments in the program. They also manually simulated the interruptions which move the experiment from the real world development environment.

Through user studies, Sillito *et al*. (2008) complemented Ko *et al*. (2006)'s study by identifying four categories of questions developers ask during maintenance task and report actions undertaken by developers to answer these questions. When *finding a focus point* (*i.e.,* starting point to find relevant elements), developers may used text-based search or open type tool and debugger to answer relevance questions. After finding a focus point, developers *expand the focus point* (*i.e.,* look at other elements that they believe to be relevant to the task) by often exploring relationships (*i.e.,* reference searches). The *understanding of a subgraph* (of program element) is made using multiple windows/monitors. *Questions over groups of subgraphs* are high level questions that developers answer by answering related low level questions.

While searching for the relevant information is one of the activity reported in these studies, Lawrance *et al*. (2013) showed that Information Foraging Theory (IFT) can be used to assess developers' behavior when searching relevant information. They applied the IFT on debugging tasks and use verbal protocol (think-aloud) and screen captures to collect data about developers' behavior.

Instead of conducting an exploratory study to assess how developers comprehend program, Maalej *et al*. (2014b) observed and surveyed professional developers to understand their comprehension strategies and the tools used during program comprehension. They reported that developers follow a recurring, structured comprehension strategy depending on context (*i.e.,* type of task, developer personality, knowledge on the system, and the type of system).

These previous work on developers behaviors are all performed in the lab setting and involved few developers from one organisation. They mostly analysed video capture and use think-aloud or protocol analysis to examine how developers' work. Except Robillard *et al.* (2004) who focused on comparing the behavior of successful and unsuccessful developers, the others work are consistent on the fact that developers explore programs to search were to change by finding the starting point, extending it and collecting necessary information to perform the appropriate changes. More specially developers spend lot of time going back and forth to the related code (Ko *et al.*, 2006).

Instead of conducting exploratory studies using video capture, think-aloud, and protocol analysis, other works used eye-tracking technology to understand how developers perform tasks. However, due the the limitations of the eye-tracking technology that do not allow to observe developers in the real-world development environment, we think that this branch of research is not strongly related to our thesis and we do not report any literature review on the eye-tracking studies.

3.2 Mining Interaction Traces Data

Interaction traces provide a great opportunity to assess how developers perform tasks. Moreover, they have the advantage to be processed automatically compared to explanatory study using videos capture or protocol analysis. ITs have been used to study developers' behaviors, their productivity, and other software engineering activities.

3.2.1 Developers' Behaviors

Murphy *et al.* (2006) analysed how developers use the Eclipse IDE, *e.g.,* which Eclipse views and perspectives they mostly use. They mined ITs collected from 41 programmers and observed that developers differently used views available in Eclipse, for example, none used the declaration view. However, the view and perspective usage do not provide insight on how developers perform changes on programs.

Instead of analysing how developers use the IDE, Ying et Robillard (2011) studied developers' editing styles and characterised edit-first, edit-last, and edit-throughout editing styles. They observed that enhancement tasks are associated with a high fraction of edit events at the beginning of the programming session (*i.e.,* edit-first). They studied the editing style of individual developers'.

Zhang *et al.* (2012) studied how several developers concurrently edit a file and derived concurrent, parallel, extended, and interrupted file editing patterns. They stated that file editing patterns impact software quality because they are related to future bugs.

Other works used PHARO IDE for Smalltalk and the interaction profiler named DFlow to collect developers interaction and study how developers perform maintenance tasks (Minelli *et al.*, 2014, 2015). They quantified program comprehension and report that the vast majority of the time is spent by developers to read and understand source code (Minelli *et al.*, 2014). They show that program comprehension cost about 70% of maintenance time that has been underestimated by previous studies (Minelli *et al.*, 2015).

These work on developers' behaviors using ITs show how developers use the IDE and perform changes. They do not inform on how developers explore program and navigate through program entities. Yet, how developers explore program may impact their effort and productivity.

3.2.2 Developers' Productivity

Many approaches and tools have been proposed to assist developers when performing maintenance and evolution tasks. To address the problem of information overload *i.e.,* when developers interact with a large information spaces, Kersten et Murphy (2005) used developers' frequency and recency interaction with an entity to propose a degree-of-interest (DOI) model. The model is used to build the task context (Kersten et Murphy, 2006) and reduce the developers' information space. However, developers still have to look into the reduced information space to find relevant program entities.

Recommendation systems have been introduced to provide developers with information deemed relevant for a software engineering task in a given context (Robillard *et al.*, 2010). Different approaches have been proposed to develop recommendation systems, including the use of ITs (Maalej *et al.*, 2014a). (DeLine *et al.*, 2005) proposed TeamTrack that uses the association between previously-visited entities to recommend the next program entity to visit. NavTrack (Singer *et al.*, 2005) used only selection- and open-actions on files to discover hidden dependencies between files and recommend files related to the file of interest. NavClus (Lee et Kang, 2013) improved over previous approaches by clustering navigational cycles. MI (Lee *et al.*, 2015) included view/selection histories of program entities in association rules and recommend entities that are not yet edited.

Robbes et Lanza (2010) also used change history to propose a code completion tool that reduces developers' scrolling effort and that users found more accurate than the previous tools they used.

3.2.3 Others Development Activities

Task Interruption

While developers can work on many tasks at a time, Coman et Sillitti (2008) used the degree of access (*i.e.,* the amount of time a method is accesses) and the time interval a method is intensively access to automatically infer task boundaries and split developers' sessions.

As developers sometimes interrupt their work (Parnin et Rugaber, 2011), Parnin et Görg (2006) counted the number of prior consecutive interactions on an entity to extract the usage context of the task when it has been interrupted.

Sanchez *et al.* (2015) studied work fragmentation (through interruption) and related it to developers' productivity. They found that work fragmentation is correlated to low productivity.

Change Impact and Prediction

Bantelay *et al.* (2013) combined ITs with commit data to calculate evolutionary couplings. They reported that the combined ITs and commit information could improve the recall of prediction models by up to 13%, with a drop in precision of less than 2%. Similarly, Zanjani *et al.* (2014) improved an existing approach of impact analysis using ITs data.

Coupling, Collaboration and Effort

Zou *et al.* (2007) used the number of transitions between files to study the impact of interaction couplings on maintenance activities. They conclude that restructuring activities are more costly than other maintenance activities.

Schneider *et al.* (2004) investigated the benefits of tracking developers' local interactions trace when developing in a distributed environment.

Fritz *et al.* (2007) found that the DOI indicates the developers' knowledge about the structure of the code. Fritz *et al.* (2010) combined developers' interactions and authorship information (from change history) to model source code familiarity, *i.e.,* the degree of knowledge (DOK).

Robbes et Röthlisberger (2013) also used developers' interactions to assess developers' effort and analysed how developers' expertise is correlated to the effort on a task. They found a negative correlation between the time spent on a task and the experience of the developer performing the task.

3.3 Summary and Limitations of Previous Works

Several studies have been conducted to understand how developers perform maintenance and evolution tasks. The main limitations of the program comprehension theories (Section 3.1.1) is that they explain the process taking place in the developers' minds, which is difficult to assess in practice. To have more practical understanding of developers comprehension process, explanatory studies have been conducted to assess the external behaviors of developers (Section 3.1.2). These studies reported a common finding that developers explore programs to find where and how to changes code. However, they did not provide a way for systematic identification of developers exploration strategies (*i.e.,* how developers moved between program entities). Yet, some of the previous works (Ko *et al.*, 2006; Robillard *et al.*, 2004) found that program exploration is time consuming and developers usually navigate between and within related program entities.

Through other previous works that used interaction traces to study developers' behaviors (Section 3.2.1), their productivity (Section 3.2.2), and other software engineering activities (Section 3.2.3), we expect that interaction traces are valuable source of information to understand how developers work and assess the effort that they spend during maintenance and evolution task. However, these previous work made several assumptions when mining ITs data and only Robbes et Röthlisberger (2013) defined a way to assess developers effort form ITs data. They computed the effort and related the effort to developers' expertise. They did not focused on understanding how developers spend their effort during maintenance and evolution tasks. Yet, similarly to understanding how developers perform maintenance tasks, understanding how developers spend their effort is a early step towards designing tools to reduce their effort and thus improve their productivity. This understanding open an opportunity of assess whether different exploration strategies result in different effort (*i.e.,* exploration strategies impact the developers' effort).

The limitations above motivate our thesis statement that **we can collect and use accurate interaction traces to assess developers' effort, understand how developers' spent their effort, and assess their program exploration strategies during maintenance and evolution activities.** However, as previous work made several assumptions when using interaction traces, we first study if these assumptions are correct (*i.e.,* if interaction traces contain noises).

CHAPTER 4 NOISE IN INTERACTION TRACES DATA

4.1 Introduction

As reported in Section 3.2, developers' interaction traces (ITs) are commonly used in software engineering to understand how developers maintain and evolve software systems and to improve developers' productivity. Previous studies usually used ITs provided by third-party developers. These ITs collected in the developers' uncontrolled environments are subject to noise because of the interleaving of activities and the interruptions that occur in any normal real-work environment.

Thus, researchers made several assumptions when mining ITs, *e.g.,* (1) edit events with non 0-duration are assumed to be the change performed on source code and (2) the time computed from ITs is assumed to be the time spent by the developers to perform the tasks.

Yet, current tools, such as Mylyn, (1) record 0 duration events, (2) generate events not directly triggered by developers, and (3) define edit events every time a developer selects a piece of text in an editor[1] even if the code does not change. Consequently, interaction traces are likely to contain noise. To the best of our knowledge, no work assessed these assumptions: are edit-events really change activities performed on code? Are the times mined from ITs really the times spent by developers on their tasks?

This chapter aim to (1) quantify noise in current ITs collected by Mylyn, and (2) propose approaches to clean these noise. To reach our goals, first, we conduct a controlled experiment (Wohlin *et al.*, 2000) involving 15 developers to whom we asked to complete maintenance tasks on four open-source Java systems (ECF, jEdit, JHotDraw, and PDE). During the tasks, we collected both Mylyn ITs and video captures of developers' screens. Mylyn ITs—RITs for Raw Interaction Traces in the following—have been often used in the literature to study developers' activities. Video captures have been used as alternatives to ITs (Ko *et al.*, 2006; Robillard *et al.*, 2004; Wang *et al.*, 2013) when transcribed into ITs—VITs for Video-based Interaction Traces in the following. Second, we compare VITs and RITs to assess noise between the two data sets and propose approaches to clean noise in ITs.

4.2 Is There any Noise in Interaction Traces?

In this section, we present how we design the experiment (See Section 4.2.1), collect and process the data (See Section 4.2.2), and study noise in interaction traces (See Sections 4.2.3 and 4.2.4).

[1] http://wiki.eclipse.org/index.php/Mylyn/Integrator_Reference (visited 15/10/2015)

4.2.1 Experiment Setup

Subject Systems

We choose four Java open-source systems[2] because we need their source code, which the participants will modify to accomplish their tasks. We choose two Eclipse-based (plugin) systems (ECF and PDE) and two non-Eclipse systems (jEdit and JHotDraw). ECF (Eclipse Communication Framework) is a set of frameworks for building communications into applications and services. PDE (Plug-in Development Environment) provides tools to create, develop, test, debug, build, and deploy Eclipse plug-ins. JEdit is an open-source text editor. JHotDraw is a Java GUI framework for technical and structured Graphics. We choose these two kinds of systems because (1) we want to decrease threats to generalisability by using two Eclipse-based systems and two other non-Eclipse systems; (2) their source code is open; (3) they belong to different application domains; and, (4) they are well-known and studied in the software engineering community. We also choose these systems because there are RITs available in the bug reports of the two first ones while the other two have no relation with Mylyn.

Object Tasks

We seek concrete maintenance tasks. Thus, for each Eclipse-based system (ECF and PDE), we consider one of their bugs[3]: 202958 and 265931, respectively for ECF and PDE. We consider these bugs because (1) they are already fixed so we know that a solution exist and (2) these solutions can be implemented in reasonable times, about 45 minutes, which we estimated through a pilot study.

For the non-Eclipse systems, we choose one of their version randomly and define one task for each system so that each task requires around 45 minutes. The tasks were defined by exploring the two systems and identifying a possible need: in jEdit, to add lines number and error messages in the select line range dialog and, in JHotDraw, to change the colors of labels and background of the FontChooser.

Participants

We recruit participants via emails, which we send to the authors' research groups and individual contacts. We provide potential participants with a link[4] to an online form for registration, collecting information about their level of study, gender, numbers of years of Java and Eclipse experiences.

[2] https://eclipse.org/ecf/, https://eclipse.org/pde/, http://www.jedit.org/, and http://www.jhotdraw.org/

[3] https://bugs.eclipse.org/bugs/show_bug.cgi?id=202958 and id=265931

[4] http://goo.gl/DyQu6j

We use this information to assign participants to systems so that participants with different profiles work on a same system to minimise threats to internal validity. To avoid learning bias, each participant perform only one task.

In total, 19 participants answered our emails. Out of 19, four did not complete the tasks because (1) three did not have enough Java knowledge and (2) one abandoned the experiment. Thus, we consider only 15 participants: 13 are Ph.D. students in the Department of Computer and Software Engineering of Polytechnique Montréal; one is M.Sc. student at the Czech Technical University in Prague; and, one is a professional software developer.

We perform a Kruskal-Wallis rank-sum test to assess whether the numbers of years of Java and Eclipse experience differ between groups of participants performing the tasks on different systems and found that the differences are not statistically significant (p-values of 0.67 and 0.96). Tasks are performed by participants with similar numbers of years of Java and Eclipse experience.

We use a checklist of the steps of the experiment to consistently provide the same (and only the same) information to each participant. We let each participant know before the experiment that s/he will perform one maintenance task on one Java system for about 45 minutes, although there is no time limit. We also inform he/r that the collected data is anonymous. We ask the participants to try their best to complete the tasks but also tell them that they can leave the experiment at any time for any reason without penalty whatsoever.

4.2.2 Data Collection and Processing

Figure 4.1 summarises how we collect, process, and assess noise in ITs. Participants performed their tasks using the Eclipse IDE together with the Mylyn plugin to collect RITs. The subject systems were hosted in our SVN repository[5]. We imported the systems into the participants' environments and collected their RITs at the end of the tasks (as well as patches for future studies). To collect RITs, we create the task in the IDE. Mylyn starts collecting RITs when the participant activates the task. After the participant completes the task, s/he deactivates the task to stop collecting RITs.

During the tasks, we also captured video recordings of the participants' screens using VLC Media Player 2.0.5[6] at 15 images per second. We transcribed the video captures into ITs manually. We design with two intern students from the Ptidej Team a transcription template after viewing the videos once to minimise subjectivity. Then, we transcribe the videos into CSV files, including the following information. Finally, we construct VITs from these CSV files. (All the data is available

[5] http://goo.gl/Mskalh

[6] http://www.videolan.org/vlc/releases/2.0.5.html

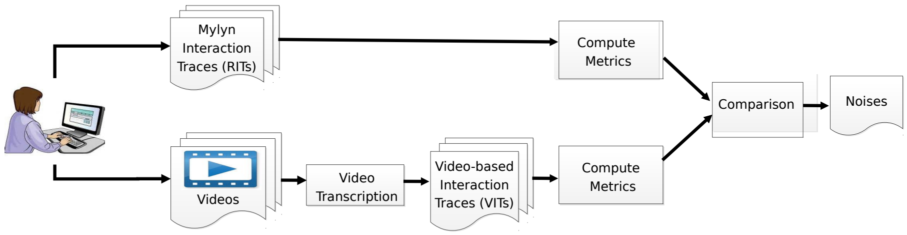


Figure 4.1 Overview of the Approach to Identify Noise in ITs

online[7].)

- Start (hh:mm:ss): The timestamp when the event starts.
- End (hh:mm:ss): The timestamp when the event ends.
- EntityName (string): The name of the entity concerned by the event.
- EntityType (project, package, or file): The type of the entity concerned by the event.
- Origin (Package Explorer, Outline View, Search Result, etc): The part of the IDE where the event was triggered.
- Technique (Visual, Static Relation, Text Search, Reference): The method used by the participants to move from the previous event to the current event. For example, "visual" is when the participant is visually searching a relevant entity by scrolling the package explorer or the code editor, then open a file through the package explorer. We consider that the participant reaches the opened file through visual search. "Static relation" is for example the navigation from a method call to its declaration.
- Activity (Selection, Open, Search, Run, Edit, Close, Other): The activity performed by the participants such as selecting a file (in the package explorer), opening a file, searching through a keyword, running the system, editing the code, or closing an opened file.
- Comments (string): Other information that the transcriber found useful for further analysis.

After collecting data, we study noise at both trace and event levels. In fact, noise can be at the trace-level and–or at the event-level: at the trace-level, the noise would impact the overall time spent on the task (*i.e.,* time-related noise) while at the event-level, the noise impacts the characterisation of edit-events (*i.e.,* time- and edit-related noises).

[7] http://www.ptidej.net/downloads/replications/emse15b/

4.2.3 Trace Level

We compare both RITs and VITs to access all the entities involved in the ITs, the times spent and the activities performed on these entities, and the IDE views used by participants. At trace level, we study the possible noise introduced by 0-duration events and overlapping events because some researchers compute the times spent on tasks by participants considering the start and end timestamps of the whole ITs, *e.g.,* (Robbes et Röthlisberger, 2013). Yet, as ITs may contain noise, we consider both idle and overlaps when computing the times.

Approach

We consider the times spent performing the tasks as defined below and illustrated in Figure 4.2, which is an IT involving three events e_1, e_2, and e_3. We study the mismatches between VITs time (T_{VITs}) and RITs time (T_{RITs}). We compute both T_{VITs} and T_{RITs} in two ways:

- Global time: we compute the global time using the start and end timestamps of the ITs as $GT = End(IT) - Start(IT)$.

  The global time of the IT in Figure 4.2 would be $GT = End(e_3) - Start(e_1)$

- Accumulated time: we compute the accumulated time in an IT as the sum of the times spent on each event:
  $AT = \sum_{event \in IT} End(event) - Start(event)$.

  The accumulated time of the IT in Figure 4.2 would be
  $AT = \sum_{e_i \in \{e_1, e_2, e_3\}} End(e_i) - Start(e_i) = d_1 + d_2 + d_3$

To assess if there is noise in the times computed from ITs, we first compare the global and accumulated time of VITs, on the one hand, and RITs, on the other hand. This comparison aims to figure out whether the two ways of computing times give the same results. If global and accumulated times are the same, to study the noise in time spent on task, we must only compare one of them between VITs and RITs. However, if they are different, we must compare both of them between VITs and RITs. The comparison of the times between VITs and RITs aims to assess the possible noise in RITs (compared to VITs). We use the two-sided Wilcoxon unpaired test to assess the differences stated above because we are not interested in the direction of the difference; we only want to know if there is a difference. We consider that the difference is significant at $\alpha = 0.05$.

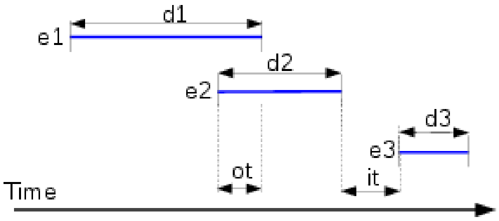


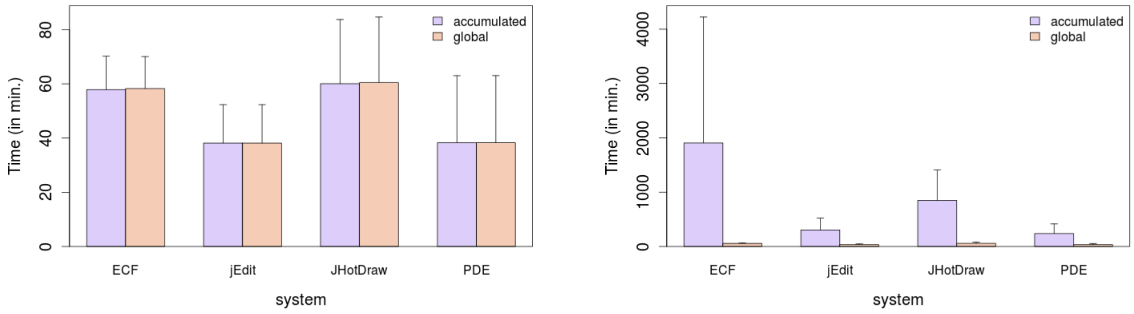
Figure 4.2 Illustration of idle time (*it*) and overlap time (*ot*)

Results and Discussions

Figure 4.3 presents the difference between GT and AT computed from VITs (See Figure 4.3a) and RITs (See Figure 4.3b). Regarding RITs, the difference is statistically significant for two systems (ECF and JHotDraw) and borderline for PDE system (*p*-value exactly 0.05), as shown in Table 4.1 (first part). This result suggests that the two ways of computing time from RITs do not provide the same results. Thus the way the time spent on a task is computed in the previous studies may affect the results found. Figure 4.4 presents the differences in the times spent performing the tasks between RITs (GT_{RITs}) and VITs (GT_{VITs}). Figure 4.4a shows that GT_{VITs} is higher than GT_{RITs}. On the contrary, AT_{RITs} is higher than AT_{VITs} in Figure 4.4b.

We studied whether the differences are statistically significant. Table 4.1 (second part) shows that for ECF and JHotDraw, there are statistical significant differences between AT_{VITs} and AT_{RITs}. For jEdit and PDE, the differences are not statistically significant, but are close, with *p*-values of exactly 0.05. The absence of significant differences for global times between RITs and VITs hints that global time is closer to the actual time spent. We expected this result because participants performed their tasks without interruptions. However, in real-work settings, the GT may not reflect the time spent on the tasks. We argue that in real-work settings, developers may interrupt their work and turn off the collection of the trace. Later, when they turn on the task (collecting traces), Mylyn reloads the previous trace and adds the collected activities. To support this statement, we observe in the RITs (from Bugzilla) some traces that ended a month or more after their start dates. So, the global time in this cases includes times during which developers did not work on the task. Thus, even without statistical significant differences between GT_{VITs} and GT_{RITs}, we claim that global times are not reliable for RITs gathered by developers in their daily work. Robbes et Röthlisberger (2013) relate the development time to developers' expertise by considering the global time as the time spent performing a task (*i.e.,* "the difference between the timestamps of the last and the first events"). The global times seem reliable in our data-set because they come from a controlled experiment.

Moreover, the differences, even not statistically significant, observed in Figure 4.4a indicate a mis-



(a) Video (VITs) time

(b) Mylyn (RITs) time

Figure 4.3 Difference in global and accumulated times

Table 4.1 p-values of the comparison between GT and AT, and between RITs and VITs

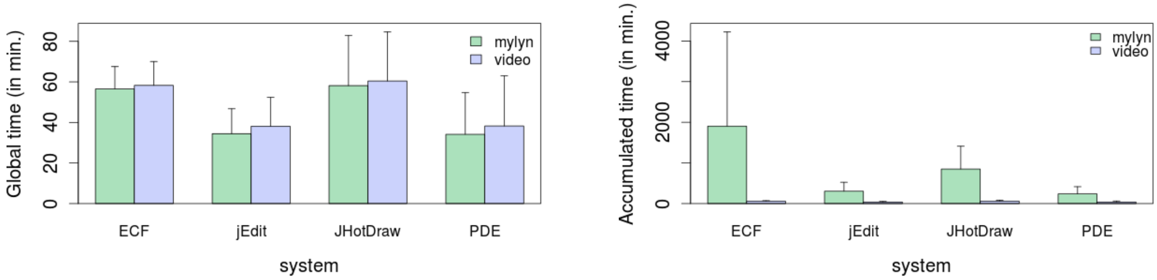
	GT vs. AT		RITs vs. VITs		RITs vs. VITs	
	VITs	**RITs**	**GT**	**AT**	**Idle**	**Overlap**
ECF	0.88	0.02	0.68	0.02	0.02	0.02
jEdit	1	0.1	0.4	0.1	0.06	0.06
JHotDraw	0.68	0.02	1	0.02	0.02	0.02
PDE	0.77	0.05	0.68	0.05	0.02	0.02

match between GT_{VITs} and GT_{RITs}. We explain that this mismatch may be caused by the tool for gathering ITs and the delays between the developers' actions and the collection of the data. We observe that Mylyn starts collecting data when the participants interact with the first program entity, *e.g.,* if a participant starts the task by scrolling without interacting with any entity, Mylyn will not collect any data. This noise does not concern Mylyn only, any monitoring tool may be subject to this noise. Overall, RITs miss on average 2.91 minutes (about 6%). Hence the following observation.

Observation 1: *Mylyn ITs miss on average about 6% of the times spent during the maintenance task.*

Figure 4.4b and Table 4.1 (column "AT" in second part) show that accumulated times are also different between RITs and VITs and that this difference is statistically significant. We assert that this observation comes from idle and overlap times.

We consider that there is an idle time between two consecutive events if there is a non-zero time period between these events, *i.e.,* the second event was not triggered immediately after the end of the first event. There is idle time (*it*) between the events e_2 and e_3 in Figure 4.2. The lack of interactions with the IDE (thinking, reading code or task description) and the interactions that



(a) Global time

(b) Accumulated time

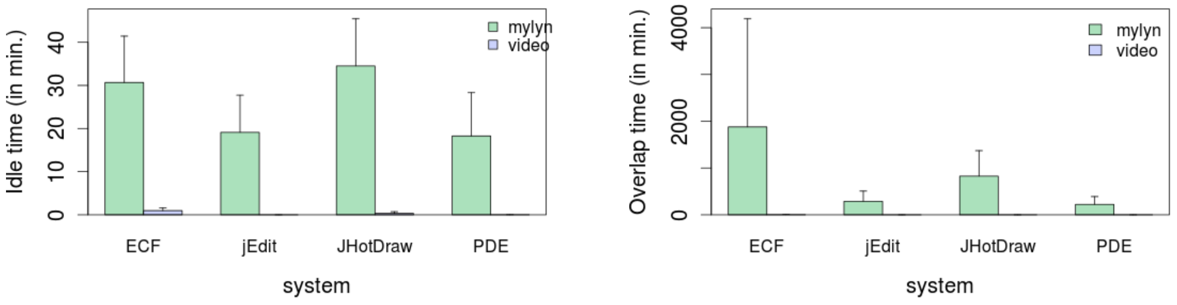
Figure 4.4 Difference in task resolution times between RITs and VITs

the monitoring tool cannot handle (*e.g.,* scrolling) appear in RITs as idle. We cannot distinguish idle times from real inactivities that can happen in real-world settings. Thus, we name all the "inactivity" periods in the RITs as idle times. We consider that there is an overlap between two consecutive events if the second event starts before the end of the first event. There is overlap time (ot) between the events e_1 and e_2 in Figure 4.2. An overlap may occur because of the aggregation of the events and–or the activities involving several program entities (*e.g.,* selection of many files in the package explorer). Idles and overlaps affect the overall times spent to perform tasks. Yet, previous study did not consider idles and overlaps when computing global times, *e.g.,* (Robbes et Röthlisberger, 2013).

Figure 4.5 shows the cumulative durations of idles and overlaps. It shows that these phenomena are prevalent in RITs compared to VITs. It also shows that overlap times are more important in duration than idle times. Table 4.1 (third part) shows that the differences between idle and overlap times in RITs and VITs are statistically significant for three systems (ECF, JHotDraw and PDE) while it is closed to be significant (p-value = 0.06) for JEdit. Overall, RITs contain a average cumulative idle and overlap times of 26.08 (median 27.55) minutes and 579.3 (median = 250.4) minutes, respectively. While idle time can be seen as the time spent reading code, thinking or understanding the code, the overlap time seems too much. However, it can be due to the overlap between many events (multi-overlaps).

Observation 2: *Mylyn ITs involve on average about 53% of idle times and 1,171% of overlap times.*

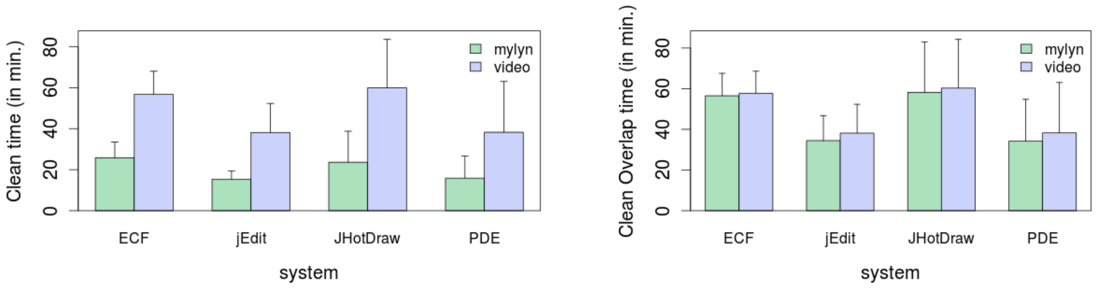
When we recompute the accumulated times after removing idle and overlap times, as shown in Figure 4.6a, the new accumulated times of RITs drop down, but is still different from the global time (See Figure 4.6a vs. Figure 4.4a). However, if we keep the idle times but remove only the overlap times (See Figures 4.6b), then the accumulated and global times are similar, both for VITs



(a) Idle times

(b) Overlap times

Figure 4.5 Difference in idle and overlap times between RITs and VITs



(a) Clean idle and overlap times

(b) Clean overlap times

Figure 4.6 Difference in clean times between RITs and VITs

and RITs (See Figure 4.6b vs. Figure 4.4a). Thus, we consider that idle times that may appear in the RITs collected in a real-world setting are not always interruptions.

The last observation arises from our interest to quantify the magnitude of individual idle times. We investigate this magnitude because the cumulative durations of these idle times can reach an average of 26 minutes, and the data collected by developers in their daily work may involve real interruptions. Figure 4.7 (durations of individual idle) shows that idle times in RITs can reach more than five minutes. However, overall, idle times lasted 0.5 (median = 0.07) minutes.

Observation 3: *Mining times from ITs requires to remove overlap times from accumulated times. Idle times that last less or equal to half a minute should not be consider as interruptions and should be included in the time spent on task.*

By quantifying the average duration of idle events that are not interruptions, our result complements Sanchez *et al*. (2015) who was inspired by previous study and defined interruptions in RITs as a

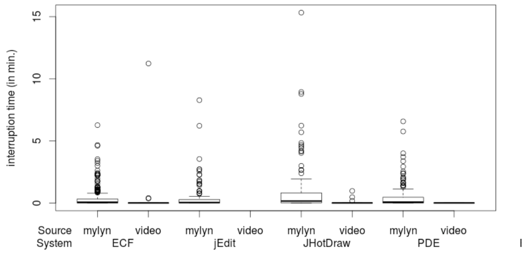


Figure 4.7 Distribution of the time of individual idle times for RITs

pause of programming of duration greater or equal to 3 minutes. Our threshold of idle times is in the context of RITs *i.e.,* considering idle times within developers' activities. The threshold of interruption times defined by González et Mark (2004) was found by considering more general workers' activities as the considered workers were four developers, six business analysts, and four managers.

4.2.4 Event Level

At the event level, we focus on the noise for edit events because (1) edit events are more accurately identifiable from video data than other events as it is trivial to see in the videos if the code is being changed and (2) edit events are subject to several assumptions, such as being representative of the activity of changing source code (Lee *et al.*, 2015; Sanchez *et al.*, 2015; Soh *et al.*, 2013). Also, edit events have been used to build code recommendation tools (Kersten et Murphy, 2005; Lee *et al.*, 2015) and as proxy to productivity (Kersten et Murphy, 2006; Sanchez *et al.*, 2015). Thus, any noise related to edit events would impact such previous studies.

Approach

We consider the numbers of edit events and the times spent performing edit events to investigate noise in the identification of edit events, *i.e.,* any differences between VITs and RITs in numbers of edit events and the times spent performing edit events.

- Numbers of edit-events: we count the edit-events after removing events whose start and end timestamps are equal (*i.e.,* 0-duration), which were considered selection-events in previous studies (Lee *et al.*, 2015; Sanchez *et al.*, 2015).
- Edit duration: we compute the edit durations after removing overlap times.

To evaluate the difference between VITs and RITs for the numbers of edit events and edit durations, we use the same statistical test as at trace level (Section 4.2.3).

Results and Discussions

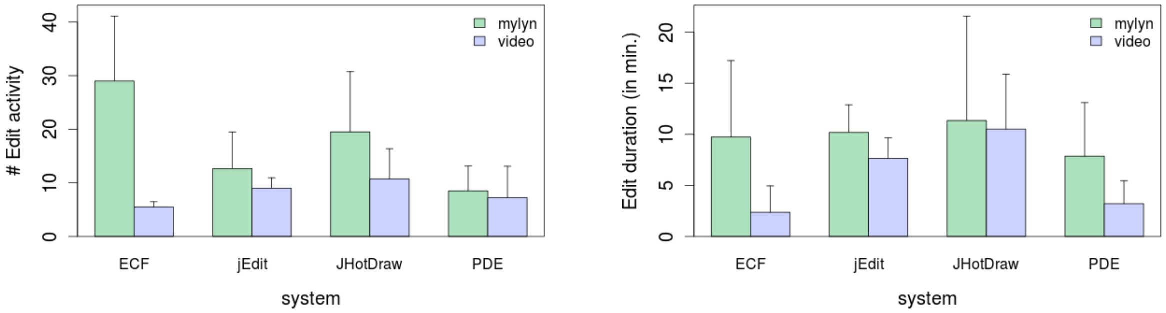
We observe differences in VITs and RITs according to their numbers of edit-events, see Figure 4.8a, and the durations of the edit-events, see Figure 4.8b. The differences are not statistically significant as revealed by the Wilcoxon test results from Table 4.2. The numbers of edit events and the durations of the events mined from RITs are not statistically different to that from VITs.

The lack of statistical significant difference in the numbers and durations of edit events is the consequence of the task resolution *i.e.,* how successfully participants resolved the tasks. Indeed, participants performed few changes. Only 12 out of the 15 participants performed at least one edit event while only five participants successfully resolved their tasks (and three participants partially). Overall, eight participants (53.33%) changed the code during their task. As resolving the task requires to change the code, results of the statistical tests would be different if all participants performed changes.

However, even though they are not statistically different, the observed difference in Figure 4.8 reveals that there are some edit-events that are not real changes to the code (See Section 4.3.1 for the details about these edit-events). We call these edit-events “false edit-events” and define the measure edit bias $edit\_bias = \frac{false\_edit}{real\_edit}$, where $false\_edit = edit(RITs) - edit(VITs)$ and $real\_edit = edit(VITs)$. Using the edit bias, we observe a bias of, on average about 28%, which contradicts the assumption that all non 0-duration edit events correspond to changes to the code.

Observation 4: *Interaction traces contain about 28% of false edit-events.*

Without any statistical difference between the number of edit-events of VITs and those of RITs, we look at the proportion of false and true edit events in RITs. We observe that RITs contain overall about 36% of edit-events vs. 74% of selection events (See Figure 4.9a). About 33% of these edit-events are edit-events with 0-duration (See Figure 4.9b). Considering the edit-events with non 0-duration (about 67%), 92% of them are false edit-events and only 8% are true edit-events (See Figure 4.9c). We reported our observation to the Mylyn community to have their opinion on these false edit-events. The feedback we got from the Mylyn community confirms our observation. One leading developer of Mylyn stated that “... *the argument that there is noise in the edit events makes sense to me.*”. They justify the prevalence of false edit-events by the original premise of Mylyn edit events. “*The original premise of the edit events was that time spent in the editor is more productive than time spent looking around the structure views that eventually get you*



(a) Numbers of edit-events

(b) Durations of edit-events

Figure 4.8 Difference in edit-events between RITs and VITs

Table 4.2 Comparison between T_{VITs} and T_{RITs} edit-events

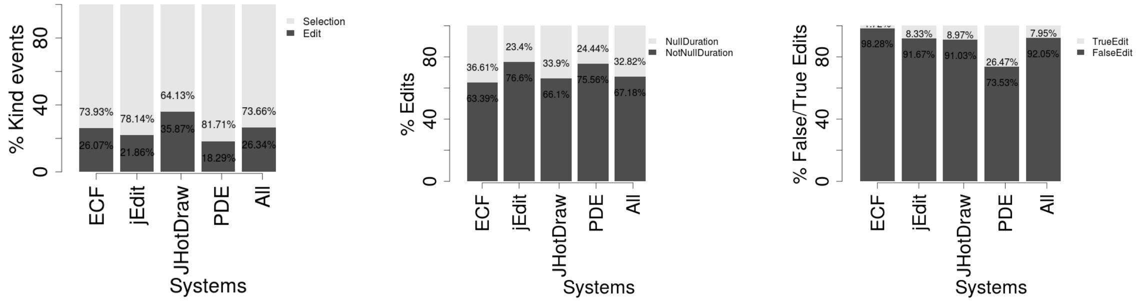
	p-value	
	Number edit	**Edit duration**
ECF	0.13	0.26
jEdit	0.8	0.4
JHotDraw	0.34	1
PDE	0.66	0.34

to the editor. The edit events don't have to be textual edits, which is why we decided not to base this event on something like keystrokes in the editor. For example, selecting a bunch of text, then copying and pasting it, could produce a number of edit events, and be more meaningful work than 30 keystrokes.". However, when considering selections of text as true "edit" events as suggested by the Mylyn developers, we could still find about 75% of false edit-events in the studied ITs.

After observing that noise is prevalent in ITs, we looked at how we can address false edit-events noise.

4.3 Can we Address the Noise in Interaction Traces?

In this section, we focus on addressing edit-related noise (*i.e.,* false edit-events). We first present the threshold-based approach, then the prediction-based approach that outperforms the threshold-based approach.



(a) Edit vs. Selection Events

(b) Edit Events with null and not null duration

(c) False vs. True Edit Events

Figure 4.9 Proportion of the Events

4.3.1 Threshold-based Approach

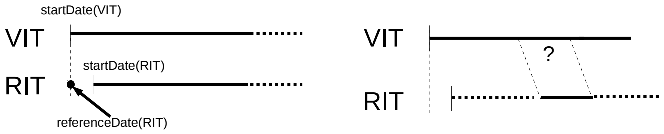
The threshold-based approach consists to (1) align RITs and VITs, (2) use the alignment to identify false edit-events in RITs, and (3) define a threshold and rules to automatically classify RITs edit-events as false or true edit-events.

Alignment

We align VITs with RITs to generate cleaned ITs (CITs) corresponding to RITs. The alignment consists in identifying VITs events corresponding to RITs events. We use the timestamps as keys for traces alignment. First, we manually identify the *RITs start timestamps reference* ("reference timestamps" in the remainder), which is the RITs timestamps corresponding to the start timestamps of VITs, see Figure 4.10a. Timestamps are in different format and have different precision. RITs timestamps are more precise (in term of milliseconds) than reference timestamps (in term of minutes). This difference in precision could affect the alignment of the events but, given the durations of the events, a precision in minutes is sufficient.

Second, we use the reference timestamps to compute the exact time when an event starts and ends. For example, consider that an event e started at timestamp d_1 and ended at timestamp d_2. We compute the start and end time of the event e as $startTime(e) = time(d_1) - time(d)$ and $endTime(e) = time(d_2) - time(d)$ where d is the reference timestamp (*i.e.,* the timestamp corresponding to the start of the video recording) and $time(x)$ is the number of milliseconds[8] to reach the timestamp x. Having the start and end time of each event in a VIT and a RIT, we can align two events if they (ideally) start at the same time and end at the same time. However, the alignment is not always ideal. To avoid wrong alignments due to overlap times between events, we

[8] We use the method getTime() of the Java Date class.



(a) Reference start timestamps (b) Alignment principle

Figure 4.10 Illustration of the alignment of RITs and VITs

remove overlap time intervals before aligning the events, *e.g.,* for consecutive events e_1 and e_2, if $End(e_1) > Start(e_2)$, we consider that $End(e_1) = Start(e_2)$. This cleaning does not affect the start time of an event which the alignment is mostly based on. The alignment consists to search the events in a VIT corresponding to each event in RITs, see Figure 4.10b. For each event in RITs, we scan the VITs and search for the corresponding event(s).

We consider that an event in a RIT is aligned to an event in a VIT if the intersection of their time period is not empty. For example, if an edit event e in a RIT is aligned with n events $\{e_1, e_2, ...e_n\}$ in a VIT, we consider that the edit event e is aligned to the first edit event in $\{e_1, e_2, ...e_n\}$. We could use DTW (Dynamic Time Warping) for alignment, but our goal is not to minimise the distance between RITs and VITs as does DTW. Instead, we align vertically events and identify matching events in a RIT and in its corresponding VIT.

Identification of False Edit-events

Through the alignment, we identified VITs events that correspond to RITs events. RITs involved two main kinds of events: selection and edit. We consider that all edit events with non 0-duration in RITs that are not aligned with any edit event in VITs are false edit-events. An edit event in RITs aligned with an edit event in VITs is a true edit-event. Table 4.3 shows the distribution of the time spent on false and true edit-events. It reveals that false edit-events with non 0-duration take on average 24.01 seconds (median = 4.14), while true edit-events take on average 143.7 seconds (median = 114.4).

For the purpose of scalability (*i.e.,* to be able to identify false edit-events in the existing RITs), we must define a threshold to accurately classify false and true edit-events. Thus we use different values of thresholds, then we evaluate their accuracy and finally we choose the most convenient threshold.

Table 4.3 Distribution of the Time spent on Edit Events with non 0-duration

	Time (in seconds)					
	Min	**Q1**	**Median**	**Q3**	**Max**	**Mean**
False Edits	0.001	0.52	4.14	28.33	371.00	24.01
True Edits	1.14	62.6	114.4	221.8	341.2	143.7

Threshold for False Edit-events

We classify false and true edit-events using the thresholds per steps ±5 seconds from the rounded mean duration of false edit-events (*i.e.,* 24 seconds in Table 4.3). For each threshold, we use the precision, recall and F-measure to measure the accuracy of our classification. We compute these metrics using the confusion matrices in Table 4.4.

- Precision of false edits measures the proportion of classified false edits that are correct.
  $Precision = \frac{TP}{TP+FP}$
- Recall of false edit events measures the proportion of correctly classified false edit events.
  $Recall = \frac{TP}{TP+FN}$
- F-measure is used as a trade-off between precision and recall.
  $F\text{-}measure = \frac{2*Precision*Recall}{Precision+Recall}$

The metrics to evaluate the classification of true edits are computed in the same way.

Figure 4.11 shows the accuracy of false and true edit-events. The classification of false edit is more accurate (highest F-measure) at 179 seconds, while the threshold of 110 seconds maximizes both precision and recall of false edit-events (See the gray vertical lines in Figure 4.11a). Similarly, the classification of true edits is more accurate (highest F-measure) at 179 seconds, while the threshold of 187 seconds maximizes the precision and recall of true edits (See the gray vertical lines in Figure 4.11b). The threshold of 189 seconds maximizes the F-measure of both false and true edit-events (See the gray vertical lines in Figure 4.11c). Maximizing the F-measure of false and true edit events (using 189 seconds as threshold) results in the misclassification of most of the true edit-events (See the gray horizontal line in Figure 4.12). Because there is a low proportion of true edit-events in our dataset, we also aim to prioritize the recall of true edit events. Moreover, in practice some true edit-events may last for only a short time (*e.g.,* refactoring that none of our participant did). Using the mean duration of false edit-events misclassifies only three true edit-events (See the blue horizontal line in Figure 4.12). Thus, we consider the mean duration of false edit-events as the threshold that achieves 93% precision and 64% recall for false edits, and 18% precision and 81% recall for true

Table 4.4 Confusion Matrices for the Evaluation of False and True Edits

	Actual Label	
Classified as	**true edit**	**false edit**
true edit	TN	FN
false edit	FP	TP

(a) Identification of false edits

	Actual Label	
Classified as	**true edit**	**false edit**
true edit	TP	FP
false edit	FN	TN

(b) Identification of true edits

edits. Using both the mean and third quartile (28.33 seconds) of the times of false edit-events as thresholds does not yield major differences on the accuracy of false edit-events. Thus, we assume that:

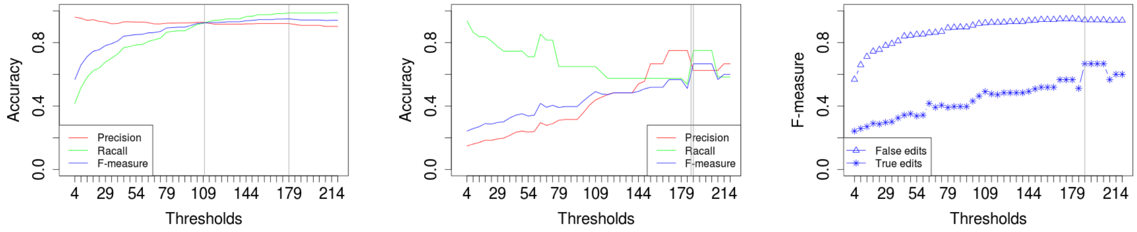
Observation 5: *An edit-event is a false edit-event if it takes less than 24 seconds but more than 0 second.*

When looking at the alignments, most of the false edit-events correspond to the opening of a file (*e.g.,* double click to open the file), the navigation of static relation between program entities and the text-based search in the file. When we look at RITs, we did not find any indicator to distinguish which false edit-event corresponds to each of the VITs event above. However, one case of open event was identifiable, *e.g.,* a false edit-event which occurred after an event triggered through the search view. We build our cleaned approach with the annotation rules in Table 4.5. For example, the first row in the table means that if an event is an edit event that lasted less than or 24 seconds and if the previous event was triggered through a search view, the event should be considered an open event.

Regarding selection events, the alignment reveals that selection events can occur through different views (package explorer, outline view, search view, editor). While RITs equally considers these selection events, the video captures show that selection events in the editor or type hierarchy views (compare to selection events through other views) are performed when the participants expect the entities to be relevant. This observation is consistent with Ko *et al.* (2006), who stated that the selection in the editor view indicates a perception of relevance.

Hence, a selection event in the editor and type hierarchy views is likely to indicate the relevance of an entity, we consider the selection of an entity in the editor as the inspection of the entity because the participants seem to investigate the entity to see if it is relevant or to understand it. Thus, we annotate the selection event as indicated in Table 4.6.

Our annotation rules (Tables 4.5 and 4.6) provide a way to clean RITs. Some edit-events really should be navigation- and open-events while some selection-events really should be inspection-



(a) Accuracy False Edit (b) Accuracy True Edit (c) F-measure False and True Edit

Figure 4.11 Precision, Recall, and F-measure of False and True Edit Events

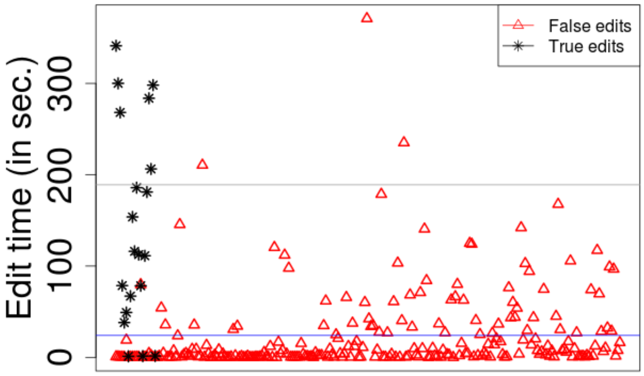


Figure 4.12 Time Spend on False and True Edit Events

events. However, this threshold-based approach achieved low precision for classifying true edit-events.

4.3.2 Prediction-based Approach

To overcome the limitations of the threshold-based approach (*e.g.,* low recall for true edit events), we propose an approach for systematic classification of edit events. We use several classification algorithms from Weka[9] (Naive Bayes, J48, Random Forest) to predict the kind of an edit event. We first compute the following metrics that we expect to be related to the kind of edit events:

- Element type: The type of the element on which the event occurred (package, file, class,

[9] http://www.cs.waikato.ac.nz/ml/weka/ (visited 15/10/2015)

Table 4.5 Rules for annotation of edit events

RITs events	Times	Previous event	Annotated events
Edit	$time \leq 24sec$	Search view	Open
Edit	$time \leq 24sec$	Other	Navigation
Edit	$time > 24sec$	N/A	Edit

Table 4.6 Rules for annotation of selection events

RITs events	Origin	Annotated events
Selection	Editor	Inspection
Selection	Other	Selection

method, attribute).

- Origin ID: The IDE part that the event is triggered on.
- Interest: The degree of interest that measures the relevance of the program entities involved in the event.
- Raw duration: The duration of the event including overlaps if any.
- Clean duration: The duration of the event after removing overlaps.
- Idle time: The idle time between the event and the next event.

To predict the kind of edit events, we first split the data into training and test sets. We use the random sampling method without replacement (*i.e.,* training and test sets do not contain common instances). The training set contains 60% of the data and test set 40% of the data. We obtain an imbalanced training set *i.e.,* the classification categories (false and true edits) are not approximatively equally represented in the training set (225 = 95% of false edit events vs. 12 = 5% of true edit events). In our training set, false edit is the majority class while true edit is the minority class. To build our models, we need to sample the data to avoid over fitting. There are two ways to sample the data: under sampling and over sampling. Under sampling consists to reduce false edit events (majority class) from the data set and over sampling consists to add true edit events (minority class) to the data set. We apply both under and over sampling to the training set because the sampling method can results in different improvements of the prediction model (Kamei *et al.*, 2007).

We apply random under sampling using the *SpreadSubSample* method from Weka environment, sampling majority class from 225 instances to 12 instances. We apply the over sampling using the implementation of SMOTE (Synthetic Minority Over-sampling Technique) method (Chawla *et al.*, 2002) in Weka, sampling the minority class from 12 instances to 221 instances. We build the models using both under and over sampled training set. Then we re-evaluate the models on the test set without any balancing of the test set. Table 4.7 and 4.8 show the evaluation of the prediction algorithms for under sampling and over sampling, respectively.

All the models with under sampling show the low precision of true edits that are improved by the over sampling. Overall, using over sampling provides better results than under sampling. The Random Forest and J48 provide the best results with over sampling. But J48 is better in term of the prediction of false and true edit events, in term of the recall of true edit events, and in term of the low Mean Absolute Error (MAE). Thus we conclude that J48 using over sampling provides the best results for the prediction of edit events. The most important attributes reveal by J48 for edit events prediction are in order of importance: clean duration, raw duration, element type, origin ID, interest, and idle time.

Our approaches for cleaning ITs show that we address these noises. However, as any empirical study, our study suffers from some threats.

4.4 Threats to Validity

This section discusses the threats to the validity of our study.

Interaction Traces

We use interaction traces collected by the Mylyn Eclipse plugin[10]. While many tools exist for collecting interaction traces (*e.g.,* Mimec (Layman *et al*., 2008)), we cannot guarantee that Mylyn provides the most accurate ITs and we cannot assume that interaction traces collected by other tools contain similar noise (*i.e.,* false edit actions) and–or that our cleaning approaches may be suitable for other tools' interaction traces. However, Mylyn is the most used tool for collecting ITs in industrial projects.

Another threat when using Mylyn is the version of the tool. During our experiment, we use different versions of Eclipse and, consequently, different versions of Mylyn, for different tasks. We were constrained in our controlled experiment to use specific versions of the Eclipse-based systems (*i.e.,* ECF, PDE) because we aimed to use the versions against which the bugs were reported. Using different versions means that ITs could be different. However, as features are not significantly different between versions of Mylyn, the differences between the ITs of two versions of Mylyn are negligible.

The overlap phenomenon can also affect our results. We considered the overlap between two consecutive events but not among several events, which could affect our results.

[10] http://eclipse.org/mylyn/ (visited 15/10/2015)

Table 4.7 Evaluation of the Prediction Algorithms (under sampling)

Algorithm	Accuracy	Precision	Recall	F-measure	MAE
NaiveBayes	69.18	(99, 14.3)	(68, 88.9)	(80.6, 24.6)	32.48
J48	76.72	(99.1, 18.2)	(76, 88.9)	(86, 30.2)	23.27
Random Forest	77.98	(99.1, 19)	(77.3, 88.9)	(86.9, 31.4)	28.84

(x, y) = (false edits, true edits)
MAE = Mean Absolute Error

Table 4.8 Evaluation of the Prediction Algorithms (over sampling)

Algorithm	Accuracy	Precision	Recall	F-measure	MAE
NaiveBayes	88.67	(97.8, 28.6)	(90, 66.7)	(93.8, 40)	16.52
J48	89.30	(98.5, 31.8)	(90, 77.8)	(94.1, 45.2)	10.77
Random Forest	89.93	(97.2, 29.4)	(92, 55.6)	(94.5, 38.5)	13.2

(x, y) = (false edits, true edits)
MAE = Mean Absolute Error

Transcription of the Video

Three peoples in our research team independently translated the videos into ITs. Because some actions may be difficult to interpret and because the exact start and end timestamps of an action may be difficult to identify, different transcriptions of a video may provide different results. To mitigate this threat, we defined a common template for video transcription. Moreover, the noise studied are related to the time and the edit events and edit events are easy to identify on a video capture.

Alignment of Mylyn and Video ITs

To identify the mismatch between RITs and VITs, we align both Mylyn and video captures ITs. The alignment consists of identifying the events corresponding to one another in the two sets of ITs. As the alignment is based on the times when the events occur, any shift may result in the lost of alignment between two events that actually correspond. To mitigate this threat, we manually checked some alignments and observed that the false edit-events, for example, were not due to shifts in alignments. We also evaluated the accuracy of our alignment approach by manually classifying edit events as true or false edit events. We computed the precision, recall, and F-measure of the alignment. Our alignment approach achieves 91% to 100% precision and 91% to 98% recall for false edits. To identify true edit events, our alignment approach achieves 25% to 75% precision and 14% to 100% recall. The accuracy of our alignment of false edits supports the limited impact of the alignment on our results because we studied and aimed to identify false edits.

Cleaning of ITs

We clean ITs using two approaches. The accuracy of the threshold-based approach depends on the alignment and time spent on false edit-events. The time spent on false edit-events may be different for other systems and–or other developers (*e.g.,* developers' experience and speed when triggering events). We introduce the prediction-based approach to mitigate the threat from the threshold-based approach.

Task and Systems

When investigating noise in ITs, the participants performed maintenance tasks on Java systems only. Hence, we cannot guarantee that using other systems and other programming languages would yield the same results. We choose the tasks to be accomplished in about 45 min. More complex tasks would take more time and developers would have more interruptions and would switch more often between tasks. Therefore, the tasks considered in our experiments may not be representative of tasks performed by developers in industrial settings. However, as we aimed to assess possible noise, we must have an oracle against which to compare collected ITs. Thus, we were constrained to collect ITs in an experimental setting.

Reliability

We provide all the data online. For the manual work required to replicate our study, we design a video transcription template and three people transcribed the videos. Thus, we think that the transcriptions by other people would not be significantly different from ours (*e.g.,* identifying edit-events from the video is not challenging). Moreover, we automated the alignment of events between RITs and VITs. Our tool can be used after video transcription without any manual work if the replication setting allows videos and RITs collection to start at the same time (*i.e.,* no need to manually identify the reference timestamps).

4.5 Summary

We designed and conducted an experiment to study noise in ITs. During our study, we collected both ITs and video captures of the participants' screens. We compared the collected ITs (RITs) and video captures (VITs) and observed that ITs miss up to 6% of the times spent performing tasks and that they contain about 28% of false edit-events.

By looking in details at the difference between the collected data sets (*i.e.,* RITs and VITs), we proposed two approaches to clean false edit-events (*i.e.,* threshold- and prediction-based approaches). The prediction-based approach outperformed the threshold-based approach by achieving up to 98% precision and 90% recall for classifying false edit-events.

In light of these results, we conclude that previous studies have made incorrect assumptions on ITs.

We recommend that researchers and practitioners interested in ITs do not make these assumptions on ITs. Applying our recommendations, we investigate if noise impacts the results of previous studies.

CHAPTER 5 IMPACT OF NOISE ON PREVIOUS STUDIES

5.1 Introduction

The presence of noise in ITs shows that previous studies that made assumptions related to these noises are likely to have some inaccuracies. For example, the IT in Figure 4.2 shows an excerpt of the edit events recorded for one developer during our empirical study explained in the previous chapter. The edit events e_1, e_2, e_3 would be labelled by Mylyn as edit-events but they are *false* edit-events: events during which the developer did not modify the source code (when observing the corresponding screen capture). Therefore, using this IT, the developers' edit style (Ying et Robillard, 2011) would be wrongly inferred as *edit-throughout* instead of *edit-last*, even though edit-events *really* occurred at the end of the task.

Moreover, using false edit-events when recommending entities to edit would affect the accuracy of the recommendation systems. Indeed, a recommendation approach based on ITs (Lee *et al.*, 2015) considers the edit actions that occurred on an entity as an indication of the relevance of the entity. Thus, an entity would be wrongly considered as relevant if false edit-events occurred on it. Therefore, studies that use the number of edit-events in RITs may over/under estimate the number of edit-events because RITs contain false edit-events.

In this chapter, we use our cleaning approaches to revisit previous studies to investigate the impact of noise on their findings. We select two previous works representative of two families of studies: (1) studies that used ITs to understand developers' behaviors during development and maintenance activities and (2) studies that leveraged IT data to recommend program entities to developers performing development and maintenance tasks. More specially, we assess how noise impact the identification of developers' editing styles (Ying et Robillard, 2011) and the accuracy of recommendation of relevant program entities (Lee *et al.*, 2015). We choose these two studies for two reasons. First, they are studies that investigate the aspects pertaining to the assumptions made on ITs. Second, we have access to the data and–or the scripts/source code used in these studies and we can use them for fair comparison of their results with and without cleaning noise.

5.2 What is the Effect of the Noise on Developers' Editing Style?

We now assess whether applying our approach impacts the results from Ying et Robillard (2011). We first give an overview of the approach to identify editing style, then we explain how we assess the impact of noise on the results reported by Ying et Robillard (2011), and finally we report our findings.

5.2.1 Developers' Editing Styles

Ying et Robillard (2011) defines the developers' editing style as when a developer edits code during a programming session. They characterise three types of editing styles based on the fraction of edit events at different points in the trace: *edit-first*, *edit-last*, and *edit-throughout*. An ITs is categorised as edit-first if a high fraction of edit events occurs in the first half of the trace. The edit-last as if a lower fraction of edit events occurs in the first half of the trace, and edit-throughout if a trace if not edit-first or edit-last.

To categorise an IT, Ying et Robillard (2011) first normalized the distribution of the IT timestamp between 0 and 1 where 0 represent the time of the first event and 1 the time of the last event. Then, they classify all edit events in the trace as the first half edit event if more percentage of the duration of the event resides in the first half of the trace. Figure 5.1 illustrates the classification of edit event in the trace. For example, the edit event e_1 is in the first halp while the event e_2 is not in the first half. Finally, they categorise the trace using the fraction of edit events in the first half (*i.e.,* the number of edit event in the first divided by the total number of edit events in the IT). By using the distribution of the edit first event, they observe that a trace is edit-last if the proportion of first half edits is between 0% and 19%, edit-last if the proportion of first half edits is between 87% and 100%, otherwise, they categorise the trace as edit-throughout.

Our intuition is that as the categorisation of trace is mainly based on the edit events and their duration, these categorisation could be affected by the noise in ITs.

5.2.2 Approach

We use the script[1] provided by the authors of the study to compute the editing styles of ITs. We apply their approach on two data-sets. The first data-set contains the RITs and VITs collected during our experiment in Section 4.2. The second data-set contains ITs collected from bugs reports in Bugzilla for the ECF, Mylyn, PDE and Platform, called bRITs (for Bugzilla's RITs) in the following and containing 1,970 ITs. We clean RITs and bRITs using the two approaches described in Section 4.3 to obtain CITs and bCITs, respectively.

We use precision and recall to compare the editing styles identified in RITs, VITs, and CITs,

[1] http://www.cs.mcgill.ca/aying1/2011-icpc-editing-style-data.zip

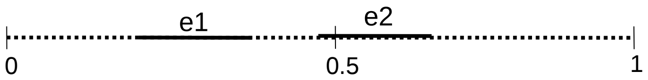


Figure 5.1 Overview of the Classification of Edit Event

considering VITs as oracles.

$$Precision(s) = \frac{\text{\# traces correctly identified as style } s}{\text{\# traces identified as style } s}$$

$$Recall(s) = \frac{\text{\# traces correctly identified as style } s}{\text{\# traces of style } s}$$

where $s \in \{\text{edit-first}, \text{edit-throughout}, \text{edit-last}\}$

To evaluate the result of bRITs and bCITs, we compute the numbers of miscategorisation between bRITs and bCITs, *e.g.,* when an IT categorised as *edit-first* (using bRIT) becomes *edit-throughout* or *edit-last* (using the corresponding bCITs), which we note "style X → others" with $styleX \in \{\text{edit-first}, \text{edit-throughout}, \text{edit-last}\}$. Similarly, an IT may change the editing style from other style (*e.g., edit-throughout* or *edit-last*) to a style X (*e.g., edit-first*).

We report and discuss the result of our analysis in the next section.

5.2.3 Results and Discussions

Table 5.1 presents the results of identified editing styles for VITs, RITs and CITs (using the threshold-based approach) for the 12 participants who performed at least one edit event in our experiment. The table shows that one, nine, and two traces are identified in VITs as *edit-first*, *edit-last*, and *edit-throughout*, respectively. Compared to editing styles identified in RITs, nine traces $(11-2)$ are wrongly identified as *edit-throughout*. The nine traces are one *edit-first* and eight *edit-last*. Similarly, seven traces $(9-2)$ are wrongly identified as *edit-throughout* using CITs. The seven traces are one *edit-first* and six *edit-last*. These results shows that both RITs and CITs contain noise that affect the identification of editing styles. However, when comparing RITs with CITs, using CITs improves the precision and recall by about 22% (*edit-last*) and 4% (*edit-throughout*). The use of prediction-based approach to clean ITs improves the accuracy of edit-first style from 0% to 50% for precision and from 0% to 100% for recall. On the contrary, the prediction-based approach decreases the accuracy of edit-throughout style from 22% to 14% precision and from 100% to 50% recall. The accuracy of edit-last style remains unchanged. We compare the average

Table 5.1 Editing Styles Betwwen VITs, RITs, and CITs

	VITs	RITs			CITs		
		#	P(%)	R(%)	#	P(%)	R(%)
edit-first	1	0	0	0	0	0	0
edit-last	9	1	100	11.11	3	100	33.33
edit-throughout	2	11	18.18	100	9	22.22	100

Table 5.2 Editing Styles Between bRITs and bCITs

	bRITs	bCITs	bRITs ∩ bCITs	style → others	others → style
edit-first	163	521	142	21 (1.06%)	379 (19.23%)
edit-last	546	438	279	267 (13.55%)	159 (8.07%)
edit-throughout	1,261	1,011	872	389 (19.74%)	139 (7.05%)
All	1,970	1,970	1,293 (65.63%)	677 (34.36%)	677 (34.36%)

precision and recall of the two approaches (threshold-based vs. prediction-based) and observe on the experiment dataset that the prediction-based approach improves the precision by about 14% (40.74% vs. 54.76%) and recall by about 7% (44.44% vs. 61.11%).

Table 5.2 shows similar trends for bRITs and bCITs. About 66% of the ITs conserve their editing style in both bRITs and bCITs (column "bRITs ∩ bCITs") while editing styles changed for 34% (column "style → others"). These results show that our approach (modification of traces) yields in some cases different categorisation of ITs and, hence, that noise impacts the results of this previous study. Using the prediction-based approach on Bugzilla dataset, 58.55% of traces conserve their editing styles, while 41.44% change their editing styles. Thus, the prediction-based approach of trace correction increases by 7% the number of traces that change their editing styles.

Using our experiment data-set and the two approaches to clean traces shows a small difference in the identification of editing styles. We use the categorisation technique from the original work, which is based on the percentage of edit-events in the first half of an IT (Ying et Robillard, 2011). The considered threshold to identify editing style was inferred from a large set of ITs. We explain the small difference on the experiment data-set by the small size of the data-set as the threshold used may not be suitable for this amount of IT. However, the use of Bugzilla data-set, which may be more suitable for the threshold due to the size of the data-set, shows about 34% and 41% of changes in editing styles using the threshold- and prediction-based approach, respectively.

As example of differences between ITs, we observe that the trace of participant S06 is categorised as *edit-first* when using VITs because, as expected, it has only edit-events in its first half. The corresponding trace has 29% and 50% of edit-events when considering RITs and CITs, respectively. We assumed that applying our cleaning approaches, which improve the categorisation of RITs wrt.

VITs for the experiment data-set would also improve the categorisation of RITs from the Bugzilla data-set. We cannot verify this assumption because we do not know the true categorisation for the Bugzilla dataset (*i.e.,* the oracle).

5.3 Can Correcting Noise Improve the Accuracy of Recommendation Systems and enable Early Recommendations?

Complementary to the impact of noise on editing styles (Section 5.2), we investigate the impact of false edit-events on the accuracy of recommendation systems built using ITs. We provide the background on recommendation systems using ITs (Section 5.3.1) and show how false edit-events can impact the recommendation (Section 5.3.2). Then, we describe the design of our study (Section 5.3.3). Finally, we report and discuss our results (Section 5.3.4).

5.3.1 Recommendation Systems using Interaction Traces

A typical recommendation approach (Lee *et al.*, 2015) includes the following steps: (i) mining interaction traces on the fly, (ii) forming context at each timepoint (*i.e.,* event), and (iii) using association rules to recommend files to edit. For N interaction traces $T_1,...,T_N$, each trace is used in both training and testing sets, except the first trace which is used only in the training set. In trace $T_k, k \in 1,...,N$, all traces $T_1,...,T_{k-1}$ are used for training and T_k is used for testing. Each trace is a pair of two sets $T_k = (V_k, E_k)$, where V_k is the set of files on which view actions occurred (*i.e., kind* = *"selection"*) and E_k a set of files on which edit actions occurred (*i.e., kind* = *"edit"*). Similar to Lee *et al.* (2015) and for the sake of simplicity, we use the terms "edited files" to denote the files on which edit actions occurred and "viewed files" to denote files on which selection actions occurred. Figure 5.2 summarises the steps of a recommendation approach where a recommendation is made at each timepoint i (*i.e.,* when the developer triggers an event e_i on a program entity). At timepoint i, the context consists of the most recently viewed and edited files. A context is also considered as a query that is used to trigger a recommendation. After forming the context, we build the association rules in the database (traces that are already used for training, then for testing) to find traces that contain the viewed files V_k and the edited files E_k in their context. An association rule corresponds to a trace in the database, which is an implication in the form $Antecedent \Rightarrow Consequence$ where $Antecedent$ is a set of viewed files and $Consequence$ is a set of edited files. The edited files found in the association rules are considered candidates for recommendation. The support and confidence of the association rules are used to rank and select the association rules. The recommended candidates are ranked using their confidence. The support is the number of traces in the database containing both the antecedent and the consequence. The confidence is the support divided by the number of traces in the database containing the antecedent (Tan *et al.*, 2006). To evaluate a recommendation, the edited files in the trace that are not in the context are considered to be files that must be changed and they are compared to the recommended files.

The key point of the recommendation approach is the context used to trigger the recommendations. Three rules are used for context formation: (1) the conditional operation rule that aims to combine the context viewed and edited files using AND or OR operator; (2) the selection range rule that aims to improve the recommendation by fixing the number of events to be used to build the context. For example, the context can be built by considering any combination of three viewed files within the last 100 events; (3) the recommendation timepoint rule which defines when a recommendation is triggered (for example, a recommendation can be triggered after an edit action or after any kind of action).

5.3.2 Motivating Example

Consider two interaction traces T_1 and T_2. T_1 has the sequence of files $a-b-c-d-b-e-f$ where edit-events occurred on d and e. $T_1 = (V_1, E_1)$ with $V_1 = \{a,b,c,f\}$ and $E_1 = \{d,e\}$. T_2 is a sequence of files $b-a-f-h-b-e-d-g$ where edit-events occurred on h, e, and d. $T_2 = (V_2, E_2)$ with $V_2 = \{a,b,f\}$ and $E_2 = \{h,e,d\}$. When mining T_2, T_1 is used as training set. Consider that the recommendation timepoint is the moment when the developer interacts with f in T_2. The current context will be $\{b,a,f\}$ for viewed files and $\varnothing$ (empty set) for edited file. The recommendation approach proposed by Lee *et al.* (2015) looks at the training set (*i.e.,* $\{T_1\}$ with the rule $V_1 \Rightarrow E_1$) and finds the association rules whose the antecedent contains the actual context (*e.g.,* $V_1 \subset \{a,b,c,f\}$). Thus, the recommended candidates, $\{d,e\} = E_1$, are the consequence of the association rules $V_1\ RightarrowE_1$ found in the training set. The precision and recall at this timepoint are respectively 1 *i.e.,* $\frac{|\{d,e\} \cap \{h,e,d\}|}{|\{d,e\}|}$ and 2/3 *i.e.,* $\frac{|\{d,e\} \cap \{h,e,d\}|}{|\{h,e,d\}|}$ (See Section 5.3.3 for details about precision and recall). The precision and recall are computed at each timepoint and averaged according to the number of recommendations made.

The fact that some edit-events may be false edit-events impacts the recommendation and their precision and recall. Consider that T_1 and–or T_2 involved false edit-events. Table 5.3 shows precision and recall at timepoint f in T_2 when considering all combinations of false edit-events recorded for e. For example, the upper right cell shows precision and recall when the edit-event that occurred on e in T_1 is not a false edit-event but the edit-event that occurred on e in T_2 is a false edit-event. Table 5.3 reveals that when interaction traces contain false edit-events, the precision decreases from 1 (upper left cell) to 1/2 (upper right cell). The recall decreases from 2/3 (upper left cell) to 1/3. This example shows that the presence of false edit-events decreases the precision and recall of recommendations at timepoint i. It will also decrease the overall precision and recall, which motivated us

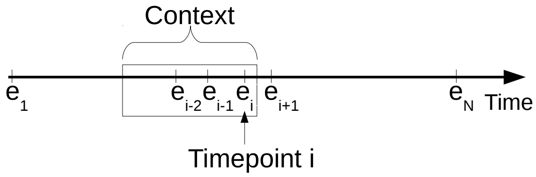


Figure 5.2 Illustration of Recommendation based on Interaction Traces

to investigate, in detail, the impact of false edit-events on the accuracy of recommendations.

5.3.3 Approach

Overview

We replicate the approach described in Section 5.3.1, which we use as ***baseline approach***. We clean the false edit-events using the threshold-based approach described in Section 4.3.1 and the prediction-based approach proposed in Section 4.3.2 and consider both as the ***approaches with correction***. Because the association rules in the baseline approach consider two kind of events (selection and edit), we consider in the threshold-based approach all events resulting from our annotation (*e.g.,* open and navigation) as selection events.

The baseline approach orders the traces with the goal of helping developers with recommendations using the traces collected in the past. However, the traces are ordered using the bug IDs (for which the traces were collected) and the trace IDs. We think that ordering the traces by their bug and trace IDs does not reflect the order in which the traces were collected. Therefore, we order the traces by using the start date of the first event that occurred in the trace. We also order the interaction events belonging to a trace by their start timestamps. Although, the baseline approach mines a trace regardless of the order of the events, we think that it makes sense to order the events to reflect the flow of activities performed by the developers. Thus, we ensure that at any timepoint *i*, the recommendations are made to suggest entities that should be edited after the timepoint *i*.

To make sure the additional changes (*i.e.,* the ordering of events and traces) do not affect our investigation of the effect of false edit actions, we analyse the results provided by each change individually before combining them together and comparing them to the baseline approach.

Subjects Systems

We use the Mylyn ITs collected and shared by the developers involved in Eclipse-based projects. We use the same dataset as the baseline approach (Lee *et al.*, 2015). The dataset involved 72 projects and contains 5,764 interaction traces. Mylyn, Platform, PDE, ECF, and MDT are major

Table 5.3 Precision and recall for all combinations of false edit-events recorded on e in T_1 and T_2

		T_2	
		True	**False**

T_1	**True**	P=1
		R=2/3
			P=1/2
			R=1/2

	False	P=1
		R=1/3
			P=1
			R=1/2

True=real edit-event
False=false edit-event
P=Precision R=Recall

systems with most interaction traces while the 67 other projects have an average of 20 interaction traces (Lee *et al.*, 2015). Table 5.4 shows the number of traces (#T), the number of viewed files per trace ($\frac{\#V}{\#T}$), the number of edited files per trace ($\frac{\#E}{\#T}$) for each system and approach. For example, for the Mylyn system that contains 2,726 traces, on average, 14 files are viewed and 3 files are edited per trace. After the correction of the traces, 14 files are still viewed and 2 files are edited per trace. The “Difference" column in Table 5.4 indicates the number and percentage of files wrongly considered as edited. For example, 40 of files per trace in the Mylyn project are wrongly considered as edited files.

Dependent and Independent Variables

We use the following variables to measure the accuracy of recommendations and early recommendations and to evaluate how the corrections of false edit actions affect the recommendations.

Independent variables: We consider the two approaches, *i.e.,* the ***baseline approach*** and the ***approaches with correction*** of interaction traces.

Dependent variables: At each recommendation point, we measure the accuracy using the set A of recommended files and the set E of expected files (*i.e.,* files that are edited after the recommendation point and that must be recommended). We measure the accuracy of the recommendations with the following metrics (Lee *et al.*, 2015):

- Precision: We compute the precision using the formula:

  $P = \frac{|A \cap E|}{|A|}$

  The higher precision, the lower is the number of false positives, *i.e.,* more recommended files are correct.

- Recall: We compute the recall using the formula:

Table 5.4 Subject Systems

	#T	Baseline		Correction		Difference
		$\frac{\#V}{\#T}$	$\frac{\#E}{\#T}$	$\frac{\#V}{\#T}$	$\frac{\#E}{\#T}$	
Mylyn	2,726	14.39	2.75	14.40	1.64	1.10 (40%)
Platform	582	24.23	2.67	24.23	1.55	1.12 (42%)
PDE	536	6.36	1.07	6.36	0.45	0.62 (58%)
ECF	308	9.30	0.72	9.30	0.33	0.39 (54%)
MDT	245	54.48	0.71	54.48	0.39	0.32 (44%)
Others (67)	1,367	22.81	1.41	22.82	0.78	0.63 (44%)

#T = number of Traces
#V = number of viewed files
#E = number of edited files

$R = \frac{|A \cap E|}{|E|}$

A high recall means that a high proportion of files that will be edited is recommended.

- F-measure: We use the F-measure as a metric to measure accuracy combining both precision and recall. We compute the F-Measure using the formula:

  F-measure $= \frac{2*P*R}{P+R}$

  The higher F-measure, the more accurate is the recommendation.

- Feedback: The Feedback measures the efficiency of the recommendations (Zimmermann *et al.*, 2005). Feedback is computed as the number of recommendations divided by the number of queries (*i.e.,* context). We compute the feedback using the formula:

  $Fb = \frac{\#Recommendations}{\#Query}$.

  where $\#Query$ is the number of query (*i.e.,* number of contexts) and $\#Recommendations$ is the number of recommendations made by the system.

  The higher feedback, the more efficient is the recommendation, according to the number of query.

We define the early recommendation point (Nth) by counting the number of actions that occurred before the first recommendation and averaging the counts over the number of recommendations made. Lee *et al.* (2015) used the same metric to measure the early recommendations:

$Nth = \frac{\sum \#Action\_to\_Recommendations}{\#Recommendations}$

The lower is the early recommendation point, the quicker the system provides recommendations.

To evaluate the early recommendation, we look at the first recommendation point of the baseline wrt. the first recommendation point of the approaches with correction. We consider that the corrections of interaction traces enable early recommendations if the value of the early recommendation point after correcting interaction traces data is lower than the value of the first recommendation point obtained with the baseline approach.

Analysis Method

To assess whether the correction of interaction traces affect the accuracy of recommendations, we first simulate the recommendations without any corrections and compute the metrics defined above. Then, we correct the traces, simulate the recommendations again, and recompute the metrics. We compare the metric values obtained for the baseline approach with those obtained for the approaches with corrected traces. We use the Mann-Whitney Wilcoxon test using a 95% confidence level (*i.e.,* we conclude that the difference in accuracy is statistically significant only if p-value $<$ 0.05). We use the Mann-Whitney test because it is a non parametric test that does not make any assumption about the data distribution. In case of differences between the accuracies of the baseline approach and of the approaches with correction of interaction traces, we measure the magnitude of the difference using the Cliff's d non-parametric effect size. According to J. Romano (2006), we consider that the effect size is negligible if $|d| < 0.147$, small if $|d| < 0.33$, medium if $|d| < 0.474$, and large if $|d| >= 0.474$.

5.3.4 Results and Discussions

Recommendation Accuracy

Table 5.5 shows the results obtained with the baseline approach. These results (Table 5.5) include the precision (P), recall (R), Feedback (Fb), number of recommendations (#Rs) and query (#Qs) for each system. Compared to the results with corrections using threshold-based approach (See Table 5.6a), we observe that the precision and recall increased after the correction of the traces, except for the precision of Platform (which decreased by 1%), the recall of PDE (which decreased by 2%), and the precision of MDT (which remained constant at 100%). The precision and recall for the Mylyn system increased from 69% to 75% and from 55% to 64%, respectively. Similarly, the prediction-based approach improves the baseline approach (See Table 5.6b). The prediction-based approach also improves the recall up to 24% more than the threshold-based approach (See columns ”R“ (Recall) in Table 5.6a vs. Table 5.6b). The results of ECF system for prediction-based approach show that the correction of traces results in some traces without any edit events. Thus, the number of expected files is zero (See the formula of precision and recall in Section 5.3.3). This is a threat reported in the threat to validity section (See Section 5.4). However, we think that is some cases developers may provide some traces where they only explore source code without any changes and perform their changes later in a separate trace. This may also happen for example if developers have a quick refactoring session that last in few seconds.

Because the F-measure combines and represents the tradeoff between precision and recall, we perform the Mann-Whitney Wilcoxon test on F-measure. The statistical test confirms that the observed differences in the accuracies of the recommendations are statistically significant, except for the PDE system using the threshold-based approach (See Table 5.7).

Moreover, using the threshold-based approach for trace correction, the effect size of the difference is large for ECF, MDT, and Other systems, the effect size is medium for Platform system, and small for Mylyn system. Figure 5.3 illustrates the magnitude of the differences in F-measure. The prediction-based approach for trace corrections shows the same trend as the threshold-based approach (See “Prediction-based” columns in Table 5.7). From these results, we conclude that correcting false edit actions in interaction traces data can improve the accuracy of recommendation systems.

Beside the improvement of the recommendations accuracies, the improvements in feedback show that the corrections of interaction traces provide more recommendations than the baseline approach. For example, in the case of the Platform system, the baseline approach provides about 22 recommendations out of 100 queries (Feedback = 0.22 in Table 5.5) while the corrections of the traces yield about 35 and 38 recommendations out of 100 queries, respectively using the threshold- and prediction-based approach (Feedback = 0.35 in Table 5.6a and Feedback = 0.38 in Table 5.6b). The improvement of feedback for all the systems shows that the corrections of traces decrease the number of queries and–or increase the number of recommendations. While the number of recommendations did not increased for all the systems in our experiment, the number of query decreased for all the systems.

The corrections of traces reduce the numbers of edit actions and, thus, the numbers of recommendation timepoints (*i.e.,* the number of queries). The decrease of the numbers of queries adds value to our approaches with corrections of the traces because it is an indication of the performance of the trace correction. Regardless of the simulation of the recommendations, if the implementation is integrated into a tool, the greater the number of queries, the more the tool will be slow because each query may result in the mining of the training set (interaction traces database) to provide a recommendation. Thus, we argue that in addition to more accurate recommendations, the correction of traces may be faster if integrated in a development tool. However, this hypothesis should be addressed in our future work.

Table 5.5 Recommendations Accuracy for the Baseline Approach

	Baseline Approach				
	P	**R**	**Fb**	**#Rs**	**#Qs**
Mylyn	0.69	0.55	0.24	8,920	36,563
Platform	0.88	0.39	0.22	2,722	12,020
PDE	0.49	0.73	0.07	176	2,240
ECF	0.30	0.72	0.04	43	966
MDT	1.0	0.29	0.09	194	2,041
Others	0.77	0.36	0.11	1,817	16,356

Table 5.6 Recommendations Accuracy for the Approach with Correction

	P	**R**	**Fb**	**#Rs**	**#Qs**
Mylyn	0.75 ↑	0.64 ↑	0.32 ↑	8,104	25,156
Platform	0.87 ↓	0.54 ↑	0.35 ↑	2,861	8,099
PDE	0.78 ↑	0.71 ↓	0.15 ↑	196	1,271
ECF	0.81 ↑	0.95 ↑	0.11 ↑	80	705
MDT	1.0 →	0.62 ↑	0.12 ↑	211	1,742
Others	0.79 ↑	0.60 ↑	0.16 ↑	1,559	9,441

(a) Threshold-based Approach

	P	**R**	**Fb**	**#Rs**	**#Qs**
Mylyn	0.73 ↑	0.67 ↑	0.27 ↑	5,153	18,838
Platform	0.81 ↓	0.70 ↑	0.38 ↑	2,239	5,883
PDE	0.86 ↑	0.85 ↑	0.10 ↑	94	935
ECF	0.00	NaN	0.00	2	484
MDT	1.0 →	0.86 ↑	0.14 ↑	180	1,218
Others	0.73 ↓	0.64 ↑	0.15 ↑	1,027	6,779

(b) Prediction-based Approach

Early Recommendations

In Table 5.8, we present the results of early recommendations for both the baseline approach and the approaches with correction of interaction traces. For each system, we show the F-Measure (F), the early recommendation point (Nth), and the numbers of first recommendations (#FR). The F-measures for early recommendations are different from the ones of recommendations accuracy because we use different context formation and different rules for recommendation point. For example, a recommendation is made at each edit timepoint for the simulation of recommendation accuracy, while a recommendation is made at edit and view timepoints for the simulation of early recommendations (Lee *et al.*, 2015).

The results of early recommendations do not show the same trend for all systems (See Table 5.8). These results however show that, after correcting interaction traces data with the threshold-based approach, recommendations are made earlier for Platform (26th vs. 46th timepoint) and Others systems (29th vs. 40th timepoint). On the contrary, the baseline approach makes recommendations earlier for ECF (78th vs. 47th timepoint) and MDT (56th vs. 17th timepoint) systems. The early recommendation points are almost the same for Mylyn and PDE systems (the baseline approach recommended at only one or two timepoints before the recommendation made after correcting with the threshold-based approach). Figure 5.4 summarises early recommendation results, highlighting whether early recommendations also provides accurate recommendations (F-measure). Figure 5.4

Table 5.7 Difference in Accuracy (F-measure) Between the Baseline Approach and Correction

	Threshold-based		Prediction-based	
	p-value	**Cliff $\|d\|$**	**p-value**	**Cliff $\|d\|$**
Mylyn	$<$ 2.2e-16	0.15	$<$ 2.2e-16	0.16
Platform	$<$ 2.2e-16	0.40	$<$ 2.2e-16	0.63
PDE	0.30	N/A	0.001	0.21
ECF	1.8e-12	0.74	NA	NA
MDT	$<$ 2.2e-16	0.99	$<$ 2.2e-16	0.99
Others	$<$ 2.2e-16	0.49	$<$ 2.2e-16	0.47

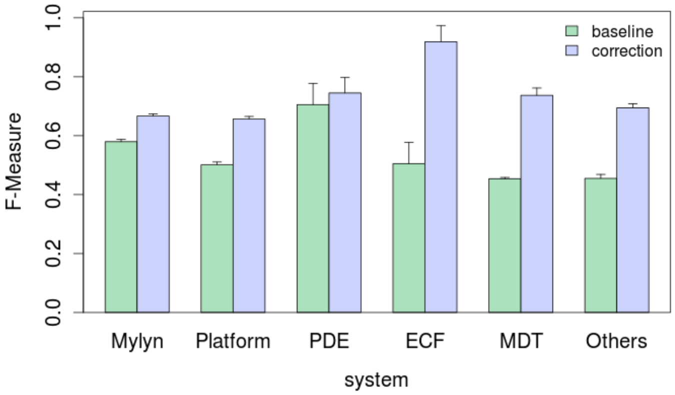


Figure 5.3 Difference in F-Measure Between the Baseline Approach and Threshold-based Approach

shows that regardless of whether ITs are corrected or not, early recommendations are not always accurate. As shown in Figure 5.4, the baseline approach provides recommendations earlier than the threshold-based approach, for ECF and PDE systems. However, the accuracies of theses recommendations are lower than those of the recommendations made after correcting ITs with the threshold-based approach.

Using the prediction-based approach for trace correction shows the same trend as the threshold-based approach, except for the F-measure of Mylyn and Others systems. Thus, we conclude that correcting false edit actions in interaction traces data does not guarantee better early recommendations.

We explain the lack of improvement on early recommendations by the fact that the corrections of interaction traces decrease the numbers of edit actions in the traces. Intuitively, during an evolution task, developers most likely explore the program when they start working on the task (*i.e.,* at the beginning of the trace) to search for relevant entities and perform changes after finding the relevant

Table 5.8 Results for Early Recommendations for the Baseline Approach and the Approach with Correction of Interaction Traces

	Baseline			Threshold-based			Prediction-based		
	F	**N-th**	**#FRs**	**F**	**N-th**	**#FRs**	**F**	**N-th**	**#FRs**
Mylyn	0.17	14 th	1,349	0.17 →	16 th ↑	1,166 ↓	0.16 ↓	19 th ↑	1,088 ↓
Platform	0.19	46 th	177	0.21 ↑	26 th ↓	148↓	0.20 ↑	27 th ↓	127 ↓
PDE	0.16	11 th	113	0.11 ↓	12 th ↑	68 ↓	0.08 ↓	16 th ↑	47 ↓
ECF	0.07	47 th	49	0.08 ↑	78 th ↑	22 ↓	0.18 ↑	119 th ↑	16 ↓
MDT	0.08	17 th	15	0.16 ↑	56 th ↑	14 ↓	0.22 ↑	71 th ↑	13 ↓
Others	0.20	40 th	177	0.19 ↓	29 th ↓	135 ↓	0.21 ↑	36 th ↓	106 ↓

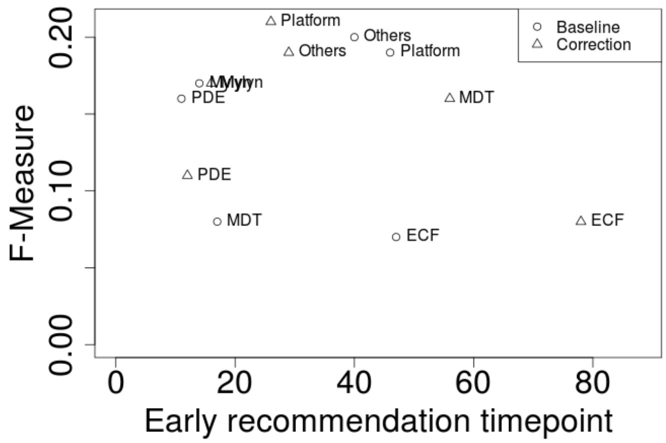


Figure 5.4 Comparison of the early recommendation of the Baseline Approach and the Approach with Correction of Interaction Traces

entities (*i.e.,* usually towards the end of the trace). Hence, as discussed in Section 4.3.1, false edit actions mostly occur at the beginning of the interaction traces (when the developer is searching for relevant entities to modify). Thus, the correction of false edit actions early at the beginning of the trace is likely to impact the generated association rules because these association rules may not contain enough consequences to trigger recommendations earlier. Nevertheless, Figure 5.4 shows that early recommendations that are based on false edit-events are usually inaccurate.

5.4 Summary

We studied the impact of noise on the identification of developers' editing style and the accuracy of recommendation systems. By cleaning ITs, we report that noise may have led researchers to mislabel some participants' editing styles in about 41% of the ITs. Cleaning false edit-events significantly improves the accuracy of the recommendations by up to 51% for precision and up to

57% for recall. Our work may suffer from the accuracy of our ITs cleaning approaches described in Chapter 4.

Our results show that noise impacts previous studies. To avoid biased results, we recommend to researchers and practitioners to clean noise in ITs prior to their usage or to evaluate if noise in ITs would impact their work. Consequently, when studying how developers spend their effort during maintenance and evolution in the next chapter, we evaluate whether noise would impact our results by using both raw ITs and cleaned ITs.

CHAPTER 6 DEVELOPERS' EFFORT DURING MAINTENANCE TASKS

6.1 Introduction

When performing a maintenance task, developers spend a certain effort exploring the program, finding relevant entities, understanding and making changes to these entities (Ko *et al.*, 2006). The cost of each of these developers' activities has a direct impact on the overall cost of maintenance.

Despite the large body of work on software productivity (Banker *et al.*, 1987; Boehm, 1987; Maxwell et Forselius, 2000), there are very few studies that empirically investigated how developers spend their effort during software maintenance. The relation between the severity of a task, the complexity of the implementation required for a task, and the effort required to understand and perform the task has yet to be studied in details.

In this chapter, we analyse developers' ITs from four open-source projects, ECF, Mylyn, PDE, Platform. We aim to (1) understand how developers spend their effort when finding the solution to a task and (2) identify some of the factors affecting developers' effort. In fact, practice and common sense show that developers are not equal when facing a software maintenance task. Some developers perform their tasks quickly, spending less effort, while others perform their tasks slowly and, worse, with more effort. If these differences are due to factors which can be influenced through tooling, then we can identify these factors and propose such tooling. Of course, we expect that developers' differences are partly due to individual differences and partly due to tooling.

To achieve the aforementioned goal, we investigate the following research questions:

RQ1: *Does the Complexity of the Implementation of a Task Reflect Developer's Effort?*

The effort spent by some developers can be disproportionate to their results. We measure developers' effort with the time spend performing the task and the Cyclomatic complexity of their exploration graph (*i.e.,* how they move from a program entity to another). We consider the changes in a patch (*i.e.,* the implementation of a task) as the result of a developers' effort. We match interaction trace logs with patches (*i.e.,* identify the patch that is the implementation of a given interaction trace). Finally, we correlate the effort to the complexity of the implementation.

RQ2: *How do Developers Spend their Effort?*

We use the similarity between the matched interactions and patches to identify the number of additional files used by developers. We find that when performing a task, developers use on average about 62% of additional files, and that developers spend part of their effort exploring additional files. Then we correlate the additional files to the effort to assess whether effort is related to the use of additional files.

6.2 Data Collection and Processing

We download 2,408 developers' interaction histories and 3,395 patches from four open-source software projects (ECF, Mylyn, PDE, Eclipse Platform).

6.2.1 Interaction Traces

We parse the interactions to extract the program entities on which the events occurred and the time spent on each entity. A program entity can be a Java entity (file, class, field, method) or a resource (other project entities). For Java entities, we consider that all the actions that occurred on an entity (class, field, method) in a Java file are the actions on the Java file. We aggregate the actions at file level because we match the interaction data with patch data that contains only changes at file level. We use the word "file" to name the Java files and resource entities involved in an interaction. Table 6.1 presents the number of interactions per project. Without distinguishing among projects, an interaction involves on average 46.87 files (standard deviation 220.05).

6.2.2 Patches

A patch can involve many files. For each file involved in a patch, the changes made in the file can be grouped into many *deltas*. A *delta* contains a snippet code before (old code) and after (new code) the change, respectively called *original chunk* and *revised chunk*. We use the DiffUtils[1] library to parse the patches and extract the files involved in the patch and the changes for each file. The DiffUtils library is a Java open-source library for applying patches, generating unified diffs, or parsing them. We adapt the parsing feature to recover the qualified name (package and file names) of each modified file and the changes performed in these files.

We observe that 26 attachments are not in the patch unified diff format (*i.e.,* we are not able to distinguish the source code before and after the changes) and 11 attachments did not contain the modifications date of the files. Yet we need both the modification dates in the patch and the patches in unified diff format to match them with the interactions, and to compute some patch metrics. Therefore, we remove these attachments from the patch data. We finally retain 3,395 patches as shown in Table 6.1. Without distinguishing among projects, a patch involves 6.32 files (standard

[1] http://code.google.com/p/java-diff-utils/ (visited 20/10/2015)

Table 6.1 Number of interactions and patches

	# Interaction	# Patch	Total attachment
ECF	60	83	143
Mylyn	1,644	1,631	3,275
PDE	373	683	1,056
Platform	331	998	1,329
Total	2,408	3,395	5,803

deviation = 18.57).

6.2.3 Interaction and Patch

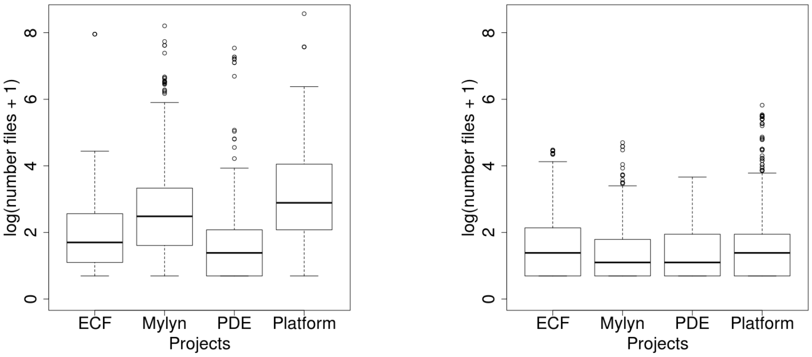
In general, the number of files involved in interactions and patches (the mean is 46.87 for interaction vs. 6.32 for patch) indicates that developers use/explore more files (in the interaction) than they modify (in the patch) as expected. Figure 6.1 presents the comparison between the files involved in the interactions (Figure 6.1a) and patches (Figure 6.1b) for each project. We plot the logarithm of the number of files to make plots readable. The explored files (in the interaction) that are not modified (in the patch) can be seen as (1) files that are useful to understand the program or (2) accidental files that indicate some disorientation when developers are looking for the files that must be modified. We suspect that the exploration of these additional files may affect developers' effort.

6.3 Does the Complexity of the Implementation of a Task Reflect Developer's Effort?

Developers perform many change requests daily. To implement a change request, a developer must change the file(s). In this research question, we want to verify if the complexity of the implementation of a change request usually reflects the effort spent by developers when performing the task.

6.3.1 Motivation

Although one expects that complex implementations would required more effort from developers, sometimes, a simple implementation can also require a lot of effort from a developer. In general, developers are not equal when facing a software maintenance task. Even if some developers are not always efficient (depending on the task and project), they may mostly perform their tasks quickly, spending less effort, while others may mostly perform their tasks slowly and, worse, with more effort. By understanding if this difference in performance between developers is related to the complexity of tasks, software organizations would be able to better assign tasks to their developers in order to improve the overall productivity of their development teams.



(a) Files in interactions

(b) Files in patches

Figure 6.1 Distribution of logarithm of the number of files involved in the interactions and patches

To study whether the complexity of the implementation required by a change request reflects developer's effort, we compute a set of metrics to measure developers' effort and the complexity of the implementation of tasks (Section 6.3.2). We discuss the approach in Section 6.3.3 and Section 6.3.4 discusses the obtained results.

6.3.2 Metrics

Developers' Effort

We use the interaction data and compute two metrics to assess a developers' effort.

- Time: The time spent when performing a task. We consider the effort as the accumulated time defined in Section 4.2.3 and we remove the idle and overlap times. We assume that the more time developers take to understand and perform changes, the more they spent effort.

- Cyclomatic complexity: The Cyclomatic complexity of a developer's interaction is a bridge to assess his effort. We use the cyclomatic complexity to quantify the complexity of the developer's interaction. Thus, we consider an interaction as an *exploration graph i.e.,* a graph in which a node is a file and an edge is an exploration from one file to another. Because developers can move back and forth between files when exploring a program, an exploration graph is a directed pseudograph, *i.e.,* both graph loops and multiple edges are permitted[2]. We use the JGraphT[3] library to compute an interaction as a directed pseudograph. The Cyclomatic complexity is defined by the following formula: $Cyclomatic = m - n + k$

  where m is the number of edges, n is the number of vertex, and k is the number of connected components. In an exploration graph, the number of connected components is one ($k = 1$)

[2] http://mathworld.wolfram.com/Pseudograph.html (visited 20/10/2015)

[3] http://jgrapht.org/ (visited 20/10/2015)

because the files involved in the interaction are explored/connected one to another *i.e.,* when the developers move from one node (file), they always go to another node (file). We think that the more is the Cyclomatic complexity of the exploration graph, the more is the complexity of the interaction.

The Spearman correlation coeficient between the two metrics are 0.71, 0.59, 0.62, and 0.69 for ECF, Mylyn, PDE, and Platform systems, respectively. These correlations show that the two metrics are not strongly correlated. Thus, they may measure different aspects of developers' effort.

Complexity of Developers' Implementations

We use the patches to compute two metrics that measure the complexity of the implementation of the tasks:

- Entropy: The patch entropy measures how much the changes are scarttered/expanded between files. We use the number of files involved in a patch and the number of inserted and deleted lines of code per file to compute the Shannon entropy of the patch. The Shannon entropy is defined by:

  $H_n(P) = -\sum_{k=0}^{n}(p_k * \log_n p_k)$

  where P is a patch; $p_k \geq 0, \forall k \in 1,2,...,n$ and $\sum_{k=0}^{n} p_k = 1$. In the formula, n is the number of files involved in the patch and p_k is the probability of the file k to be modified *i.e.,* the number of modified lines of code in the file k divided by the total number of modified lines of code in all files involved in the patch. When all the files have the same probability to be modified ($p_k = \frac{1}{n}, \forall k \in 1,2,...,n$), there is a maximum entropy. When only one file i is modified ($p_i = 1$), there is minimal entropy. The higher the entropy, the more the changes are scattered between files.

- Change distance: Change distance measures how much is the difference between the old source code (before the change) and the new source code (after the change). We define the change distance as the Levenshtein distance between the old source code and the new source code. As the changes on a file can be grouped into *deltas*, we avoid the mismatch mapping between old code and new code by computing the change distance for each *delta*. To address the confounding effect of the length of the old and new source code, we normalize the (*delta*'s) change distance between zero and one by dividing the distance by the maximum length of old and new source code in the *delta*. The value zero of the normalized distance means that there is no change (*i.e.,* old code = new code) and the value one means that there is a complete difference. We define the change distance of a file as the mean of the change distance for all the file's *deltas*. Then we define the change distance of a patch as the mean of the change distance for all the files involved in the patch. The greater is the difference between the old and the new code (*i.e.,* more change distance), the more the change is complex.

When computing the complexity of implementations, we remove the blank lines and keep the comments. We think that the comments in the patch cannot affect the matching between developers' effort and their implementations. In fact, when developers comment their code, they spend some time. However, when we follow the approach explained in Section 6.3.3 using the data with comments and without comments, the removal of the comments did not affect our results.

There is not a strong correlation between the entropy and the change distance. The Spearman correlation coefficient are 0.59, 0.24, 0.17, and 0.35 for ECF, Mylyn, PDE, and Platform, respectively. This may indicate that the two metrics do not measure the same aspect of the complexity of the implementation.

6.3.3 Approach

To answer our research question (Does the Complexity of the Implementation of a Task Reflect Developer's Effort?), we process in two steps. First, we match/link the interactions to the patches. Second, we assess the relation between developers' effort (extracted from interactions) and the complexity of the implementations (extracted from patches).

Interaction and Patch Matching: A bug report sometimes has only interactions, only patches, or both. In the following, we consider a bug report to which are attached a set of interactions $\mathcal{I}$ ($|\mathcal{I}| = m$ is the number of attached interactions) and a set of patches $\mathcal{P}$ ($|\mathcal{P}| = n$ is the number of attached patches). We can have $m \leq n$ or vice versa, $m = 0$ and–or $n = 0$. The matching consists, for each interaction $I \in \mathcal{I}$, to find the patch $P \in \mathcal{P}$ that is the result of the interaction I. The matching is possible if and only if $m \neq 0$ and $n \neq 0$ (even if $m = n$ or $m \neq n$). An interaction attached to a bug report (for a given project) cannot be matched to a patch attached to another bug report (or another project). Therefore, we look at the possible matching for each pair of interaction/patch attached to the same bug report. For the bug report considered above, we should have $m \times n$ interaction/patch pairs to investigate. Since multiple developers can attach interactions and patches to the same bug report, we reduce the number of interaction/patch combination by considering only the interaction(s) and the patch(es) attached by the same developer. We also use the attachment date (in the Bugzilla's date format *i.e.,* date, hour, minute, and timezone) to match the interactions and patches. In fact, an interaction or a patch is attached to a bug report at a specific date. We assume that *an interaction is matched to a patch (i.e., the patch is the result of the corresponding interaction) if and only if both are attached to the same bug report, by the same developer at the same date and time*.

Using the above criteria to match interactions to patches, we face the *unbalanced matching* problem. An unbalanced matching is when developers modify files without interacting with them *e.g.,* through refactoring. In fact, a refactoring does not require much effort (in the interaction) to propagate changes, but the propagation of the changes is materialized in the patch. We use the number of files involved in both the interaction and the patch to avoid *unbalanced matchings*.

Consider that we use the above criteria to match the interaction I with the patch P. The notation $f \in I$ means that the interaction I involves the file f. The same notation is used for the patch ($f \in P$). An unbalanced matching appears in the following cases:

- $P \not\subseteq I$ *i.e.,* $\exists f \in P, f \notin I$: There are files in the patch that were not involved in the developer's interaction. This situation may be due to refactorings performed by the developer, or to changes that were not collected by the Mylyn Plugin (*e.g.,* changes performed between interruption periods when the Mylyn Plugin was inactive).
- $|P| \cap |I| = \varnothing$: There are no common files between the interaction and the patch. This situation may be caused by developers performing the changes appearing in the patch, when the Mylyn Plugin was inactive.

To match an interaction to a patch and avoid unbalanced matching, we can also use the working dates in the interaction and the modifications dates in the patch. With these dates, we can make sure to consider only patches that were created after the developer completed the task. However, some developers may collect their interactions when performing the task. But they can create the patch (containing all the performed changes) before disabling the task or vice versa (*i.e.,* they disable the task before they create the patch). In both cases, the developers perform the task while collecting their interaction. A few differences will occur between the end of interaction date and the modification date in the patch. Therefore, we cannot automatically use the working dates and modifications dates to match interactions to patches. We use a sample of interaction/patch matchings to manually validate the relation between the working dates and the modifications dates. We choose our sample size to achieve a $95\% \pm 10$ confidence level. The sample was proportionally distributed among projects, except for ECF project where instead of two interaction/patch matchings (due to the small number of matchings), we used half of all the ECF matchings.

Unbalanced matchings are the matchings where the patch is not (or is the part of) the result of the corresponding interaction. Because we want to compare developers' effort (interaction) and the complexity of the implementation of tasks (patch), we remove the unbalanced matchings from our data before performing the comparison.

Developers' effort vs. Complexity of the implementation: After the matching between the interactions and the patches, we examine the correlation between the metrics that measure the effort (time spent and cyclomatic complexity) and those that measure developers' implementations (entropy and change distance). We use the Spearman correlation coefficient because it is a non-parametric test that does not make assumptions about the distribution of our metrics.

6.3.4 Results and Discussions

The number of raw interaction/patch pairs (raw combination of interactions and patches of the same developer) and the number of the matching pairs are shown in Table 6.2. The number of raw interaction/patch pairs is less than the number of interaction multiplied by the number of patches because we made the combination interaction-patch only for each bug report and each developer. The column "#Matching" shows the number of matchings with the number of unbalanced matchings in the parenthesis. Overall, we removed the unbalanced matchings (217 in total) and retained 1,028 interaction/patch matchings. Cases where a developer attach one interaction along with many patches and vice versa did not appear in our dataset *i.e.,* the number of matching pairs (1,028) is equal to the number of interactions and the number of patches involved in the matchings.

We examined cases of unbalanced matchings to understand developers' habits when working with interactions and patches. We observed that unbalanced matchings occur when a developer gather the interactions (*i.e.,* the working task is activated), after finding where and how to perform the task, and stops collecting interactions (*i.e.,* the working task is disabled) before performing the changes. For example, this practice is observed on the Platform's bug #263816 when the developer "qualidafial" tried to fix a null pointer exception in the class "ObservableSetContentProvider.java"; there is an unbalanced matching between the interaction #124830 and the patch #124829 *i.e.,* their intersection is empty. The same practice where $P \nsubseteq I$ is observed on the ECF's bug #194975; there is an unbalanced matching between the interaction #96731 (2 files) and the patch #967330 (3 files). The file involved in the patch that is not in the interaction is a "property" file.

We observe from the manual validation of matchings that developers mostly disable the task before creating the patch (73.95% of matchings) vs. 26.04% where the patch was created before the task was disabled. However, in most cases the time difference between these two actions were just a few seconds, with the maximum being 12 minutes. The sample size was 96 matchings (ECF: 8, Mylyn: 57, PDE: 12, and Platform: 19). While this time difference may affect our results, we believe that in most cases, developers can not disable the task and create its patch at the same time.

Table 6.2 Number of interaction/patch pairs and matching

	Raw interaction/patch pairs			#Matching
	#Interaction	#Patch	#Pairs	
ECF	31	38	53	17 (2)
Mylyn	946	1,123	2,634	785 (122)
PDE	212	272	397	159 (27)
Platform	314	599	2,331	284 (66)
Total	1,503	2,032	5,415	1,245 (217)

Table 6.3 shows that for all combinations of effort/implementation metrics, the effort spent by developers when performing a task is not strongly correlated to the complexity of the implementation of the task. This result means that **developers do not necessary spend more effort on tasks requiring more complex implementations**.

6.3.5 Sensitivity Analysis

We analyse the sensitivity of our results to noise. Noise can only impact our results on the correlation between time and complexity of implementation *i.e.,* results in Table 6.3a. Thus, we compute the time by adding the sum of the idle time that are less than half a minute as reported in Section 4.2.3.

Table 6.4 shows the correlation between time and complexity metrics (entropy and change distance) using CITs. We observe the same trend as with RITs (see Table 6.3a). We did not look at other correlation using CITs as they are not affected by the noise found in Chapter 4. We can conclude that the time-related noise does not affect our study on the relation between developers' effort and complexity of implementation *i.e.,* our results are robust to the time-related noise.

The lack of strong correlation between the effort spent on a task and the complexity of the implementation of that task also suggest that some of the effort spent by developers on a task does not materialise in the patch of the task. For example, a developer can spend time exploring some files that should not be modified for a given task, but which are useful to understand the program and perform the required changes on other files. In the next section, we examine this phenomenon in more details. More specifically, we investigate how developers spend their effort and the factors affecting developers' effort.

Table 6.3 Spearman correlation between the developers' effort and the complexity of the implementation

	Entropy	Change Distance
ECF	-0.05	0.36
Mylyn	0.17	0.21
PDE	0.22	0.30
Platform	0.06	0.27
All	0.16	0.27

(a) Time

	Entropy	Change Distance
ECF	0.25	0.42
Mylyn	0.26	0.27
PDE	0.46	0.35
Platform	0.27	0.36
All	0.31	0.33

(b) Cyclomatic complexity

Table 6.4 Spearman correlation between the developers' effort and the complexity of the implementation using CITs

	Entropy	Change Distance
ECF	0.009	0.34
Mylyn	0.16	0.21
PDE	0.29	0.30
Platform	0.15	0.32
All	0.18	0.28

6.4 How do Developers Spend their Effort?

This research question aims to (1) assess how developers spend their effort *i.e.,* how much they use additional files and (2) study the factors that affect developers' effort.

6.4.1 Motivation

When performing a maintenance task, developers must navigate through the program entities (*i.e.,* methods, class, files, etc.). They must know where and how to perform the changes on these entities to address the task. However, developers sometimes explore files that are not significantly relevant to the task. This use of additional files may increase the developers' effort. These additional files can also mislead the developer; making him perform the wrong changes and introduce bugs. On the positive side, additional files can help better understand the context of a task and identify program entities that should be modified to complete the task. The number of additional files explored by developers may depend on the severity of the task and developers' experience and knowledge about the program. Bug severity indicates the impact the bug has on the successful execution of the software system (Lamkanfi *et al.*, 2010). It measures how much a bug can affect the performance and stability of the system, or the (percentage of) developers that can be affected by the bug.

A high severity typically represents fatal errors and crashes (Lamkanfi *et al.*, 2010). Therefore developers may be careful when fixing severe bugs. We think that, when fixing severe bugs wrt. less severe bugs, developers may spend more effort (1) to make sure that they are performing the right change, and (2) to ensure that they are not introducing new bugs i.e., by revalidating their changes because revalidation is one of the three main activities (understanding, modifying and revalidating) involved in software maintenance (Boehm, 1976a). To find significantly relevant files, developers may need to explore some additional files. Developers who have more experience and knowledge of the project may be able to find significantly relevant files while those with less experience and knowledge (about the project) may guess when looking for significantly relevant files. We hypothesize that the more developers use additional files, the more they spend effort.

By knowing how much efforts developers spend on additional files, software organisations could make use of recommendation systems to reduce the amount of additional files explored by their developers *i.e.,* by guiding them during the exploration of the program and improve their productivity.

6.4.2 Metrics

To study how developers spend their effort, we consider the similarity between the explored files and the significantly relevant files. We use the Jaccard similarity coefficient as similarity measure. The Jaccard similarity between the matched interaction I and patch P is:

$Jaccard(I,P) = \frac{|I \cap P|}{|I \cup P|}$

where $|I \cap P|$ is the number of file involved in both the interaction I and the patch P, and $|I \cup P|$ is the total number of files involved in I and P.

As we remove the unbalanced matchings in the matching dataset, it means $P \subseteq I$ *i.e.,* $I \cap P = P$ and $I \cup P = I$. The Jaccard similarity shows the degree of the use of additional files. The set of additional files is $I \backslash P$ *i.e.,* $f \in I$ and $f \notin P$. We believe that the more developers use additional files, the more they spend the effort.

Beside the use of additional files, developers' effort may depend on the severity of the task and–or developers' experience. In fact, Panger (Panjer, 2007) found that bug severity is an important variable to predict bug lifetimes (from the time of confirmation to resolution) *i.e.,* the resolution time of severe bugs is greater than the resolution time of less severe bugs. Thus, because developers may be careful when fixing severe tasks, the effort spent to perform less severe tasks must be different to the effort spent to perform more severe tasks. Similarly, it is expected that an experienced developer would spend less effort compared to inexperienced developers. We use the following metrics to assess developers' experience:

- The number of bug (NB) fixed before;
- The number of files (NF) modified before;
- The number of lines of code (NLOC) inserted and deleted before (sum of inserted and deleted LOC).

We use the patch to measure these developers' experience (NF and NLOC). We use the patch because when mining the source code repositories of our subjects projects (to capture NLOC for example), we observed that some developers who attached the patches were not found as authors in the source code repository. These developers probably lack commit privileges and therefore submit their contributions to more seasoned developers acting as reviewers. This phenomenon have been observed in many open-source projects. For example the ECF bug #199366, the interaction #76609 and the patch #76608 are matched. The attacher was not found in the code repository and another developer (probably the one who reviewed the patch) congratulated him in the bug report by saying: "*Fixed. Thanks Abner for the patch. IP log updated*". Thus, we were not able to mine developers' experience and knowledge through the source code repositories. Because we want to study the effect of the experience on developers' effort, for a given interaction/patch matching, we must consider developers' experience before they attach their interactions/patches. Therefore, we consider developers' experience before the interaction and the patch attachment date (interaction and patch have the same attachment date since they are matched). Instead of using only the matching dataset, we use all the data (See Table 6.1) to compute developers' experience in order to avoid missing parts of some developers' experience.

For some tasks, the experience of a developer may not be helpful *e.g.,* when the task does not need the files that were used before. The experience is more helpful when the significantly relevant files for a given task have already been used in previous tasks. For example, consider a developer D performing a first task T_1 by making changes on files f_1 (two LOC) and f_2 (five LOC). Because T_1 is the first task of D, the experience before performing T_1 is 0 (0 task, 0 files, and 0 LOC before). Suppose that D have a new task T_2 to perform. Before performing T_2, D had experiences on f_1 and f_2 (one task, two files and seven LOC). We have two scenario: (1) The significantly relevant files for T_2 are f_2, f_3, and f_4. According to these significantly relevant files that are needed to perform T_2, the "relevant" experience of D is on f_2 because f_2 was already modified when performing T_1 and f_2 is significantly relevant to T_2; (2) The significantly relevant files for T_2 are f_3, f_4, and f_5. According to these significantly relevant files needed to perform T_2, D may have no "relevant" experience before performing T_2 if f_1 or f_2 are not used in f_3, f_4, or f_5. On the contrary, D may have experience before performing T_2 if f_1 or f_2 are used in f_3, f_4, or f_5. However, our dataset does not allow us to capture such a relationship and we compute developers' experience without considering relations between files. We consider two kinds of experience:

- Overall experience (OE): It is the total number of files and LOC in the patches already attached by a developer *e.g.,* two files and seven LOC in the example above.
- Relevant experience (RE): A task relevant experience. It is the number of files and LOC in all the files that are significantly relevant to the given task *e.g.,* one file and two LOC for the first scenario above, and zero file and zero LOC for the second scenario above.

6.4.3 Approach

We compute the similarity between all interaction/patch that we matched in Section 6.3.4. We identify how much developers use additional files *i.e.,* the percentage of additional files vs. significantly relevant files. We study how developers spend their effort according to the number of additional files by computing the Spearman correlation coefficient between developers' effort and the number of additional files. Then, we examine whether developers' effort depends on the bug severity and–or the developers' experience.

In **RQ1**, we observed that the effort spent by developers when performing a task is not correlated with the complexity of the implementation of the task. As mentionned in Section 6.4.1, developers must be careful when fixing severe bugs *i.e.,* they may spend more effort. However, the result of **RQ1** do not advise us whether the effort is different among tasks with different severity levels. To study the effect of bug severity on developers' effort, we first check whether the bug severity is related to the complexity of the implementation of tasks *i.e.,* do the implementations of tasks with different bug severities have different complexities? We perform the Kruskal-Wallis test to assess differences among the complexity of the implementation of tasks associated with different bug severity levels. Then, we investigate whether developers' effort depends on the bug severity by performing the Kruskall-Wallis test. We chose the Kruskal-Wallis test because it is a non-parametric method for testing whether samples originate from the same distribution. The Kruskal-Wallis test make no assumption about the distribution of the complexity of the implementation of tasks (for the first test) and developers' effort (for the second test).

To investigate whether the developers' effort depends on their experience, we compute the developers' experience as explain in Section 6.4.2. Then we use the Spearman correlation coefficient to assess the relation between the developers' experience and their effort. We use the Spearman correlation coefficient because it is a non-parametric test that does not make assumptions about the distributions of the metrics (*i.e.,* developers' experience and effort).

6.4.4 Results and Discussions

Additional Files

The similarity between the matched interactions and patches shows that some matchings have a low similarity (See Figure 6.2). Developers who attached interactions and patches with low similarity used more additional files. Since we removed unbalanced matchings as described in Section 6.3.4, there are no matchings with a similarity value equal to zero. The median, mean and standard deviation of similarities are respectively 0.26, 0.38, and 0.33. This result shows that **on average, developers use about 38% of *significant relevant files* and about 62% of *additional files***. We wonder whether the use of additional files affect developers' effort. We observe that developers who explore a large number of additional files spend more effort to perform the task *i.e.,* developers spend part of their effort exploring additional files. Table 6.5 presents the Spearman correlation coeficient between the number of additional files and developers' effort. These strong correlations suggest that most of the developers' effort is spent trying to understand the program and making the solution.

The perfect matching is when the matched interaction and patch are 100% similar *i.e.,* the developers did not use any additional files. There are 176 perfect matchings. The median, mean and standard deviation of files involved in perfect matchings are respectively 1, 2.25, and 2.83. Thus, developers did not use *additional files* to perform a task only when the number of *significantly relevant files* needed to perform the task was 2.25 on average. The converse is not true *i.e.,* developers can use additional files for some tasks that require less than two significantly relevant files.

Bug Severity

The distribution of entropy and change distance for different bug severities is shown in Figure 6.3. Figure 6.3a shows that changes made for *critical*, *enhancement*, *minor*, and *normal* bugs involved more files than changes made for other severities (*i.e., blocker*, *major*, *trivial*). Bug severities that involved fewer files did not necessarily required fewer changes. For example, a *blocker* bug that involved less files than a *minor* bug (See Figure 6.3a) can require more changes than a *minor* bug (See Figure 6.3b). The statistical analysis of the complexity of the task implementations wrt. bug severity (See Table 6.6) reveals that, except for the ECF project where *p*-value is 0.71 (for entropy) and 0.75 (for change distance), the complexity of task implementations is statistically significantly different among bug severities. This means that both the entropy and change distance are related to the severity of the bug.

Knowing that there is a relation between the entropy and the bug severity on one hand, and the change distance and the bug severity on the other (except for ECF project as mentionned above),

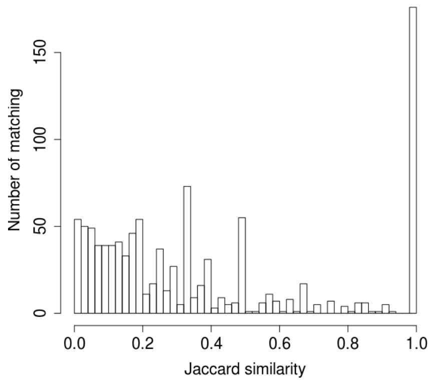


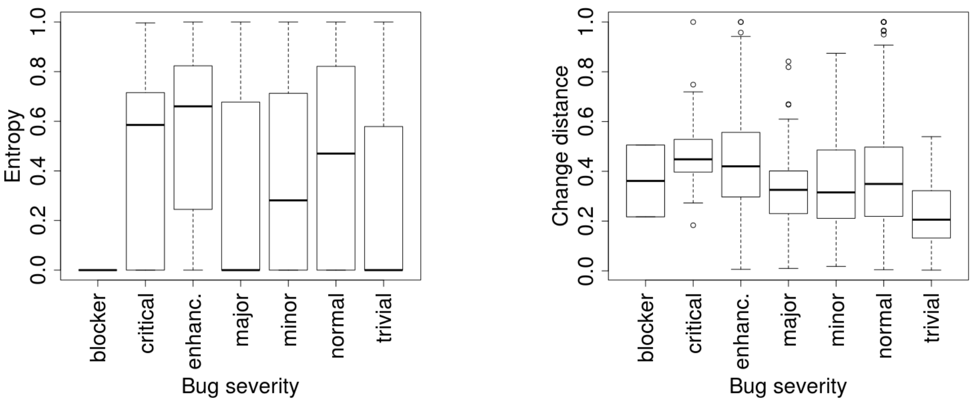
Figure 6.2 Similarity between matched interaction/patch

Table 6.5 Spearman correlation between the number of additional files and developers' effort.

	Time	Cyclomatic
ECF	0.72	0.87
Mylyn	0.58	0.81
PDE	0.57	0.76
Platform	0.69	0.78
All	0.63	0.78

we argue that **bug severities is related to the complexity of the implementation of tasks**. While bug severity is usually filed from project perspective (performance, stability, affected developers) by bug reporters or triager team, this result suggests that (1) severe bugs must be also complex or (2) those who assign severities may also consider severities from the perspective of complexity.

Table 6.7 shows that developers' efforts are different among bug severities for Mylyn and Platform projects. On the contrary, it seems that the bug severity does not affect developers' effort for ECF and PDE projects. We attribute this difference (of results) between the projects to our dataset of matchings. (1) some projects did not contained some bug severity levels (only Mylyn contains all bug severity levels) and (2) the number of matchings per bug severity is high for Mylyn and Platform projects compared to ECF and PDE. The standard deviation of the number of matchings per bug severity is 120.13 for Mylyn and 50.28 for Platform compared to 2.62 for ECF and 27.67 for PDE. The small number of matchings data for ECF and PDE projects may justify why the effort spend to fix the bugs is not different among severity levels.



(a) Entropy

(b) Change distance

Figure 6.3 Distribution of entropy and change distance per bug severity

Table 6.6 p-values of Kruskall-Wallis test for the complexity of implementation compared to bug severity

	Entropy	**Change Distance**
ECF	0.71	0.75
Mylyn	**4.7e-4**	**1.04e-5**
PDE	**8.5e-3**	**0.03**
Platform	**9.4e-4**	**2.8e-4**
Total	**2.4e-07**	**3.1e-11**

Developers' Experience

Concerning developers' experience, Table 6.8 shows that there is no consensus about the benefits of experience on the time spent (the Spearman correlation coefficient vary between -0.60 and 0.39). Table 6.9 shows the same trend for the complexity of exploration graphs *i.e.,* Cyclomatic complexity (the Spearman correlation coefficient vary between -0.66 and 0.57). The experience may reduce the effort in some cases *e.g.,* ECF project where the relevant experience reduce both the time spent (Table 6.8) and the Cyclomatic complexity of exploration graphs (Table 6.9). For the Platform project, the relevant experience tend to increase the complexity of exploration graphs (correlations 0.55 and 0.57 in Table 6.9). However, for the different measures of effort (time spent and Cyclomatic complexity of the exploration graphs), the relevant experience tend to have extreme values of correlation.

When looking at the size of the projects (in terms of the number of subprojects), we observe that the larger a project, the more the correlation coefficient between the effort and the number of relevant

Table 6.7 p-values of Kruskall-Wallis test for the developers effort compared to bug severity

	Time spent	Cyclomatic
ECF	0.91	0.38
Mylyn	**3.5e-9**	**1.06e-9**
PDE	0.24	**0.02**
Platform	**9.69e-5**	**7.9e-6**
Total	**1.2e-12**	**9.2e-12**

Table 6.8 Spearman correlation coefficient between the time spent and developers' experience

	NB	Overall Experience		Relevant Experience	
		NF	NLOC	NF	NLOC
ECF	0.17	0.03	**-0.12**	**-0.54**	**-0.60**
Mylyn	-0.08	**-0.10**	-0.07	0.34	0.31
PDE	**-0.27**	**-0.21**	0.04	0.31	0.28
Platform	0.18	0.24	0.28	0.33	0.39
All	-0.01	0.01	0.09	0.34	0.34

LOC increase *i.e.,* in increasing order of the number of subprojects ECF (one) - PDE (four) - Mylyn (11) - Platform (14), the respective values of correlation are -0.60, 0.28, 0.31, and 0.39 (for the time spent - See Table 6.8), and -0.66, 0.26, 0.29, and 0.57 (for Cyclomatic complexity - See Table 6.9). Thus, the way in which the experience can help save effort may depend on the size of the project since developers may work on different parts of a project. As shown with gray cells in Table 6.8 and 6.9, the number of bugs (NB) and the number of files (NF) for the overall experience may weakly decrease the time spent and the Cyclomatic complexity of the exploration graph for Mylyn and PDE projects. Robbes et Röthlisberger (2013) mined the interactions data for PDE and Mylyn and used different metrics to measure developers' experience. They found a negative correlation between the time spent and the developers' experience. They considered the experience based on the number of commits in the source code repository. The comparison of our correlation values (from -0.08 to -0.27 between the time spent and the number of bugs and the number of files) to their correlation values (-0.15 and -0.22) seems to indicate that one can use the number of bugs and the number of files to measure developers' experience.

The way a developer perform his maintenance tasks (and acquire experience) could also explain why this developer's experience does not reduce his effort over time. Figure 6.4 presents the evolution over time of the number of LOC implemented by three developers. We observe that some developers almost always perform tasks on the parts of the program that they never used before *i.e.,* they never had a relevant experience (See Figure 6.4a). Other developers perform tasks both on files used before and files that they never used *i.e.,* they always had some relevant experience, but

Table 6.9 Spearman correlation coeficient between the cyclomatic complexity of exploration graphs and the developers' experience

	NB	Overall Experience		Relevant Experience	
		NF	NLOC	NF	NLOC
ECF	0.07	-0.02	0.06	**-0.55**	**-0.66**
Mylyn	-0.05	-0.06	-0.04	0.36	0.29
PDE	**-0.11**	-0.08	0.18	0.32	0.26
Platform	0.34	0.48	0.48	0.55	0.57
All	0.07	0.14	0.21	0.41	0.40

they also performed more modifications on files that they never used before (See Figure 6.4b). We also observe that some developers always work on tasks that require almost the same set of files (*i.e.,* their overall experience and relevant experience are almost the same). After acquiring the experience on the files that they frequently used, they start performing the tasks that require some files that they never used before (See Figure 6.4c). Therefore, developers' experience may not reduce the effort required to perform a task because when a program evolves, developers may increasingly perform tasks on parts of the program on which they have no previous experience.

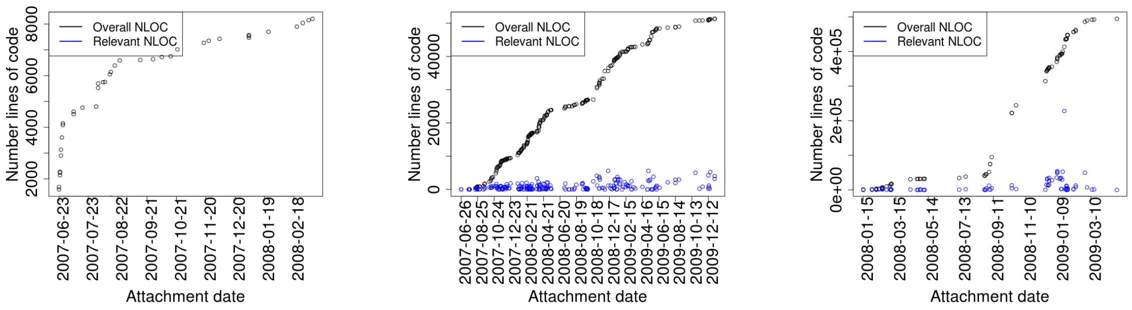
6.4.5 Sensitivity Analysis

We now study the sensitivity of our results to noise. The analysis that can be impacted by noise are the study of the correlation between the number of additional files and the time spent by the developers (the first column of Table 6.5), the relation between the time spent by developers and the bug severity (the first column of Table 6.7), and the analysis of developers' experience and time (in Table 6.8). We revisit these results using the time to which we add the sum of the idle time that are less than half a minute as reported in Section 4.2.3.

Our results using CITs show that considering noise in ITs improve a little bit the correlation between the number of additional files and the time spend by the developers (the first column of Table Table 6.5 vs. Table 6.10). This improvement reinforce the fact that developers spend a part of their time exploring additional files and, thus, reinforce our thesis on the assessment of the developers' exploration strategies and their relation with the effort.

The use of the CITs to relate the time spent by the developers to the bug severity does not have any impact on our results (the first column in Table 6.7 vs. Table 6.11) which shows that the time-related noise does not affect the relation between bug severity and the time spent by the developers.

We also observe a small improvement when we use the CITs to correlate the time spent by the developers to their experience (Table 6.8 vs. Table 6.12).



(a) Developers almost always perform the task on the files that they never used before

(b) Developers perform more changes on the files that they never used before

(c) After a period of time, developer perform more changes on the files that they never used before

Figure 6.4 Developers perform more modifications on the files that they never used before

Table 6.10 Spearman correlation between the number of additional files and the time spend by the developer (using CITs).

	Time spent
ECF	0.78
Mylyn	0.60
PDE	0.62
Platform	0.73
All	0.66

6.5 Threats to Validity

Construct Validity: Construct validity threats are related to our matching approach and the metrics used to measure developers' effort and the complexity of the implementation of tasks. Our matching approach may lead to some mismatchings *i.e.,* developers gather interactions and patches at different time period and attach them at the same time or vice versa. We mitigated the mismatching threat by removing the unbalanced matchings; we assumed that all interactions and patches gathered at different time and attached at the same time may be unbalanced. However, by removing these unbalanced matchings, our matching approach could have missed some matchings when developers gathered the interaction and the patch at the same time and attached them at different time. Therefore, we cannot guarantee that no matching was missed. However, we did not observed such cases in our dataset. The use of JGraphT and DiffUtils libraries may affect the computation of our metrics. We assume that the JGraphT tool is accurate because of its popularity *e.g.,* about

Table 6.11 Time spend by developers (using CITs) compared to bug severity

	Time spent
ECF	0.67
Mylyn	**3.9e-10**
PDE	0.13
Platform	**3.2e-5**
Total	**2.7e-13**

Table 6.12 Spearman correlation coefficient between the time spent (using CITs) and developers' experience

	NB	**Overall Experience**		**Relevant Experience**	
		NF	**NLOC**	**NF**	**NLOC**
ECF	0.10	-0.04	**-0.19**	**-0.49**	**-0.61**
Mylyn	-0.05	**-0.04**	-0.07	0.35	0.33
PDE	**-0.29**	**-0.25**	0.01	0.31	0.28
Platform	0.19	0.28	0.34	0.40	0.45
All	0.009	0.04	0.12	0.36	0.37

2,000 downloads per month[4] (from january, 1st to june, 30th 2013). We minimized the effect of the DiffUtils library by adapting its implementation and ensuring the accuracy of the parsing. While cyclomatic complexity indicates developers' effort, it is not necessarily accurate and complete. To overcome this limitations, we also used the time spend to measure the effort. Our expertise metrics is based on the files and LOC in the considered project. The developers' background and previous expertise in other projects may influence their performance within the considered project, and we may miss some experience for bugs that do not contain interaction traces or patches.

Conclusion Validity: Conclusion validity threats are related to the violation of the assumptions of the statistical tests and the diversity of our dataset. We used non-parametric tests (Kruskal-Wallis and Spearman correlation) that make no assertion about the distribution of the data. We used data from four open-source projects that have different sizes and involve many developers. Also, we do not claim causation, we simply report observations and correlations, although we try to explain these observations in our discussions.

Internal Validity: Internal validity threats relate to the tools used to collect interaction traces and the choice of our subject projects. We used Mylyn's interaction traces because the Mylyn plugin is the only tool that contributors to many open-source projects used to gather the interactions and provide them publicly. Our subject projects are the top four open-source projects that have more

[4] http://sourceforge.net/projects/jgrapht/files/stats/ (visited 20/10/2015)

interaction traces.

External Validity: External validity threats relate to the generalization of our results. Because our subject projects are open-source, we cannot guarantee that the findings of this study can generalize to proprietary and non-Java software projects.

6.6 Summary

Developers perform different kinds of tasks daily. Sometimes the implementation required for a task does not reflect the effort spent by developers on the task. If we want to improve the efficiency of developers, it is important to understand how these developers spend their effort when finding the solution to a task.

We study how developers spend their effort and find that the effort spent by developers when performing a task is not correlated to the complexity of the implementation of the task. Most of the effort appears to be spent during the exploration of the program. On average, 62% of files explored during the implementation of a task are not significantly relevant to the final implementation of the task. Developers who explore a large number of files that are not significantly relevant to the solution to a task take a longer time to complete the task.

The results above show that **we can use interaction traces to understand how developers spend their effort during maintenance and evolution tasks**. Yet, because parts of the developers' effort seem to be spent during the program exploration, we must understand how developers explore programs and the impact of their exploration strategies on maintenance effort.

CHAPTER 7 IMPACT OF PROGRAM EXPLORATION ON MAINTENANCE EFFORT

7.1 Introduction

The results reported in the previous chapter, together with the reality that developers always navigate through program entities when performing maintenance tasks motivate our investigation of developers' program exploration strategies. The purpose of program navigation is to find the subset of program entities that are relevant to the maintenance tasks. Program exploration involves three activities (Ko *et al.*, 2006; Sillito *et al.*, 2008): (1) looking at the initial entity that seems relevant (starting point), (2) relating the starting point to other entities and exploring them (expanding the starting point), and (3) collecting relevant information/knowledge to perform a task (understanding a set of program entities).

The way in which developers explore a program for a specific task, *i.e.,* their exploration strategy, depends on various factors, such as the characteristics of the task at hand (Ko *et al.*, 2006), developers' experience and proficiency, tool support, and the software design. The strategy may affect the successfulness of maintenance tasks (Robillard *et al.*, 2004) as well as the time spent to perform the task and the developers' productivity.

Thus, studying exploration strategies can help (1) to improve our knowledge on developers' comprehension process; (2) to characterise developers' expertise, *e.g.,* how experienced developers explore a program can differ from the way inexperienced ones do a program and how the strategies of experienced developers can be used to help inexperienced ones; (3) to find techniques and tools to reduce the developers' search effort and guide them when exploring a program.

As an initial step to achieve the above benefits, we study the developers' interaction traces collected from four open-source Eclipse projects to link exploration strategy to the duration and effort spent on maintenance tasks. For example, if a program contains entities $\{e_1, e_2, e_3, e_4\}$ and, for a particular maintenance task, a developer uses either exploration A = $e_1 \rightarrow e_2 \rightarrow e_1 \rightarrow e_3 \rightarrow e_1 \rightarrow e_4 \rightarrow e_1$ or B = $e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow e_4$, then A revisits entity e_1 multiple times compared to B. These revisits could mean that the developer did not explicitly recognized that e_1 is an important entity or that the developer uses e_1 as a reference point to come back to after losing the flow of exploration. In both cases, time and effort seems to be greater in the referenced exploration A compared to B.

To understand how developers' exploration strategies affect the time and effort spent on maintenance tasks, we mined 1,705 Mylyn interaction traces (IT) from four open-source projects (ECF, Mylyn, PDE, and Eclipse Platform). From the IT of each task, we computed the time and effort spent by developers performing the task. We performed a user study with nine participants who were asked to manually classify 104 ITs into referenced exploration (RE) and unreferenced exploration (UE). Next, based on the manually classified ITs, we automatically classify the remaining ITs and answer the following research questions:

RQ1: *Do developers follow a referenced exploration when performing maintenance tasks?*

We consider two extreme cases of exploration: referenced exploration (RE) and unreferenced exploration (UE). RE occurs when a developer reinvestigates one (or a set of) entity(ies) already visited (referenced entities). On the contrary, in an UE strategy, a developer visits program entities with almost the same frequency *i.e.,* there is no set of referenced entities. Results show that developers mostly follow the unreferenced exploration (UE) strategy when performing a maintenance task.

RQ2: *Does any difference exist in maintenance time between referenced exploration (RE) and unreferenced exploration (UE)?*

Maintenance time is the time spent performing a maintenance task. We found that the time spent on a maintenance task for UE is on average 12.30% less than for RE.

RQ3: *Does any difference exist in exploration effort between referenced exploration (RE) and unreferenced exploration (UE)?*

Exploration effort is the effort spent by a developer finding relevant program entities to modify in order to complete a task. Our results show that the more exploration effort a developer spends on a task, the more he is likely to follow the UE strategy.

We first explain how we collect and process the data (Section 7.2). Then, we conduct a user study to investigate if developers follow RE and UE and define an approach for automatic identification of developers' strategy (Section 7.3). Finally, we study whether RE and UE may results in different developers' time spent and exploration effort (Section 7.4).

7.2 Data Collection and Processing

We downloaded 3,609 bug reports from Eclipse Bugzilla. We consider 2,601 bug reports from four projects with the highest number of bug reports and at least one IT for each bug. We download, clean (*i.e.,* remove ITs that have at least one event with negative duration, or that have zero duration), and retain 2,375 interaction traces. We removed interaction traces in which only one or two classes were involved since they cannot be either UE or RE *e.g.,* the difference between the number of moves from one class to another will be always one for the interaction trace involving two classes. Overall, we kept 1,705 interaction traces. Table 7.1 presents a description of the data set. All data are available online[1].

We parse ITs as explained in Chapter 2. As an event can occur on a program entity at different levels, *i.e.,* file, class, attribute, and method levels, we take into account the containment principle. For example, at file level, we consider all the events that occurred on the file `Foo.java` and on the classes, attributes, and methods in the file `Foo.java`. As class is the primary concept in the object-oriented paradigm, we focus on the class level.

7.3 Do developers follow a referenced exploration when performing maintenance tasks?

The research question **RQ1** that we address in this Section is: *Do developers follow a referenced exploration when performing maintenance tasks?* There are several ways in which a developer can interact with entities while performing a maintenance task. Two extreme cases are when the developers do not have a privileged set of entities on which they concentrate their activities and when the developers concentrate their activities on a limited number of entities. In this chapter we are referring to these two extreme cases. To answer our research question, we perform a user study to investigate whether both referenced and unreferenced explorations occur in practice. This user study allows us to build a classifier to automatically classify an exploration as referenced or unreferenced.

7.3.1 User Study

We perform a user study in four steps: (1) we randomly sample the interaction traces; (2) we generate a graph representation of the sampled interaction traces; (3) we let the study participants classify the interaction trace graphs as referenced and unreferenced; and (4) we evaluate how well the participants agree on the exploration strategy.

(1) We choose the interaction trace sample size to achieve a 95%±10 confidence level *i.e.,* how certain (95% within 10% margin of error) we are that our sample accurately reflects the whole dataset. The sample was proportionally distributed among projects, except for the ECF project. Because of the small number of data from the ECF project in our data set, we consider half of all ECF interaction traces in our sample (instead of two interaction traces as suggested by the sample size). After the sampling, we observed that our sample contained program entities from multiple versions of each of our subject projects. Table 7.2 presents the size of the sub-samples of the projects.

[1] http://www.ptidej.net/download/experiments/wcre13a/

Table 7.1 Descriptive statistics of the interaction traces data

	Projects				
	ECF	**Mylyn**	**PDE**	**Platform**	**Total**
Number of bugs	138	1,603	464	396	2,601
Number of IH	158	2,309	567	579	3,613
Not Java IH	12	68	18	12	110
IH Duration < 0	2	109	8	34	153
IH Duration = 0	82	524	169	198	973
Retained IH	62	1,606	372	335	2,375
IH ≤ 2 classes	27	285	204	45	561
IH class level	26	1,273	131	275	1,705

Table 7.2 User study data and results

	Projects				Total	%
	ECF	**Mylyn**	**PDE**	**Platform**		
Sample size	13	68	7	16	104	100
Referenced	4	31	3	8	46	53.84
Unreferenced	8	22	1	5	36	34.61
Undecided	1	15	3	3	22	21.15

(2) To prepare the visual aids for manual classification, we define the number of revisits of a program entity as follows: $NumRevisit(anEntity)$ is the number of time the entity $anEntity$ is revisited, which is different from the number of events. Consider an interaction trace with five user interaction events that occurred on a set of three program entities $\{e_1, e_2, e_3\}$. If we suppose that the events occurred in the following order: $e_1 \rightarrow e_2 \rightarrow e_2 \rightarrow e_3 \rightarrow e_1$, then the number of revisits of the program entities are respectively two, one, and one, while the number of events are respectively two, two, and one. The number of revisits defines how much an entity is revisited compare to others. We generate the Graphviz (Gansner et North, 2000) representation of the interaction traces, *i.e.,* the exploration graph. Grapviz[2] is an open-source graph visualisation software. An exploration graph is a graph in which nodes are the program entities and arrow between two nodes (source and target) represents how developers move from one program entity to another, *i.e.,* a revisit of a (target) program entity.

(3) Nine subjects participated to the manual classification of exploration strategies (*i.e.,* the interaction traces). Among them, seven subjects were enrolled in the PhD program and one in the Bachelor program of software engineering at the École Polytechnique de Montréal. One subject was enrolled in a Master program of software engineering at INSA Lyon in France. There were five

[2] http://www.graphviz.org

female subjects and four male subjects. The median, mean and standard deviation of the number of years of experience with Java of the subjects are respectively 5, 4.16, and 2.80. The subjects were asked to manually analyse our sample of interaction traces and classify them into referenced exploration (RE) and unreferenced exploration (UE). In case of doubt, they were required to label the exploration trace with a D ("Doubt“). Before the user study, we gave a short training session to explain the concept of exploration graph to the participants. After the manual classification of exploration strategies, we performed a post-study interview with the following questions:

- How did you judge that a graph was RE or UE *i.e.,* wether a developer's exploration was based on a referenced set of entities or not?
- Did you had a doubt on some graphs? If so, please explain why?

(4) To aggregate the results of the subjects, we decide that an exploration is referenced (respectively unreferenced) if at least 2/3 of the subjects labeled the corresponding graph as RE (respectively UE). We consider interaction traces with less than 2/3 of either RE or UE labels to be undecided cases, *e.g.,* the interaction trace #83119 received 4/9=44.44% of RE labels, 4/9=44.44% of UE labels, and 1/9=11.11% of D labels. Table 7.2 shows the results of the user study. In total, 88.45% of interaction traces were classified by our subjects as either referenced (53.84%) or unreferenced (34.61%) explorations. We obtained 22 undecided cases representing 21.15% of the total number of interaction traces in our sample. 9 of the 22 undecided cases were labelled with D (*i.e.,* doubt) by at least one subject. The remaining 13 undecided cases were due to a lack of 2/3 agreement on either the RE or the UE label. We computed the Fleiss' Kappa interrater agreement coefficient to evaluate the classification agreement of our subjects. Fleiss' Kappa (Joseph L., 1971) is a generalized version of Cohen's Kappa that provides the interrate agreement between more than two raters on categorical data. We obtained an interrater agreement coefficient of 0.36. According to Landis and Koch's agreement benchmark (Landis et Koch, 1977), there is a fair agreement when the Kappa coefficient is between 0.21 and 0.40 and a moderate agreement when the Kappa coefficient is between 0.41 and 0.60. Thus, we can conclude that the subjects of our user study had a fair agreement. The agreement between our subjects is higher (*i.e.,* close to moderate) when distinguishing between RE (Kappa = 0.38) and UE (Kappa = 0.39). However, our subjects have a poor agreement on the undecided cases (Kappa = -0.009). Overall, these results show that our subjects are able to distinguish referenced and unreferenced exploration strategies quite successfully. Since we are studying only the two extreme exploration strategies RE and UE in this work, we decided to remove the undecided cases from the data used to train the exploration strategy classifier.

The user study post-questionnaire revealed that to classify exploration graphs, participants counted the number of nodes and the number of in/out arrows in the graphs, *i.e.,* they looked at the distribution of revisits across graphs. The subjects also explained that when they were not able to count the number of nodes and–or arrows (in large graphs) or when they thought that parts of a graph were RE while other parts were UE, they labelled the graph with D.

7.3.2 Automatic Identification of the Exploration

Based on the result of the user study, we define a technique to automatically identify exploration strategies. We use the Gini inequality index to measure the distribution of revisits.

Gini Inequality Index

Based on how participants identify developers' exploration, the goal is to measure how program entities are equally or unequally revisited. In econometrics, many inequality indices are used to measure the inequality of income among a population. We choose the Gini inequality index because (1) it has been used in previous software engineering studies (Martínez-Torres *et al.*, 2010; Mordal *et al.*, 2012; Vasa *et al.*, 2009) and (2) the mathematical properties of the Gini inequality index presented by Mordal *et al.* (2012) are conform to the *number of revisits* and the *number of entities* referred by the participants in the post-questionnaire of the user study. We are interested in the (un)equality of revisits among program entities involved in an interaction trace. The set of program entities involved in an interaction trace is our population. The income of a program entity is its number of revisits.

The Gini inequality index has a value between zero and one. Zero expresses a perfect equality where everyone has exactly the same income while one expresses a maximal income inequality. Xu (2004) presented many computational approaches for the Gini inequality index and mentioned that theses approaches are consistent with one another. As used in (Martínez-Torres *et al.*, 2010; Mordal *et al.*, 2012), we use the mean difference approach defined as "*the mean of the difference between every possible pair of individuals, divided by the mean size μ*". We calculate the Gini inequality index as follows (n is the total number of program entities and e_i represents an entity i):

$$Gini = \frac{1}{2n^2\mu}\sum_{i=1}^{n}\sum_{j=1}^{n} \mid NumRevisit(e_i) - NumRevisit(e_j) \mid$$

Identification Process

To automatically identify the two extreme cases of exploration, we must define a threshold to determine if entities are equally or unequally revisited. After the definition of the threshold (explained below), we identify the exploration as follows:

- If the Gini value is less than the threshold, the visited entities are almost equally revisited.

Thus, the developer explored the program entities almost equally. We say that the exploration is unreferenced (UE) because the developers do not have a privileged set of entities on which they concentrate their attention.

- If the Gini is greater or equal to the threshold, it means that the revisits are concentrated on a few program entities, *i.e.,* reference entities. We say that the exploration is referenced (RE).

Identification Threshold

We use the oracle build in Section 7.3.1 (*i.e.,* manual classification of the ES) to define a threshold to distinguish RE and UE. We proceed in two steps. In the first step, we use 10 threshold values ranging from 0.1 to 1 per step of 0.1. We applied the exploration identification process above to automatically classify the strategies for the sample data used in Section 7.3.1. The automatic classification was independent from the manual classification (oracle). In step two, we compare the manual classification (oracle) and the automatic classification for the considered threshold values. Then we, chose the threshold value with high precision and recall. To maximize both precision and recall, we computed the F-Measure as follows:

F-Measure $= 2.\frac{precision.recall}{precision+recall}$

Figure 7.1 shows the distribution of the F-Measure for threshold values from 0.2 to 0.5. We plot this range of threshold because the F-Measure decrease before and after threshold 0.4. Figure 7.1 indicates that the identification of the exploration is most accurate at 0.4 threshold (the median at the threshold 0.4 is 0.91 vs. 0.90 at the threshold 0.3). Therefore, we consider the value 0.4 as the threshold.

According to the considered threshold (0.4), Table 7.3 (column “Exploration") presents the percentage of RE and UE found in our studied projects. We observe that:

Observation 1*: Developers follow mostly the unreferenced exploration (UE) when performing a maintenance task.*

Observing that there are more UE than RE, we look at the frequency of the number of classes involved in the IT (See Figure 7.2b for UE and Figure 7.2a for RE). There are 711 UE interaction traces (57.94%) in which less than 10 classes are involved. For RE, there are 71 interactions traces (14.85%) in which less than 10 classes are involved. Therefore, **the UE tend to be followed when less classes are involved**.

When studying how developers explore source code, Robillard *et al.* (2004) observed that methodical developers do not reinvestigate methods as frequently as opportunistic developers. Methodical

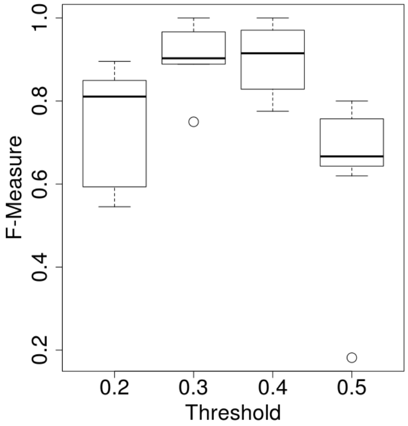


Figure 7.1 F-Measure per threshold for the oracle

Table 7.3 Percentage of referenced and unreferenced exploration and p-values

		Exploration		p-values		
		#	%	Avg. class level duration	Avg. overall duration	Avg. edit ratio
ECF	**RE**	4	15.38	0.11	**0.019**	0.12
	UE	22	84.61			
Mylyn	**RE**	306	24.03	**<2.2e-16**	**1.4e-10**	**< 2.2e-16**
	UE	967	75.96			
PDE	**RE**	31	23.66	**6.7e-05**	**0.005**	**4.1e-05**
	UE	100	76.33			
Platform	**RE**	137	49.81	**4.4e-06**	**0.016**	**9.9e-12**
	UE	138	50.18			
Total	**RE**	478	28.03	**< 2.2e-16**	**4.3e-16**	**< 2.2e-16**
	UE	1227	71.96			

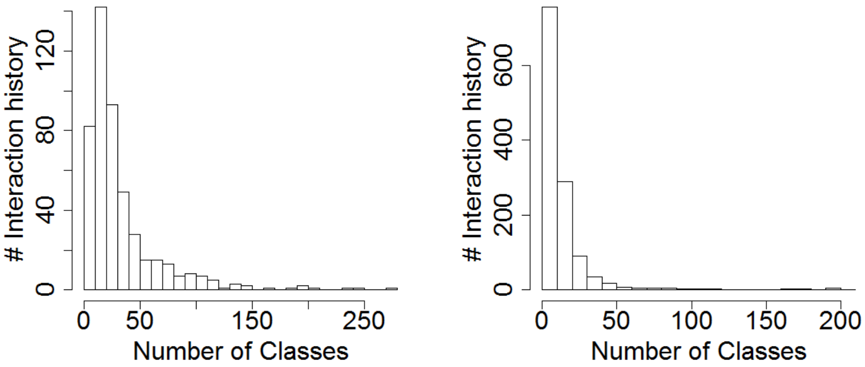
developers seem to answer specific questions using focused searches, while opportunistic developers guess more and read the source code in details (Robillard *et al.*, 2004). The number of revisits used to identify exploration somehow measures the reinvestigation frequency. We need more investigations to ascertain whether methodical developers are those who follow unreferenced exploration.

7.4 Does Exploration Strategies affect Maintenance Effort?

This section presents our empirical study that addresses the following two research questions:

RQ2: Does any difference exist in maintenance time between referenced exploration (RE) and unreferenced exploration (UE)?

RQ3: Does any difference exist in exploration effort between referenced exploration (RE) and unreferenced exploration (UE)?



(a) Referenced Exploration (b) Unreferenced Exploration

Figure 7.2 Frequency of the number of classes involved in the interaction traces

The corresponding null hypotheses are:

$H0_{Time}$: There is no difference in the average time spent between RE and UE when developers perform a maintenance task.

$H0_{Effort}$: There is no difference in the average exploration effort between RE and UE when developers perform a maintenance task.

First, we compute a set of metrics on the interaction traces. Then, we perform the statistical analysis to investigate our research questions and present the results and discussions. For statistical analysis, we perform an unpaired version of the non-parametric Wilcoxon test. We use a non-parametric test because our data is not normally distributed. For all statistical tests, we use a 5% significance level (*i.e.,* $\alpha = 0.05$).

7.4.1 Metrics

We compute the following metrics on the IT:

- *Overall duration* is the duration of an interaction trace. We consider the overall duration as the accumulated time defined in Section 4.2.3 and we remove the idle and overlap times. To control the confounding effect of the number of entity involved in an interaction trace, we divide the overall duration by the total number of entity involved in an IT to obtain the average overall duration.

- *Class level duration* is a cumulative duration spent on entities at class level in an interaction trace. We compute the duration at class level in the same manner as overall duration by considering only the events on the entities at class level. To control the confounding effect of the number of class involved in an IT, we divide the class level duration by the total number of class involved in an IT to obtain the average class level duration.

- *Exploration effort*: we use an *edit ratio* to measure the exploration effort. An edit ratio is the number of edit events divided by the number of events. The *number of events* (*NumEvent*) is the total number of user interaction events in an IT. The *number of edit* (*NumEdit*) is the total number of edit events in an IT. $anEvent$ is an edit event if $kind(anEvent) =$ "*Edit*".

  $$NumEdit(anEvent) = \begin{cases} 1 & \text{if } kind(anEvent)\text{=``}Edit\text{''} \\ 0 & \text{if } kind(anEvent)\text{!=``}Edit\text{''} \end{cases}$$

  $$EditRatio(IT) = \frac{NumEdit(IT)}{NumEvent(IT)}$$

  The exploration effort measures the effort spent by a developer to find the relevant program entities to edit. Röthlisberger *et al.* (2011) states that developers perform on average 19.31 exploration events between two edits. The motivation behind using edit ratio to measure exploration effort is that the more developers perform edit events, the less they spent effort to find the relevant entity(ies) to modify. The less they perform edit events, the more they spent effort.

7.4.2 Results and Discussions

RQ2 Does any difference exist in maintenance time between referenced exploration (RE) and unreferenced exploration (UE)?

In **RQ1** (Section 7.3), we found that developers follow both RE and UE when performing maintenance tasks. We conjecture that these explorations can affect the time spent to perform a task. In fact, when developers explore the source code, their exploration can reflect their mental model and the difficulties that they have to understand the code and perform a task. In this research question, we investigate at class level and on the whole task, whether the time spent by developers to perform a task is affected by their exploration strategy.

Table 7.3 shows that there is significant difference (at class level and overall) in the average time spent between RE and UE. Thus, we can reject the null hypothesis $H_{0_{Time}}$. In general, **the exploration strategy affects both the duration at class level and the overall duration of an IT**. For the ECF project, this does not hold at the class level, possibly because there are only 26 interaction traces.

Without distinguishing the projects, we found that **the RE is the most time consuming strategy** for both class level durations and overall durations. The mean of class level durations for RE is

41,030 sec. vs. 22,470 sec. for UE. The standard deviation of class level durations for RE is 326,081.1 sec. vs. 187,624.9 sec. for UE.

Observation 2: *For class level duration, the UE is on average 45.23% less time consuming than the RE.*

The mean of overall durations for RE is 7,817 sec. vs. 6,855 sec. for UE. The standard deviation of overall durations for RE is 49,734.88 sec. vs. 39,367.88 sec. for UE.

Observation 3: *For the overall duration, the UE is on average 12.30% less time consuming than the RE.*

Figure 7.3 compares the logarithms of the overall durations of RE and UE for each of our studied projects.

While developers who perform a RE mostly revisit the entity(ies) already investigated, it seems that (1) they guess and don't know exactly what they are looking for or (2) they come back to their reference entity(ies) after losing the flow of exploration. On the contrary, the less time spent when following a UE may be because developers who follow a UE look at explicit program entity(ies). Robillard *et al*. (2004) states that the methodical investigation of a source code does not require more time than an opportunistic investigation. More investigations should be done to tie our work to Robillard *et al*. (2004)'s one.

Typically in open-source projects, developers are volunteers. Therefore, they address the tasks (*e.g.,* bug fixing) that are assigned to them on their spare time. Because of lack of time, they could be working on one change request across several days. To analyse this, we compute the number of working days for each interaction trace. When we study the percentage of interaction traces per number of working days, Table 7.4 shows that developers work for one or two days on about 75% of interaction traces and more than two days on about 25% of interaction traces. Even with this unbalanced proportion, the distribution of the logarithm of duration (see Figure 7.4) shows that the more days developers work on a change task, the more time they spend on program entities.

As RE is more time consuming, more time spent for more working days indicates that a RE is probably the most followed strategy when a task spans multiple working days, as shown in Figure 7.5. Therefore, we conclude that when developers work on maintenance tasks for less than three days, more often, they follow a UE. On the contrary, when a maintenance task spans four or more days (*i.e.,* is extensive), developers follow the referenced exploration frequently. By extensive work, we mean that the number of days spent by a developer on a change request is greater than three days.

We think that two reasons can justify why more extensive works result into more RE. First, when the work is extensive, developers must (re)understand the entities that they explored before. So,

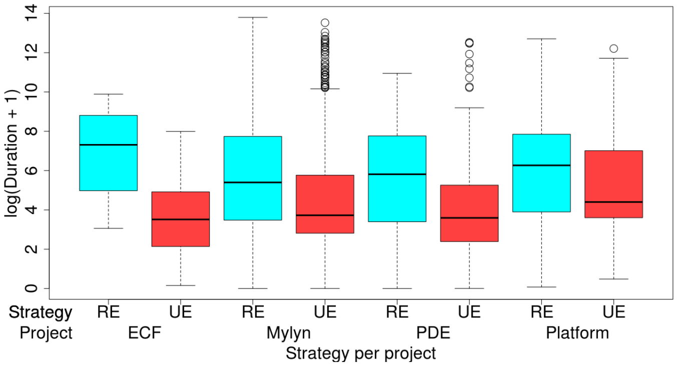


Figure 7.3 Distribution of overall duration per project

they refresh their mind by (re)exploring the entities. Second, when a developer re-activates a task on which she was already working, all the entities in the context of the task are reloaded by Mylyn and the developer usually (re)explore these entities before moving on to new entities. This feature of Mylyn is likely to push developers to (re)explore entities already explored in previous working sessions.

RQ3 Does any difference exist in exploration effort between referenced exploration (RE) and unreferenced exploration (UE)?

Similarly to **RQ2**, because a RE means that developers perform back and forth navigation on a set of program entities compared to UE, a RE may be more costly than UE in term of exploration effort. By definition (see Section 7.4.1), a low value of *EditRatio* indicates a small number of edit events and a high number of other events: the developer spent more exploration effort. When the *EditRatio* is high, the developer spent less exploration effort and performed edit events more frequently.

As shown in Table 7.3, except for the ECF project, developers' exploration efforts are significantly different for RE and UE. Thus, we can reject the null hypothesis $H_{0Effort}$. By investigating the less costly exploration in terms of exploration effort, Figure 7.6 shows that the edit ratio of UE is always smaller than that of RE for all projects, *i.e.,* UE may require more exploration effort than RE.

Observation 4: *An unreferenced exploration requires more effort than a referenced exploration.*

The fact that UE lead to more exploration effort is surprising because they require less back and forth. Yet the fact that developers who follow RE have less exploration effort can be justified by two

Table 7.4 Percentage of interaction trace per number of working days

	Number of working days						
	1	**2**	**3**	**4**	**5**	**6**	**7**
ECF	69.23	15.38	15.38	0	0	0	0
Mylyn	54.43	22.54	11.07	5.34	4.24	1.72	0.62
PDE	56.48	20.61	12.21	6.87	3.05	0.76	0
Platform	38.90	22.18	12.36	6.54	10.54	2.18	7.27
Total	52.31	22.22	11.43	5.57	5.10	1.70	1.64

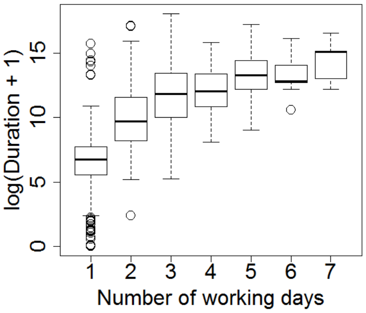


Figure 7.4 Distribution of duration per number of working days

reasons: (1) they make their code modifications almost in one place (*i.e.,* on the entities they are concentrated on) and reduce their non-edit events or (2) they start editing program entities before fully understanding the program and then could have to revert/modify their previous edits as they explore more program entities. In future work, we plan to map interaction trace modifications (edit events) and commit modifications from the source code repository to compare real modifications of the source code with revert/canceled modifications that we expect to be frequent with RE.

7.4.3 Confounding Factors

In this section, we discuss some factors that can somehow affect our study of exploration strategy.

Architecture of the System

There may be a relation between the ES and the program architecture. We use four open-source projects and, except for ECF (possibly due to the small number of IT), we did not observe an impact of the systems on the ES. However, developers explore the program by following different kind of relationships (Singer *et al.*, 2005). We conjecture that the architecture of the program can affect the exploration strategy. By definition, the program entities involved in an IT are the part (architecture)

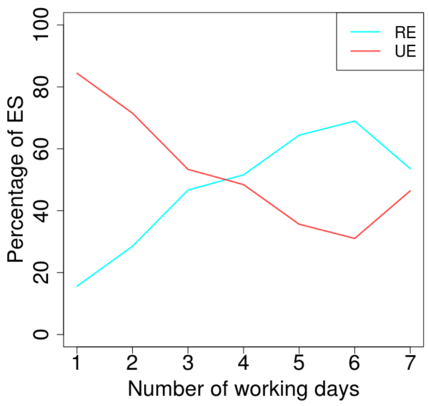


Figure 7.5 Percentage of exploration per number of working days

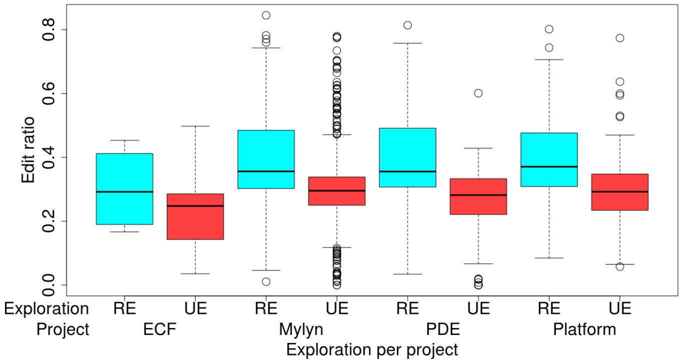


Figure 7.6 Distribution of effort per project

of the system used to perform a task. Thus, if the exploration is guided by architecture, the ITs using almost the same part of a system will result in the same exploration strategy. For example, if two ITs A and B pertain to almost the same part of a system, they could yield the same exploration (RE or UE). But, if A and B pertain to different parts of a system, they could yield different explorations (RE for A and UE for B or vice-versa). We use the *number of common entities* in A and B to capture the same part of a system involved in A and B. To investigate the architecture threat, we compute the *number of common entities* between each pair of interaction traces. Consider three ITs A, B, and C involving classes: $A = \{c1, c2, c3, c4\}$, $B = \{c1, c2, c3, c5\}$, $C = \{c6, c7, c8\}$. A and B have common classes while A and C and B and C have no common class: $A \cap B = \{c1, c2, c3\}$, $A \cap C = \varnothing$, and $B \cap C = \varnothing$. If the exploration is guided by the architecture, A and B should yield the same exploration strategy while C will have possibly different exploration strategy.

We study the number of common entities for each pair of interaction traces. Except for the Platform project (p-value = 1.1e-06), the number of common entities is not statistically different between the pairs of different ES and the pairs of same ES (ECF: p-value = 0.34, Mylyn: p-value = 1, PDE: p-value = 1). Without distinguishing the projects, there is no statistical significant difference (p-value = 1). Therefore, **architecture does not affect the ES**.

Task Interruption and Switching

Task interruption is a common problem when developers perform a task. Zhang *et al.* (2012) found that task interruption increases the risk of bugs in files while Parnin et Rugaber (2011) identified how developers address the task interruption problem. For exploration strategy, we find that **when developers follow a RE, their interruption time is higher than those of developers following a UE**.

Concerning the **task switching**, if it is true that developers can work on many task at a time, we think that Mylyn features minimise the task switching effect. In fact, when gathering the interaction traces, Mylyn requires developers to activate the task they are working on. The task ID is unique and appears in the interaction trace because only one task can be activated at a time, *i.e.,* if T1 is activated and developer try to activate T2, T1 will become automatically deactivated. Yet, we think that more empirical study must be performed for task switching. Ko *et al.* (2006) observe that developers spent on average 5% of their time switching between applications (IDE, Web browser, etc.). It is another dimension of switching that could be related to ES and that must be investigated in the future work.

Type of the Task

The exploration strategy could be related to the type of the task. We looked at the relation between RE and UE and the type of the task by using the bug severity as the type of the task. We think that developers may be careful when fixing severe bugs. Therefore, when fixing severe bugs wrt. less severe bugs, developers may have more back and forth navigation to validate their changes and make sure that they are not introducing new bugs. Because some reporters of the bugs may not follow the guideline for assigning the bug severity, we aggregate the bug severity as Ying et Robillard (2011) to address the imprecise nature of bug severity: "*enhancement tasks (only consisting of the enhancement severity category), minor bug fixes (aggregating minor and trivial severity categories), and major bug fixes (aggregating blocker, critical, major, and normal severity categories)*". Since the exploration strategy is based on the Inequality index, we perform a Kruskall-Wallis test to assess whether different types of task have a different Inequality index. Except for the Mylyn project, we found that the difference between the Inequality index of different types of tasks is not statistically significant. Moreover, the percentages of RE and UE presented in Table 7.5 show no significant relation between the type of a task and the exploration strategy of developers.

Table 7.5 Percentage of RE and UE for each type of task

	enhancement	**major**	**minor**
RE	203 (42.46%)	239 (50%)	36 (7.53%)
UE	433 (35.28%)	638 (51.99%)	156 (12.71%)
Total	636 (37.30%)	877 (51.43%)	192 (11.26%)

7.4.4 Sensitivity Analysis

We report the study of the sensitivity analysis of the impact of noise on our results. The parts of our results that could be impacted by noise are the relation between exploration strategies and overall time and edit ratio *i.e.,* columns ”Avg. overall duration“ and ”Avg. edit ratio“ in Table 7.3 which present the results of the statistical test. We recompute the time (overall duration) and the edit ratio considering our prediction-based approach for cleaning ITs *i.e.,* using CITs.

Table 7.6 presents the results of our analysis which shows that the time spent by developers is significantly different between RE and UE. This results is not different from the results obtained with RITs (See Table 7.3). On the contrary, there is no difference between edit ratios of developers following RE and the ones following UE. This results is different from the results with RITs and indicates that exploration strategies do not affect edit ratio. The absence of difference between edit ratios of RE and UE shows that, because the CITs consider the changes performed on code, the way developers explore program do not affect the modification of the code.

7.5 Threats to Validity

In this study, we examine the effect of two program exploration strategies (*i.e.,* referenced exploration (RE) and unreferenced exploration (UE)) on task duration and effort. We cannot claim causation, we simply report observations and correlations, although we try to explain these observations in our discussions. The remainder of this section discusses the threats to validity of our study following common guidelines (Yin, 2002) of empirical studies.

Construct Validity

Construct validity threat is related to the identification of exploration strategy and the metrics that we use to measure their impact on maintenance tasks. Gini inequality index is recognized to be a reliable measure of inequality and has already been applied in software engineering. The number of revisits defines how much a program entity is relevant for a developer's task. A wrong computation of the number of revisits could affect our study. We mitigate this threat by computing the number of revisits only for developers' interaction events, instead of considering also Mylyn prediction events. We based our selection of a Gini threshold on a user study. Our user study is subjective and

Table 7.6 p-values of the statistical test comparing the time and edit ratios of referenced and unreferenced exploration

	Avg. overall duration	**Avg. edit ratio**
ECF	**0.016**	0.29
Mylyn	**< 2.2e-16**	1
PDE	**2.2e-4**	0.30
Platform	**2.8e-3**	0.99
All	**< 2.2e-16**	1

depends on the way Graphviz displays the exploration graphs. We have no control on Graphviz, however, all subjects worked on the same displayed graphs.

Conclusion Validity

Conclusion validity threat concerns possible violations of the assumptions of the statistical tests, and the diversity of data used. To avoid violating the assumptions of our statistical tests, we use an unpaired version of the non-parametric Wilcoxon test because it makes no assertion about the normality of the data. Concerning the diversity of the data, our study is based on real open-source projects; we think that many developers with different expertise are involved in these projects. Moreover, theses projects evolved differently and have different developers. They have different sizes and complexity.

Internal Validity

Internal validity threat relates to the tools used to collect interaction traces and the choice of the projects. Many tools (Negara *et al.*, 2012; Röthlisberger *et al.*, 2011) can collect developers' interactions with the IDE. We use Mylyn's interaction traces because (1) Mylyn is a tool provided as an Eclipse's plug-in and (2) all contributions to Mylyn must be made using Mylyn[3], *i.e.,* in contrast to other tools, the Mylyn interaction traces are available. Concerning the projects, because we use the Mylyn interaction traces, we are constrained to use projects that have Mylyn interaction traces available. Thus, we use the top four projects using Mylyn to gather developers' interactions.

External Validity

External validity threat concerns the generalization of our results. In our study, we used Mylyn interaction traces gathered from four Eclipse-based projects. The fact that our subject projects are open-source projects may affect our results. More investigations should be done using (1) data collected with other tools and (2) other subject projects that are not open-source.

Reliability Validity

[3] http://wiki.eclipse.org/index.php/Mylyn/Contributor_Reference#Contributions

Reliability validity threat concerns the possibility of replicating this study. All data used in this study are available online for the public. Finally, it is the authors opinion that it pays to be cautious as the sub population of developers working with Mylyn and recording interaction trace is a specific developer sub population. Findings, even if interesting may or may not be representative of the general developers population. This is a first study investigating if indeed different exploration strategies impact (at least in the case of Mylyn aware developers) the time and effort in maintenance tasks. More work is needed, for example to verify if there are more fine grain exploration strategies or to verify if other metrics beside the Gini index, possibly including developers experience, application complexity, may help to better model and understand the underlying phenomenon.

7.6 Summary

When developers perform a maintenance task, they must explore some program entities. Understanding how developers explore programs can help to evaluate developers' exploration performance, improve our knowledge on developers' comprehension process, and characterise developers' expertise. We contribute to the understanding of developers' exploration strategies in two ways. First, after mining Mylyn's interaction traces, we performed a user study with nine subjects to verify if both referenced exploration (RE) and unreferenced exploration (UE) are followed by developers when performing maintenance tasks. The subjects of this user study were asked to classify developers' exploration logs (*i.e.,* ITs) into two categories: referenced exploration, when developers explore repeatedly one (or a set of) entity(ies), and unreferenced exploration, when developers explore entities without privileging a set of entities. The interrater agreement among the subjects of the user study was fair. Using the Gini inequality index on the number of revisits of program entities, we automatically classified interaction traces from ECF, Mylyn, PDE, and Eclipse Platform into RE and UE and performed an empirical study to measure the effect of program exploration on the task duration and developers' exploration effort.

Results show that although a UE requires more effort than a RE, a UE is on average 12.30% less time consuming than a RE. We also found that maintenance task taking up to more than three days typically imply a RE.
We observe that some characteristics of exploration strategies (*e.g.,* revisits and time) are common to the characteristics of methodical and opportunistic developers. However, we need more investigations to fully tie exploration strategy to Robillard *et al.* (2004)'s results.

Through this study, we show that **we can use interaction traces to assess developers' program exploration strategies and study their impact on developers' effort**. Yet, our results also show the need for tool support to help developers during program exploration. We now conclude our thesis and report some perspectives to investigate in future work.

CHAPTER 8 CONCLUSION

Nowadays, many vital software systems are complex to understand and evolve (Sommerville, 2011). Program comprehension is a major activity performed during software maintenance and evolution, prior to any change (Boehm, 1976b; Singer *et al.*, 1997). To assist developers during maintenance and evolution, especially when comprehending software systems, researchers use various methods to study how developers' understand program. Developers usually explore the program and interact with program entities to find where and how to perform appropriate changes. Therefore, the thesis of this dissertation was:

We can collect and use accurate interaction traces to assess developers' effort, understand how developers' spent their effort, and assess their program exploration strategies during maintenance and evolution activities.

8.1 Contributions

We validated our thesis through the following contributions.

We investigated the quality of interaction traces (ITs) data and study whether ITs contain noise due to incorrect assumptions. The motivations behind this investigation was that previous works usually made assumptions on ITs data and these assumptions may bias the results of subsequent analyses. We found that ITs data contain time-related noise (6% of time missed) and edit-related noise (28% of false edit-events). Then, we proposed two approaches to clean noise in ITs. Our best approach achieved form 31% to 98% precision and from 77% to 90% recall. Based on this findings, we concluded that previous works have made wrong assumptions on ITs data and we recommended researchers to not made these assumptions anymore. We reported our findings to a community that build the monitoring tool for collecting ITs (*i.e.,* Mylyn). The opinion from Mylyn community confirmed that noise is prevalent in ITs: "*... the argument that there is noise in the edit events makes sense to me ... The edit events don't have to be textual edits ...*". Therefore, we studied whether noise would impact previous studies.

We investigated the impact of noise on previous studies to assess whether noise should be considered or not when mining ITs data. We replicated two representative previous works using both ITs with noise (raw ITs — RITs) and ITs without noise (cleaned ITs — CITs). Then, we compared the obtained results and found that there are differences between RITs and CITs. About 41% of ITs lead to different editing styles, which shows that cleaning ITs may lead to improvement of the categorisation of ITs. Our results also revealed that, by cleaning noise in ITs, the accuracy of recommendation systems could be improved by up to 51% of precision and 57% of recall. This results show that noise in ITs bias the results reported in previous works. Thus, researchers should not make these assumptions anymore. We recommend the community to use our approach to clean ITs data prior to their usage. To further make our point when validating our thesis, we used both RITs and CITs to assess the sensitivity of our findings to the accuracy of the ITs data.

We studied how developers spend their effort during maintenance and evolution tasks. We first used ITs to assess developers' effort. Then, we matched ITs to the patches provided by the developers as the solutions to the tasks. We used the patches as proxies to the complexity of the implementation of the tasks. We correlated the complexity of the implementation to the developers' effort and observed that the effort spent by developers is not correlated to the complexity of the implementation. Developers used on average about 62% of files that must not changed to resolve the tasks (*i.e.,* additional files) and, the more developers used additional files, the more they spent effort. Thus, we concluded that a part of the developers' effort is spent during their exploration of the program, which calls for an assessment the impact of program exploration strategies on maintenance effort.

We assessed the developers' program exploration strategies and their impact on the maintenance effort. We identified the developers' program exploration strategies as the way they navigate through program entities. We conducted a user study to assess whether developers follow referenced exploration (*i.e.,* back and forth to the set of program entities) or unreferenced exploration (*i.e.,* almost the same frequency of revisitations of program entities). We found that developers mostly follow the unreferenced exploration strategy when performing a maintenance task and that unreferenced exploration is on average 12.30% less effort consuming than referenced exploration. Our findings showed the need for characterizing developers' exploration strategies and identifying the ”good“ strategy that may guide developers during program exploration.

Yet, our findings may suffer from some limitations.

8.2 Limitations

The limitations of our work are reported in Sections 4.4, 6.5, and 7.5. Most importantly, our work suffers from limitations related to the monitoring tools that capture developers' interactions.

Pre-processing of the interaction traces: Monitoring tools may internally pre-process the interaction traces because, depending on the goal of the tools, they may generate interactions that are not triggered by developers. For example, Mylyn uses the relation between program entities to generate propagation events that are considered as events resulting from the event that developers triggered on a related entity. Mylyn also aggregates some events of the same kind that occurred on the same program entity that results in overlaps between events.

Missing events: Monitoring tools may not collect all of developers' interactions, for example, if developers quickly move between program entities, if developers apply an action (*e.g.,* refactoring) that involved many program entities, or if developers use different ways to perform the same kind of actions (*e.g.,* navigate static relations using keyboard shortcut or contextual menu). Moreover, monitoring tools can not efficiently deal with developers' inactivity because some of the developers' inactivity belong to the maintenance task, such as thinking and reading source code, without any interaction with the IDE.

Therefore, these limitations and our contributions open several perspectives.

8.3 Future Work

We divide perspectives in short-term and long-term perspectives.

8.3.1 Short-term Perspectives

Improving Monitoring Tools

Our current knowledge of noise in ITs and the interest of Mylyn community to clean ITs lead to anticipate the cleaning of ITs to improve monitoring tools. One Mylyn developer suggested that we clean the identified noise during the creation of events as we "*can capture more information at that point, for example, whether there was a keystroke performed that correlates to the edit selection*". This is a short-term perspective to implement our contribution into Mylyn and, thus, help researchers by providing more accurate ITs and practitioners by improving their productivity. This implementation should be validated to assess how much we improve the quality of ITs and the developers' productivity.

Qualitative Analysis and Sequence of Activities

Our investigation of noise in interaction traces is mainly quantitative. While our quantitative analyses provide knowledge of noise in interaction traces, a qualitative analysis may reveal useful observations. We plan to study whether developers follow a typical problem-solving approach *i.e.,* problem-solution-test (Maalej *et al.*, 2014b). A qualitative analysis of developers' videos and development phases (problem, solution, test) may reveal whether the problems faced by developers during maintenance and evolution are different between development phases. Differences would indicate the need for different help at different development phases that also motivate one of our long-term perspective in Section 8.3.2.

Improving Effort-Aware Tools

Our contributions on assessing developers' effort through interaction traces can also help to improve effort-aware issue tracker systems. Indeed, issue tracker systems, such as JIRA, have features that enable developers to input the value of effort that an issue takes to address (Shihab *et al.*, 2013). By mining effort from interaction traces, we could automatically input the effort computed from interaction traces. Thus, effort-aware issue tracker systems could use the effective effort instead of the estimated effort provided by the developers. This effective effort could be used to predict precisely the effort needed to address new issues and then improve bug triaging.

8.3.2 Long-term Perspectives

New Feature Location Methods

One of our contributions shows that developers spend more effort when using more additional files. However, additional files can help developers to identify the files that must be changed to resolve the tasks. Feature location methods aim to retrieve program entities that are significantly relevant to the task. These methods are also validated according to the accuracy of the identification of significantly relevant entities. We think that these methods partly help developers because they do not provide developers with the entities that they should understand to change the significantly relevant ones. In particular, they limit the help provided to developers who are unfamiliar with the systems under maintenance (*e.g.,* novices, new comers) and who need to understand parts of the systems that are not relevant to the tasks, but that are useful to resolve the tasks. We aim to study the relevance and usefulness of program entities: how can we distinguish the useful entities from additional entities? Identifying useful entities and distinguishing them from additional entities may help to improve and evaluate feature location methods. For example, useful entities should be considered when evaluating recommendation and feature location methods, *i.e.,* to provide results with useful entities may be more efficient for novice developers than only significantly relevant entities.

Smart Development Environments

Our contributions on program exploration strategies open an opportunity to improve development environment towards smart IDEs. By smart IDEs, we mean IDEs that can learn and recognise inefficient developers' strategy and guide developers to follow efficient strategy. Our exploration strategies is the first step to model developers' behaviors. We first plan to analyse the relation between exploration strategies and developers' expertise to confirm or not the intuition that novice and expert developers may tend to follow different exploration strategies. Second, we want to improve our study and take into account other aspects, such as the structural relations between program entities and the semantic of program entities' names. Then, we can characterise developers' strategies and use various metrics (*e.g.,* effort, developers' experience, type of task) to build a model to identify efficient strategy (*e.g.,* exploration strategy recorded from more experienced successful developers). Therefore, we could mine ITs on the fly during maintenance tasks, use our model to identify strategies, and finally, if developers are getting lost, guide them to follow an efficient strategy. We believe that such IDEs can improve the efficiency of unexperienced developers by avoiding that they follow inefficient strategy.

End User Programming and Education

With the billion of open-source software systems to which end users have assess to the source code, we think that it could be interesting to allow end users to perform a minimum amount of maintenance and evolution tasks on the systems that they use. Interaction traces could provide great opportunities to demystify maintenance for end users. By collecting and making interaction traces available, we could build techniques including tools to replay ITs and assist end users in maintenance activities.

REFERENCES

Rajiv D. Banker and Srikant M. Datar and Chris F. Kemerer (1987). Factors affecting software maintenance productivity: An exploratory study. *Proceedings of the International Conference on Information Systems*. 160–175.

Bantelay, F. and Zanjani, M.B. and Kagdi, H. (2013). Comparing and combining evolutionary couplings from interactions and commits. *Reverse Engineering (WCRE), 2013 20th Working Conference on*. 311–320.

Bettenburg, Nicolas and Premraj, Rahul and Zimmermann, Thomas and Kim, Sunghun (2008). Extracting structural information from bug reports. *Proceedings of the 2008 international working conference on Mining software repositories*. 27–30.

Boehm, B.W. (1976a). Software engineering. *IEEE Trans. Computers*, *12*(25), 1226–1242.

Boehm, B.W. (1987). Improving software productivity. *Computer*, *20*(9), 43–57.

Boehm, Barry W. (1976b). Software engineering. *IEEE Trans. Computers*, *12*(c-25), 1226–1242.

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, *18*(6), 543–554.

Chawla, Nitesh V. and Bowyer, Kevin W. and Hall, Lawrence O. and Kegelmeyer, W. Philip (2002). Smote: Synthetic minority over-sampling technique. *J. Artif. Int. Res.*, *16*(1), 321–357.

Coman, I.D. and Sillitti, A. (2008). Automated identification of tasks in development sessions. *Proceedings International Conference on Program Comprehension*. 212–217.

DeLine, R. and Czerwinski, Mary and Robertson, G. (2005). Easing program comprehension by sharing navigation data. *Visual Languages and Human-Centric Computing, 2005 IEEE Symposium on*. 241–248.

Duchowski, Andrew (2007). *Eye tracking methodology: Theory and practice*. Springer-Verlag New York Inc.

Fjeldstad, R. K. and Hamlen, W. T. (1983). Application Program Maintenance Study: Report to Our Respondents. *Proceedings GUIDE 48*.

Fritz, Thomas and Murphy, Gail C. and Hill, Emily (2007). Does a programmer's activity indicate knowledge of code? *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 341–350.

Fritz, Thomas and Ou, Jingwen and Murphy, Gail C. and Murphy-Hill, Emerson (2010). A degree-of-knowledge model to capture source code familiarity. *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*. 385–394.

Emden R. Gansner and Stephen C. North (2000). An open graph visualization system and its applications to software engineering. *Software - Practice and Experience*, *30*(11), 1203–1233.

Giger, Emanuel and Pinzger, Martin and Gall, Harald (2010). Predicting the fix time of bugs. *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*. RSSE' 10, 52–56.

González, Victor M. and Mark, Gloria (2004). "constant, constant, multi-tasking craziness": Managing multiple working spheres. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '04, 113–120.

Hewett, Rattikorn and Kijsanayothin, Phongphun (2009). On modeling software defect repair time. *Empirical Softw. Eng.*, *14*(2), 165–186.

J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek (2006). Appropriate statistics for ordinal level data: Should we really be using t-test and cohen's d for evaluating group differences on the nsse and other surveys? Annual meeting of the Florida Association of Institutional Research.

Joseph L., Fleiss (1971). Measuring nominal scale agreement among many raters. *Psychological Bulletin*, *76*(5), 378–382.

Kamei, Y. and Monden, A. and Matsumoto, S. and Kakimoto, T. and Matsumoto, K.-i. (2007). The effects of over and under sampling on fault-prone module detection. *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*. 196–204.

Kersten, Mik and Murphy, Gail C. (2005). Mylar: a degree-of-interest model for ides. *Proceedings of the 4th International Conference on Aspect-oriented Software Development*. AOSD '05, 159–168.

Kersten, Mik and Murphy, Gail C. (2006). Using task context to improve programmer productivity. *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. 1–11.

Ko, Andrew J. and Myers, Brad A. and Coblenz, Michael J. and Aung, Htet Htet (2006). An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transaction on Software Engineering*, *32*(12), 971–987.

Lamkanfi, A. and Demeyer, S. and Giger, E. and Goethals, B. (2010). Predicting the severity of a reported bug. *Proceedings of International Workshop on Mining Software Repositories*. 1–10.

Landis, J. Richard, and Koch, Gary G. (1977). The measurement of observer agreement for categorical data. *Biometrics*, *33*, 159–174.

Lawrance, J. and Bogart, C. and Burnett, M. and Bellamy, R. and Rector, K. and Fleming, S.D. (2013). How programmers debug, revisited: An information foraging theory perspective. *Software Engineering, IEEE Transactions on*, *39*(2), 197–215.

Layman, Lucas M. and Williams, Laurie A. and St. Amant, Robert (2008). Mimec: Intelligent user notification of faults in the eclipse ide. *Proceedings of the 2008 International Workshop on Cooperative and Human Aspects of Software Engineering*. CHASE ’08, 73–76.

Lee, Seonah and Kang, Sungwon (2011). Clustering and recommending collections of code relevant to tasks. *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. 536–539.

Seonah Lee and Sungwon Kang (2013). Clustering navigation sequences to create contexts for guiding code navigation. *Journal of Systems and Software*.

Lee, S. and Kang, S. and Kim, S. and Staats, M. (2015). The impact of view histories on edit recommendations. *Software Engineering, IEEE Transactions on*, *41*(3), 314–330.

Lehman, M. M. and Belady, L.A. (1985). *Program Evolution: Processes of Software Change*. APIC Studies in Data Processing.

S. Letovsky (1986). Cognitive processes in program comprehension. *Empirical studies of programmers*. 58–79.

D.C. Littman and J. Pinto and S. Letosky and E. Soloway (1986). Mental models and software maintenance. *Empirical Studies of Programmers*. 80–98.

Maalej, Walid and Fritz, Thomas and Robbes, Romain (2014a). *Recommendation Systems in Software Engineering*, Springer, chapitre Collecting and Processing Interaction Data for Recommendation Systems.

Maalej, Walid and Tiarks, Rebecca and Roehm, Tobias and Koschke, Rainer (2014b). On the comprehension of program comprehension. *ACM Transactions on Software Engineering and Methodology*, *23*(4), 31:1–31:37.

Martínez-Torres, M. R. and Toral, S. L. and Barrero, F. and Cortés, F. (2010). The role of internet in the development of future software projects. *Internet Research*, *20*(1), 72–86.

Maxwell, K.D. and Forselius, P. (2000). Benchmarking software development productivity. *Software, IEEE*, *17*(1), 80–88.

Minelli, R. and Mocci, A. and Lanza, M. (2015). I know what you did last summer - an investigation of how developers spend their time. *Program Comprehension (ICPC), 2015 IEEE 23rd International Conference on*. 25–35.

Minelli, R. and Mocci, A. and Lanza, M. and Kobayashi, T. (2014). Quantifying program comprehension with interaction data. *Quality Software (QSIC), 2014 14th International Conference on*. 276–285.

Mordal, Karine and Anquetil, Nicolas and Laval, Jannik and Serebrenik, Alexander and Vasilescu, Bogdan and Ducasse, Stéphane (2012). Software quality metrics aggregation in industry. *Journal of Software: Evolution and Process*.

Murphy, Gail C. and Kersten, Mik and Findlater, Leah (2006). How are java software developers using the eclipse IDE? *IEEE Software*, *23*(4), 76–83.

Nadeem Ahsan, Syed and Ferzund, Javed and Wotawa, Franz (2009). Program file bug fix effort estimation using machine learning methods for oss. *Software Engineering and Knowledge Engineering*. 129–134.

Negara, Stas and Vakilian, Mohsen and Chen, Nicholas and Johnson, Ralph E. and Dig, Danny (2012). Is it dangerous to use version control histories to study source code evolution? *26th European Conference on Object-Oriented Programming (ECOOP)*.

Panjer, Lucas D. (2007). Predicting eclipse bug lifetimes. *Mining Software Repositories, 2007. ICSE Workshops MSR '07. Fourth International Workshop on*. 29–.

Parnin, C. and Görg, C. (2006). Building usage contexts during program comprehension. *Proceedings International Conference on Program Comprehension*. 13–22.

Parnin, Chris and Rugaber, Spencer (2011). Resumption strategies for interrupted programming tasks. *Software Quality Journal*, *19*(1), 5–34.

Pennington, N. (1987). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, *19*(295–341).

Robbes, R. and Lanza, M. (2010). Improving code completion with program history. *Automated Software Engineering*, *17*(2), 181–212.

Robbes, Romain and Röthlisberger, David (2013). Using developer interaction data to compare expertise metrics. *Proceedings of the 10th Working Conference on Mining Software Repositories*. 297–300.

Robillard, M.P. and Walker, R.J. and Zimmermann, T. (2010). Recommendation systems for software engineering. *Software, IEEE*, *27*(4), 80–86.

Robillard, Martin P. and Coelho, Wesley and Murphy, Gail C. (2004). How effective developers investigate source code: An exploratory study. *IEEE Transactions on Software Engineering*, *30*(12), 899–903.

Röthlisberger, D. and Nierstrasz, O. and Ducasse, S. (2011). Smartgroups: Focusing on task-relevant source artifacts in IDEs. *Proceedings International Conference on Program Comprehension*. 61–70.

Sanchez, Heider and Robbes, Romain and Gonzalez, Victor M. (2015). An empirical study of work fragmentation in software evolution tasks. *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. 251–260.

Schneider, K. A. and Gutwin, C. and Penner, R. and Paquette, D. (2004). Mining a softare developer's local interaction history. *In Proceedings of the International Workshop on Mining Software Repositories*.

Sharafi, Zohreh and Soh, Zéphyrin and Guéhéneuc, Yann-Gaël (2015). A systematic literature review on the usage of eye-tracking in software engineering. *Inf. Softw. Technol.*, *67*(C), 79–107.

Shihab, Emad and Kamei, Yasutaka and Adams, Bram and Hassan, Ahmed E. (2013). Is lines of code a good measure of effort in effort-aware models? *Inf. Softw. Technol.*, *55*(11), 1981–1993.

B. Shneiderman and R. Mayer (1979). Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer and Information Sciences*, *8*(3), 219–238.

Sillito, Jonathan and Murphy, Gail C. and Volder, Kris De (2008). Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, *34*(4), 434–451.

Singer, J. and Elves, R. and Storey, M. A. (2005). Navtracks: Supporting navigation in software maintenance. *Proceedings of the 21st IEEE International Conference on Software Maintenance*. 325–334.

Singer, Janice and Lethbridge, Timothy and Vinson, Norman and Anquetil, Nicolas (1997). An examination of software engineering work practices. *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*. CASCON '97, 21–.

Soh, Z. and Khomh, F. and Guéhéneuc, Y.-G. and Antoniol, G. and Adams, B. (2013). On the effect of program exploration on maintenance tasks. *Reverse Engineering (WCRE), 2013 20th Working Conference on*. 391–400.

Sommerville, Ian (2011). *Software Engineering*. PEARSON, ninth édition.

Song, Qinbao and Shepperd, Martin and Cartwright, Michelle and Mair, Carolyn (2006). Software defect association mining and defect correction effort prediction. *IEEE Trans. Softw. Eng.*, *32*(2), 69–82.

Margaret-Anne Storey (2006). Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal*, *14*(3), 187–208.

Tan, Pang-Ning and Steinbach, Michael and Kumar, Vipin (2006). *Introduction to Data Mining*, Pearson, chapitre 6: Association Analysis: Basic Concepts and Algorithms.

Vasa, R. and Lumpe, M. and Branch, P. and Nierstrasz, O. (2009). Comparative analysis of evolving software systems using the gini coefficient. *Software Maintenance (ICSM), IEEE International Conference on*. 179–188.

A. von Mayrhauser and A. M. Vans (1995). Program comprehension during software maintenance and evolution. *Computer*, *28*(8), 44–55.

Jinshui Wang and Xin Peng and Zhenchang Xing and Wenyun Zhao (2013). How developers perform feature location tasks: a human-centric and process-oriented exploratory study. *Journal of Software: Evolution and Process*, 1193–1224.

Weiss, Cathrin and Premraj, Rahul and Zimmermann, Thomas and Zeller, Andreas (2007). How long will it take to fix this bug? *Proceedings of the Fourth International Workshop on Mining Software Repositories*. MSR'07, 1–.

Wohlin, C. and Runeson, P. and Höst, M. and Ohlsson, M. C. and Regnell, B. and Wesslén, A. (2000). *Experimentation in Software Engineering - An Introduction*. Kluwer Academic Publishers.

Xu, K. (2004). How has the literature on gini's index evolved in the past 80 years? Rapport technique, Department of Economics, Dalhouse University, Halifax, Nova Scotia.

R. K. Yin (2002). *Case Study Research: Design and Methods - Third Edition*. SAGE Publications, London.

Ying, A.T.T. and Robillard, M.P. (2011). The influence of the task on programmer behaviour. *Proceedings International Conference on Program Comprehension*. 31–40.

Zanjani, Motahareh Bahrami and Swartzendruber, George and Kagdi, Huzefa (2014). Impact analysis of change requests on source code based on interaction and commit histories. *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014, 162–171.

Feng Zhang and Foutse Khomh and Ying Zou and Ahmed E. Hassan (2012). An empirical study of the effect of file editing patterns on software quality. *Proceedings Working Conference on Reverse Engineering*. 456–465.

Thomas Zimmermann and Peter Weiβgerber and Stephan Diehl and Andreas Zeller (2005). Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, *31*(6), 429–445.

Lijie Zou and Godfrey, M.W. and Hassan, A.E. (2007). Detecting interaction coupling from task interaction histories. *Proceedings International Conference on Program Comprehension*. 135–144.