On the Effect of Program Exploration on Maintenance Tasks

Zéphyrin Soh[1,3], Foutse Khomh[2], Yann-Gaël Guéhéneuc[1]
Giuliano Antoniol[3], Bram Adams[4]
[1]Ptidej Team, [2]SWAT, [3]Soccer Lab, [4]MCIS
DGIGL, École Polytechnique de Montréal, Canada
Email: {zephyrin.soh, foutse.khomh, yann-gael.gueheneuc, giuliano.antoniol, bram.adams}@polymtl.ca

***Abstract*—When developers perform a maintenance task, they follow an exploration strategy (ES) that is characterised by how they navigate through the program entities. Studying ES can help to assess how developers understand a program and perform a change task. Various factors could influence how developers explore a program and the way in which they explore a program may affect their performance for a certain task. In this paper, we investigate the ES followed by developers during maintenance tasks and assess the impact of these ES on the duration and effort spent by developers on the tasks. We want to know if developers frequently revisit one (or a set) of program entities (referenced exploration), or if they visit program entities with almost the same frequency (unreferenced exploration) when performing a maintenance task. We mine 1,705 Mylyn interaction histories (IH) from four open-source projects (ECF, Mylyn, PDE, and Eclipse Platform) and perform a user study to verify if both referenced exploration (RE) and unreferenced exploration (UE) were followed by some developers. Using the Gini inequality index on the number of revisits of program entities, we automatically classify interaction histories as RE and UE and perform an empirical study to measure the effect of program exploration on the task duration and effort. We report that, although a UE may require more exploration effort than a RE, a UE is on average 12.30% less time consuming than a RE.**

***Index Terms*—Software Maintenance, Program Exploration, Interaction Histories, Exploration Strategies, Mylyn**

I. Introduction

Software systems must be maintained and evolved to fix bugs and adapt to new technologies and requirements. When developers perform a maintenance task, they always need to explore the program, *i.e.*, navigate through the entities of the program. The purpose of this program navigation is to find the subset of program entities that are relevant to the maintenance task. Program exploration involves three activities [9], [23]: looking at the initial entity that seems relevant (starting point), relating the starting point to other entities and exploring them (expanding the starting point), and 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 [9], developers' experience and proficiency, tool support, and the software design. The strategy may affect the successfulness of maintenance tasks [19], as well as the time and effort spent to perform a task; hence the developers' productivity. Thus, studying exploration strategies can help to (1) evaluate developers' performance, *e.g.*, find if there is an "efficient" way to explore a program; (2) improve our knowledge on developers' comprehension process, *e.g.*, a top-down or bottom-up comprehension can be related to a specific strategy; (3) 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; (4) 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 histories 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 histories (IH) from four open-source projects (ECF, Mylyn, PDE, and Eclipse Platform). From the IH 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 IHs into referenced exploration (RE) and unreferenced exploration (UE). Next, based on the manually classified IHs, we automatically classify the remaining IHs 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 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 effort a developer spends on a task, the more he is likely to follow the UE strategy.

The remainder of this paper is organised as follows: Section II provides some background and the description of the data used in this paper. Section III describes our user study to investigate the exploration strategy and automatically identify them. Section IV describes our empirical study of the relation between the exploration strategy and the time and effort spent performing a maintenance task. Section V discusses the threats to the validity of our results. We relate our work to previous work in Section VI while Section VII summarises our findings and highlights some avenues for future work.

II. Background and Data

Data used in this paper were collected using the Mylyn plugin. In this section, we present some background information on Mylyn and describe our data collection and processing.

A. Mylyn Plugin

Mylyn is an Eclipse plugin that captures developers' interactions with program entities when performing a task. Each developers' action on a program entity is recorded as an *event*. There are eight types of *events* in Mylyn: *Attention*, *Command*, *Edit*, *Manipulation*, *Prediction*, *Preference*, *Propagation*, and *Selection* [15]. The list of interaction events triggered by a developer form an interaction history (IH). An interaction history is therefore a sequence of interaction events that describe accesses and operations performed on program entities [8]. Interaction histories logs are stored in an XML format. Each interaction history 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, in this paper, we consider only developer's interaction events: *Selection*, *Edit*, *Command*, and *Preference*.

Table I: Descriptive statistics of the data (IH: Interaction History)

	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 $\leq$ 2 classes	27	285	204	45	561
IH class level	26	1,273	131	275	1,705

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

Figure 1: The structure of a Java *StructureHandle*

B. Data Collection

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 interaction history for each bug. Interaction histories related to a bug are attached to the bug report. We extract the interaction histories ID of all the attachments with the name "mylyn-context.zip", which is the default name given by Mylyn to interaction histories. We download, clean (*i.e.*, remove interaction histories that have at least one event with negative duration, or that have zero duration), and retain 2,375 interaction histories. We removed interaction histories 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 history involving two classes. Overall, we kept 1,705 interaction histories. Table I presents a description of the data set. All data used in this paper are available online[1].

C. Data Parsing

We parse the interaction histories to extract useful data. 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). In this paper, we consider only Java *StructureHandle*. 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.* [1] to identify parts of stack traces contained in bug reports. Figure 1 shows the structure of a Java *StructureHandle* (we use [*] to ignore the rest of the *StructureHandle*.) More details about the parser used in this paper can be find in our technical report [25].
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,

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

Table II: 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

we focus on the class level in the remainder of this paper. The results for the file level can be found in our technical report [25].

III. Empirical User Study of Exploration Focus

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 paper 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.

A. *User Study*

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

(1) We choose the interaction history sample size to achieve a 95%$\pm$10 confidence level. 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 histories in our sample (instead of two interaction histories as suggested by the sample size). After the sampling, we observed that our sample contained program entities from multiple verisons of each of our subject projects. Table II presents the size of the sub-samples of the projects.

(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 history 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 [5] representation of the interaction histories, *i.e.*, the exploration graph. Grapviz (http://www.graphviz.org) 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 histories). 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 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 histories and classify them into referenced exploration (RE) and unreferenced exploration (UE). In case of doubt, they were required to label the exploration history with a D. 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: (i) 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? (ii) 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 histories with less than 2/3 of either RE or UE labels to be undecided cases, *e.g.*, the interaction history #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 II shows the results of the user study. In total, 88.45% of interaction histories 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 histories 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 [6] 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 [10], 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.

B. 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.

1) 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 [12], [13], [26] and (2) the mathematical properties of the Gini inequality index presented by Mordal *et al.* [13] 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 history. The set of program entities involved in an interaction history 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 [28] presented many computational approaches for the Gini inequality index and mentioned that theses approaches are consistent with one another. As used in [12], [13], 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):

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

2) 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.

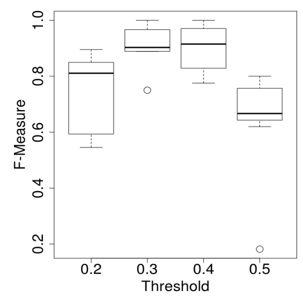


Figure 2: F-Measure per threshold for the oracle

- 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).

3) Identification Threshold: We use the oracle build in Section III-A (*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 III-A. 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:

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

Figure 2 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 2 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 in the remainder of the paper.

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

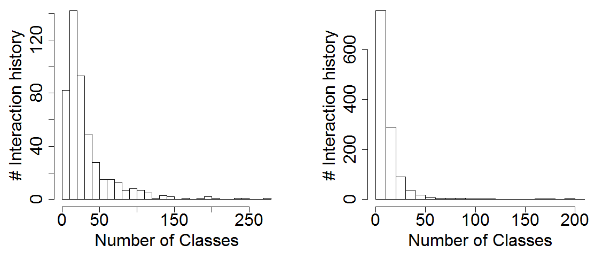
Observation 1*: Developers follow mostly the unreferenced exploration (UE) when performing a maintenance task.*

Observing that there are more UE than RE, we look at the frequency of the number of classes involved in the IH (See Figure 3b for UE and Figure 3a for RE). There are 711 UE interaction histories (57.94%) in which less than 10 classes are involved. For RE, there are 71 interactions histories (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.* [19] observed that methodical developers do not reinvestigate methods as frequently as opportunistic developers. Methodical developers seem to answer specific questions using

Table III: 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			



(a) Referenced Exploration (b) Unreferenced Exploration

Figure 3: Frequency of the number of classes involved in the interaction histories

focussed searches, while opportunistic developers guess more and read the source code in details [19]. 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.

IV. Empirical Study

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 effort between referenced exploration (RE) and unreferenced exploration (UE)?

The corresponding null hypotheses are:

$H_{0_{Time}}$: There is no difference in the average time spent between RE and UE when developers perform a maintenance task.

$H_{0_{Effort}}$: 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 histories. 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.*, $\alpha = 0.05$).

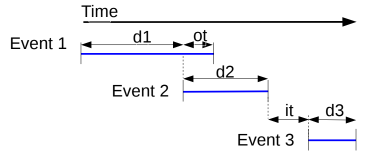


Figure 4: Interruption and overlap between events

A. Metrics

We compute the following metrics on the IH:

- *Overall duration* is the duration of an interaction history (IH). We use the *StartDate* and *EndDate* of an event to compute the duration of the event *i.e.*, $Duration(anEvent) = EndDate(anEvent) - StartDate(anEvent)$. Sometimes we can have both interruption and overlap between events. We sort the interaction events by *StartDate* and compute duration of IH after considering interruption and overlap. For example consider Figure 4, which shows three events with an overlap between *event1* and *event2* and an interruption between *event2* and *event3*. We handle interruptions and overlaps by considering the overall duration spent on three events as *d1+d2+d3*, the overlap time is *ot*, and the interruption time as *it*. To control the confounding effect of the number of entity involved in an interaction history, we divide the overall duration by the total number of entity involved in an IH to obtain the average overall duration.
- *Class level duration* is a cumulative duration spent on entities at class level in an interaction history. 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 IH, we divide the class level duration by the total number of class involved in an IH 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 IH. The *number of edit* ($NumEdit$) is the total number of edit events in an IH. $anEvent$ is an edit event if $kind(anEvent) =$ "$Edit$" (see Section II-A)

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

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

The exploration effort measures the effort spent by a developer to find the relevant program entities to edit. Röthlisberger *et al.* [20] 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.

B. Results and Discussions

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

In **RQ1** (Section III), 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 III 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 IH**. For the ECF project, this does not hold at the class level, possibly because there are only 26 interaction histories.

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 5 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.* [19] 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.*'s one [19].

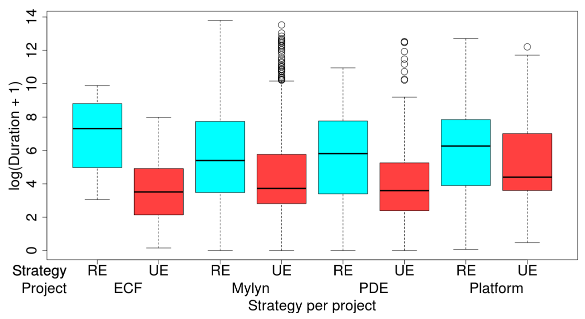


Figure 5: Distribution of overall duration per project

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 analyze this, we compute the number of working days for each interaction history. When we study the percentage of interaction histories per number of working days, Table IV shows that developers work for one or two days on about 75% of interaction histories and more than two days on about 25% of interaction histories. Even with this unbalanced proportion, the distribution of the logarithm of duration (see Figure 6) 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. 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, they refresh their mind by (re)exploring the core 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 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 IV-A), a low value of *EditRatio* indicates a small number of edit events and a high

Table IV: Percentage of interaction history 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

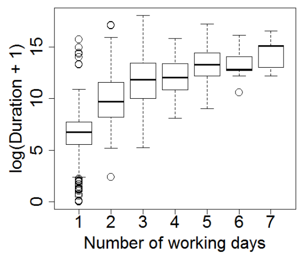


Figure 6: Distribution of duration per number of working days

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 III, 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 8 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 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 history modifications (edit events) and commit modifications from the source code repository to compare real modifications of the source code with revert/cancelled modifications that we expect to be frequent with RE.

C. Confounding Factors

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

1) 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 IH), we did not observe an impact of the systems on the ES. However, developers explore the program by following different kind of relationships [24]. We conjecture

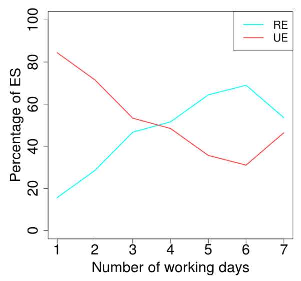


Figure 7: Percentage of exploration per number of working days

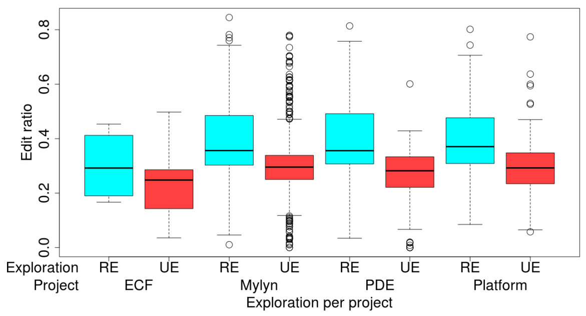


Figure 8: Distribution of effort per project

that the architecture of the program can affect the exploration strategy. By definition, the program entities involved in an IH are the part (architecture) of the system used to perform a task. Thus, if the exploration is guided by architecture, the IHs using almost the same part of a system will result in the same exploration strategy. For example, if two IHs 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 histories. Consider three IHs 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 histories. 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**.

2) Task Interruption and Switching: **Task interruption** is a common problem when developers perform a task. Zhang *et al.* [31] found that task interruption increases the risk of bugs in

Table V: 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%)

files while Parnin and Rugaber [18] 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 histories, Mylyn requires developers to activate the task they are working on. The task ID is unique and appears in the interaction history 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.* [9] 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.

3) 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 and Robillard [30] 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 V show no significant relation between the type of a task and the exploration strategy of developers.

V. 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 [29] of empirical studies.

A. 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 depends on the way Graphviz displays the exploration graphs. We have no control on Graphviz, however, all subjects worked on the same displayed graphs. For threats related to our metrics; because some developers can partly record their interaction histories, we think that for some cases the time recorded can be different from the "real" time spent. We plan to perform an experiment and collect data to investigate this threat.

B. 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.

C. Internal Validity

Internal validity threat relates to the tools used to collect interaction histories and the choice of the projects. Many tools [16], [20] can collect developers' interactions with the IDE. We use Mylyn's interaction histories because (1) Mylyn is a tool provided as an Eclipse's plug-in and (2) all contributions to Mylyn must be made using Mylyn[2], *i.e.*, in contrast to other tools, the Mylyn interaction histories are available. Concerning the projects, because we use the Mylyn interaction histories, we are constrained to use projects that have Mylyn interaction histories available. Thus, we use the top four projects using Mylyn to gather developers' interactions.

D. External Validity

External validity threat concerns the generalization of our results. In our study, we used Mylyn interaction histories 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.

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

E. Reliability Validity

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 history 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.

VI. Related Work

Our work on exploration strategies is related to works on program exploration and mining of developer's interaction histories.

A. Program Exploration and Tools

The exploration of a program is related to the cognitive process of developers. Program comprehension theories explain how developers think (processes taking place in their mind) and use their knowledge [2], [22], [27]. Robillard *et al.* [19] studied the external behavior of developers. Our work differs to Robillard *et al.*'s work in the sense that (1) they compare the behavior of successful and unsuccessful developers and characterize them, while we study how developers move through program entities, (2) they perform the study in a lab setting with only five developers performing identical task on one system, while we use the data of multiple developers from four open-source projects, (3) they use a video capture of the screen while we consider the interaction histories gathered by Mylyn. Due to the lack of capability to access the successfulness of developers from the bug report, our work provides a partial validation of Robillard *et al.* [19] findings. We plan to investigate the success dimension by performing an experiment using Mylyn.

Ko *et al.* [9] also investigate how developers explore a program. Our work is different to Ko *et al.* [9] work because they look at the main activities taking place when developers perform a task, they restrict the task to perform and limit the experiment time. They provide an unfamiliar program to the developers, without documentation nor comments in the program. They also use the screen-capture video and manually simulate the interruption.

Lawrance *et al.* [11] show that Information Foraging Theory (IFT) can be used to assess developers' behaviour when searching relevant information. Compared to our study, they apply the IFT on debugging tasks and use verbal protocol (think-aloud) and screen captures to collect data about developers' behaviour.

Regarding the tools to gather interaction histories, the Mylyn plug-in was developed by Kersten *et al.* [8] to capture developers' interactions with program entities when they perform a task using Eclipse IDE. Later on, Röthlisberger *et al.* [20] implemented SmartGroups to complement Mylyn with evolutionary and dynamic information. Similar to Mylyn, CodingTracker [16] is another Eclipse plug-in that records developers' interactions with program entities. However, to date, only few projects have adopted CodingTracker. Moreover, we couldn't find developers' interaction logs from CodingTracker in any open-source version control system that we examined. Hence our choice of Mylyn for this study. Interaction history data collected from multiple Eclipse projects, using Mylyn are publicly available in the bug report system of Eclipse.

B. Mining Interaction Histories

Interaction history logs have been used by the research community to study developers' programming behaviors and propose new tools to ease their daily activities.

Zou *et al.* [32] use 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. Kersten and Murphy [7] use developers' frequency and recency interaction with an entity to propose a degree-of-interest (DOI) model. The model is used to built the task context that help to reduce the developers' information space. Fritz *et al.* [4] found that the DOI indicates the developers' knowledge about the structure of the code. While developers can work on many tasks at a time, Coman and Sillitti [3] use 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 [18], Parnin and Görg [17] count the number of prior consecutive interactions on an entity to extract the usage context of the task when it has been interrupted. Schneider *et al.* [21] investigated the benefits of tracking developers' local interactions history when developing in a distributed environment. Murphy *et al.* [14] mined Mylyn interaction history logs collected from 41 programmers and observed that some views of the Eclipse IDE were more useful than others. Mylyn interaction histories are also used to find developers editing styles/patterns [30], [31]. Most previous studies on Mylyn interaction histories only considered the *kinds* of the Mylyn *events*. In this work, we investigate Mylyn *events* in more details by looking at the type of program entities on which an *event* occurred. Moreover, instead of counting the number of events and others metrics used in the previous work, we use the distribution of the number of revisits to study the exploration strategy.

VII. Conclusion and Future Work

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. In this paper we contribute to the understanding of developers' exploration strategies in two ways. First, after mining Mylyn's interaction histories, 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.*, IHs) 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 histories 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 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.*'s results [19].

In the future, we plan to analyze 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. We also plan to implement a tool that will monitor developers' explorations and guide them using *best* exploration strategies recorded from more experienced/successful developers. We believe that such a tool can improve the efficiency of unexperienced developers by avoiding that they follow time/effort consuming program exploration strategies.

Acknowledgment

This work has been partly funded by the Canada Research Chairs on Software Patterns and Patterns of Software and on Software Change and Evolution. We also thanks the subjects for the manual classification of exploration strategy.

References

[1] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Extracting structural information from bug reports. In *Proceedings MSR*, pages 27–30, 2008.

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

[3] I. Coman and A. Sillitti. Automated identification of tasks in development sessions. In *Proceedings ICPC*, pages 212–217, 2008.

[4] T. Fritz, G. C. Murphy, and E. Hill. Does a programmer's activity indicate knowledge of code? In *Proceedings ESEC-FSE '07*, pages 341–350, 2007.

[5] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software - Practice and Experience*, 30(11):1203–1233, 2000.

[6] F. Joseph L. Measuring nominal scale agreement among many raters. *Psychological Bulletin*, 76(5):378–382, 1971.

[7] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for ides. In *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, AOSD '05, pages 159–168, 2005.

[8] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT/FSE*, pages 1–11, 2006.

[9] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. 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, dec 2006.

[10] J. R. Landis and G. G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33:159–174, 1977.

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

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

[13] K. Mordal, N. Anquetil, J. Laval, A. Serebrenik, B. Vasilescu, and S. Ducasse. Software quality metrics aggregation in industry. *Journal of Software: Evolution and Process*, 2012.

[14] G. C. Murphy, M. Kersten, and L. Findlater. How are java software developers using the eclipse IDE? *IEEE Software*, 23(4):76–83, July 2006.

[15] Mylyn. http://wiki.eclipse.org/mylyn_integrator_reference.

[16] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig. Is it dangerous to use version control histories to study source code evolution? In *26th European Conference on Object-Oriented Programming (ECOOP)*, 2012.

[17] C. Parnin and C. Görg. Building usage contexts during program comprehension. In *Proceedings ICPC*, pages 13–22, 2006.

[18] C. Parnin and S. Rugaber. Resumption strategies for interrupted programming tasks. *Software Quality Journal*, 19(1):5–34, 2011.

[19] M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Transactions on Software Engineering*, 30(12):899–903, December 2004.

[20] D. Röthlisberger, O. Nierstrasz, and S. Ducasse. Smartgroups: Focusing on task-relevant source artifacts in IDEs. In *Proceedings ICPC*, pages 61–70, june 2011.

[21] K. A. Schneider, C. Gutwin, R. Penner, and D. Paquette. Mining a softare developers local interaction history. In *Proceedings MSR*, 2004.

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

[23] J. Sillito, G. C. Murphy, and K. D. Volder. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, 34(4):434–451, July/August 2008.

[24] J. Singer, R. Elves, and M. A. Storey. Navtracks: Supporting navigation in software maintenance. In *Proceedings ICSM*, pages 325–334, 2005.

[25] Z. Soh and Y.-G. Guéhéneuc. Towards the exploration strategies by mining mylyns' interaction histories. Technical Report EPM-RT-2013-01, École Polytechnique de Montréal, Feb. 2013.

[26] R. Vasa, M. Lumpe, P. Branch, and O. Nierstrasz. Comparative analysis of evolving software systems using the gini coefficient. In *Proceedings ICSM*, pages 179–188, sept. 2009.

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

[28] K. Xu. How has the literature on gini's index evolved in the past 80 years? Technical report, Department of Economics, Dalhouse University, Halifax, Nova Scotia, Dec. 2004.

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

[30] A. Ying and M. Robillard. The influence of the task on programmer behaviour. In *Proceedings ICPC*, pages 31–40, june 2011.

[31] F. Zhang, F. Khomh, Y. Zou, and A. E. Hassan. An empirical study of the effect of file editing patterns on software quality. In *Proceedings WCRE*, pages 456–465, 2012.

[32] L. Zou, M. Godfrey, and A. Hassan. Detecting interaction coupling from task interaction histories. In *Proceedings ICPC*, pages 135–144, 2007.