Towards Understanding How Developers Spend Their Effort During Maintenance Activities

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

***Abstract*—For many years, researchers and practitioners have strived to assess and improve the productivity of software development teams. One key step toward achieving this goal is the understanding of factors affecting the efficiency of developers performing development and maintenance activities. In this paper, we aim to understand how developers' spend their effort during maintenance activities and study the factors affecting developers' effort. By knowing how developers' spend their effort and which factors affect their effort, software organisations will be able to take the necessary steps to improve the efficiency of their developers, for example, by providing them with adequate program comprehension tools. For this preliminary study, we mine 2,408 developers' interaction histories and 3,395 patches from four open-source software projects (ECF, Mylyn, PDE, Eclipse Platform). We observe that usually, the complexity of the implementation required for a task does not reflect the effort spent by developers on the task. Most of the effort appears to be spent during the exploration of the program. In 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 their task take a longer time to perform the task. We expect that the results of this study will pave the way for better program comprehension tools to guide developers during their explorations of software systems.**

***Index Terms*—Maintenance task, developers' effort, interaction history, patch, change complexity.**

I. Introduction

Over the past decades, maintenance has become the most time and resource consuming activity in the life cycle of software systems. It is estimated that 80% of software development costs are spent on maintenance [12]. When performing a maintenance task, developers spend a certain effort exploring the program, finding relevant entities, understanding and making changes to the program [8]. The cost of each of these developers' activities has a direct impact on the overall cost of the maintenance of software systems.

Despite the large body of work on software productivity [1], [3], [11] there are very few studies that empirically investigated how developers spent their effort during software maintenance activities. The relationship 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. To perform such a study, one needs detailed information about file editing activities occurring in developers' workspaces. A tool such as Mylyn which records and monitors developer's programming activities, like the selection and the editing of files, provides the opportunity for such a study. Indeed, the effort spent by a developer when performing a task can be estimated from interaction logs recorded with Mylyn (*i.e.*, the developer's interaction with program entities).

After performing a task, developers commit their changes in a repository or provide them as a patch in a code review system (*e.g.*, Gerrit) or an issue tracking system (*e.g.*, Bugzilla). 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 performed on these entities. These changes are the result of the effort spent when performing a task.

In this paper, we analyze developers' interaction logs (*i.e.*, logs recording developers' selection and editing of files) 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 history logs with patches (*i.e.*, identify the patch that is the implementation of a given interaction history). We find that the effort spent by a developer when performing a task is not correlated to the complexity of the implementation of the task (*i.e.*, the patch).

RQ2: *How do developers spend their effort? What are the factors affecting developers' effort?*

The files in a patch are the significant relevant files because they are changed to perform a task (*e.g.*, fixing a bug). The files in the interaction logs are files explored by developers. The files that are explored, but not changed, are the additional files that developers used when performing their tasks. 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. We also find that while the bug severity indicates the complexity of the implementation of the task (*i.e.*, the patch), the impact of bug severity on developers' effort is project dependant. Finally, our study reveals that developers' experience does not reduce their effort; we observe that when a program evolves, developers perform more tasks on the parts (of the program) where they have no experience.

The paper is organized as follow: Section II describes the data that we use in this paper. Sections III and IV respectively address our two research questions **RQ1** and **RQ2** by describing their motivations, approaches, and results. After presenting previous work in Section V, we discuss the threats to the validity of our study in Section VI. Section VII concludes the paper and outlines some avenues for future work.

II. Data Collection and Processing

We download the bug reports of the subjects projects, then we parse them and extract the interactions histories and patches ID. There are some bug reports without interaction histories or patches. Interactions and patches are associated to a bug report as attachments. The interaction histories attached to the subjects projects are Mylyn's logs. In this section, we give some background information about the Mylyn plugin, then we explain how we collect and process the data.

A. Backgound

Mylyn is an Eclipse plugin that captures developers' interactions with program entities when performing a task [7]. Each developers' action on a program entity is recorded as an *event*. The list of interaction events triggered by a developer form an interaction history. Mylyn records the interaction history when the developers activate the working task. When the developers' deactivate the working task, Mylyn stops gathering the interaction history. For the sake of simplicity, we use "interaction" instead of "interaction history" in the remainder of the paper.

B. Interaction

To obtain interactions data, we download the bug reports' attachments with the name "mylyn-context.zip", which is the default name given by Mylyn to interactions. We parse the interactions to extract the program entities on which the events occurred and the time spent on each entity [17]. 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. In this paper, we use the word "file" to name the Java files and resource entities involved in an interaction. Table I presents the number of interactions per project. Without distinguishing among projects, an interaction involves on average 46.87 files (standard deviation 220.05).

Table I: 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

C. Patch

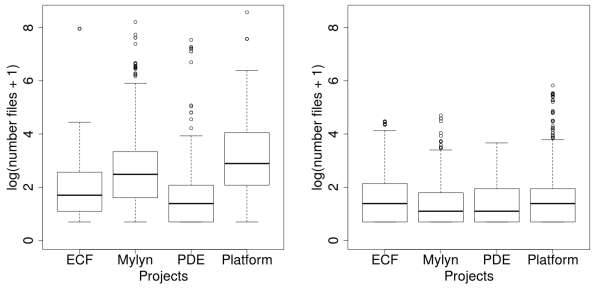
We download bug reports' attachments with the attribute "ispatch" (of the tag "attachment") equal to one, that identify the patch. A patch can involve many files. For each file involved in a patch, the changes made in the file can be grouped into many *deltas*. A *delta* contains a snippet code before (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.

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 I. Without distinguishing among projects, a patch involves 6.32 files (standard deviation = 18.57). The raw interactions and patches can be downloaded from Eclipse Bugzilla[2], and the processed data that we used in this paper can be found online[3].

D. 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 1 presents the comparison between the files involved in the interactions (Figure 1a) and patches (Figure 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

[1] http://code.google.com/p/java-diff-utils/
[2] https://bugs.eclipse.org/bugs/
[3] http://www.ptidej.net/download/experiments/wcre13b/



(a) Files in interactions (b) Files in patches

Figure 1: Distribution of logarithm of the number of files involved in the interactions and patches

the files that must be modified. We suspect that the exploration of these additional files may affect developers' effort.

III. 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.

A. 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.

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 III-B). We discuss the approach in Section III-C and Section III-D discusses the obtained results.

B. Metrics

1) 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 sort interaction events by *StartDate* and compute the time spent which is the sum of the duration on each interaction event (See Section II-A). We use the *StartDate* and *EndDate* of an event to compute the duration of the event after removing interruptions and overlaps time. In fact, we can have both interruptions and overlaps between events *i.e.*, for the interaction events *Event1* and *Event2* where *Event1* occurs before *Event2* ($StartDate(Event1) < StartDate(Event2)$), we may have $StartDate(Event2) - EndDate(Event1) \neq 0$ (interruption) or $EndDate(Event1) > StartDate(Event2)$ (overlap). 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[4]. We use the JGraphT[5] 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 ($k = 1$) 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.

2) 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

[4]http://mathworld.wolfram.com/Pseudograph.html
[5]http://jgrapht.org/

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 III-C using the data with comments and without comments, the removal of the comments did not affect our results.

C. 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.

Table II: 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)

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.

D. 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 II. 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 $P \not\subseteq I$ 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 III: Spearman correlation between the developers' effort and the complexity of the implementation

		Complexity of the implementation	
		Entropy	**Change distance**
Effort	**Time (sec.)**	0.16	0.27
	Cyclomatic	0.31	0.33

Table III shows that for all combinations of effort/implementation metrics, the effort spent by developers when performing a task is not 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**. The lack of correlation between the effort spent on a task and the complexity of the implementation of that task may also suggest that some of the effort spent by developers on a task do 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.

IV. How do developers spend their effort? What are the factors affecting developers' effort?

According to Lee and Kang [10], a significantly relevant entity is *a program entity that a developer needs to change in order to accomplish the task*. Therefore, the files in the patch are *significantly relevant files*. We use the expression *explored files* to name the files involved in an interaction. We use the term *additional files* to name explored files that are not significantly relevant. We chose the term *additional files* because these files can be important for the understanding of the program and the completion of the task (*i.e.*, *useful files*). These *additional files* can also be just *accidental files i.e.*, developers accidentally explored these files when they were looking for significantly relevant and–or useful files. Our goal in this research question is 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.

A. 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 [9]. 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 [9]. 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 [2]. 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.

B. 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 [13] 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 I) 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.

C. Approach

We compute the similarity between all interaction/patch that we matched in Section III-D. 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 IV-A, 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 IV-B. 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).

D. Results and Discussions

The similarity between the matched interactions and patches shows that some matchings have a low similarity (See Figure 2). Developers who attached interactions and patches with low similarity used more additional files. Since we removed unbalanced matchings as described in Section III-D, 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

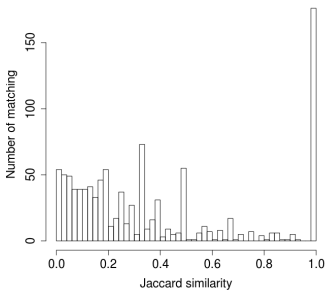
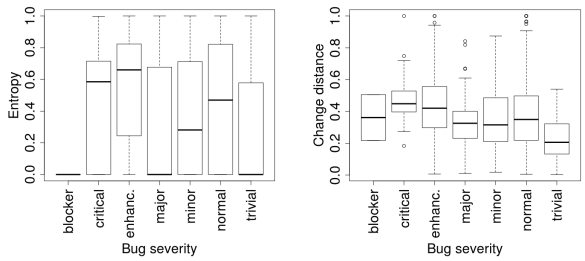


Figure 2: Similarity between matched interaction/patch

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. The Spearman correlation between the number of additional files and developers' effort are respectively 0.63 and 0.82 for the time spent and the complexity of exploration graph. Our result suggests 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.

The distribution of the complexity of the task implementations 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. The distribution of entropy and change distance for different bug severities is shown in Figure 3. Figure 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 3a) can require more changes than a *minor* bug (See Figure 3b). There is also a low correlation between the entropy and the change distance (Spearman coefficient = 0.27). This may indicate that the two metrics do not measure the same aspect of the complexity of the implementation. 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), 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



(a) Entropy (b) Change distance

Figure 3: Distribution of entropy and change distance per bug severity

Table IV: 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**

or (2) those who assign severities may also consider severities from the perspective of complexity.

Table IV 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.

Concerning developers' experience, Table V 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 VI 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 V) and the Cyclomatic complexity of exploration graphs (Table VI). For the Platform project, the relevant experience tend to increase the complexity of exploration graphs (correlations 0.55 and 0.57 in Table VI). 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

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

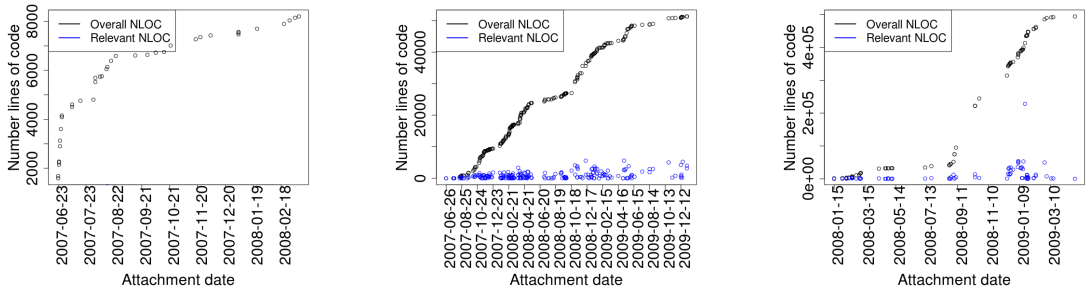
	NB	Overall Experience		Relevant Experience	
		NF	NLOC	NF	NLOC
ECF	0.17	0.03	**-0.12**	**-0.54**	**-0.60**
Mylyn	-0.08	**-0.10**	-0.07	0.34	0.31
PDE	**-0.27**	**-0.21**	0.04	0.31	0.28
Platform	0.18	0.24	0.28	0.33	0.39
All	-0.01	0.01	0.09	0.34	0.34

Table VI: 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

number of relevant 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 V), and -0.66, 0.26, 0.29, and 0.57 (for Cyclomatic complexity - See Table VI). 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 V and VI, 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 and Röthlisberger [14] 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 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 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 they also performed more modifications on files that they never used before (See Figure 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



(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 4: Developers perform more modifications on the files that they never used before

that they frequently used, they start performing the tasks that require some files that they never used before (See Figure 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.

V. Related Works

Our work is related to works that use developers' interactions and–or patches and source code repositories to study developers' effort, experience and the complexity of tasks.

A. Complexity of the tasks and bug severity vs. effort

The bug severity is used by many researchers to predict the lifetime of bugs [13] and study the re-opening of bugs [16]. Entropy measures have been used by Hassan [6] and Zaman *et al.* [18] to assess the complexity of the changes in the source code repositories and patches. These previous works have analyzed bug severities and the complexity of tasks separately. In this paper we introduce *the change distance metric* (*i.e.*, calculated using the Levenshtein distance) as another measure of the complexity of tasks and examine whether both the entropy and the change distance are related to bug severity. Serebrenik *et al.* [15] have used functions points to assess the complexity of projects. They examined the relation between the complexity of projects (in terms of number of functionalities) and development efforts and concluded that projects with similar amount of functionalities require different development efforts. Their work is related to our work in the sense that they aim to compare the complexity with the effort, but at a different level of granularity, *i.e.*, the project level in comparison to the task level investigated in this paper. They also used different metrics, *e.g.*, function points.

B. Developers' experience vs. effort

Feigenspan *et al.* [4] examined strategies used in empirical studies to control developers' experience. They argued that self estimations seems to be reliable ways to measure developers' experience. We were not able to used self estimations in our work since we do not have access to the developers of our studied projects. Also, in this study we were more interested in assessing the evolution of developers' experience *i.e.*, measure their experience at given dates (before they performed a task). Since most of these tasks were performed many years ago, developers who performed these tasks could hardly recall the experience that they had at the time of the task. Fritz *et al.* [5] combined developers' interactions and authorship information (from change history) to model source code familiarity, *i.e.*, the degree of knowledge. Robbes and Röthlisberger [14] also used developers' interactions to assess developers' effort and correlate the effort to experience. Our work differs from Robbes and Röthlisberger's work in two ways. First, we use different experience metrics mined from patches while they considered source code repositories and measured a developer's experience using the number of commits involving files explored by the developers. This approach is likely to be inaccurate since in many open source projects, some developers do not have commit privileges and have to submit their patches to reviewers, who revise them and commit.

VI. 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 mis-matchings *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 2,000 downloads per month[6]. We minimized the effect of the DiffUtils library by

[6]http://sourceforge.net/projects/jgrapht/files/stats/timeline?dates=2013-05-01+to+2013-06-01 (visited date 08/06/2013)

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. However, some developers may partly record their interactions. Thus, in some cases, the time recorded may be different from the time that they spent to perform the task. Our future controlled experiment will avoid the threat related to the time spent. 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 histories 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 histories and the choice of our subject projects. We used Mylyn's interaction histories 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 interaction histories.

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 software projects. In open source projects, developers are usually volunteers. In the future, we plan to analyze more projects, including proprietary projects and projects written in different programming languages, to draw more general conclusions.

Reliability validity: Reliability validity threats concern the possibility of replicating this study. All the raw data used in this paper are available on Eclipse Bugzilla. The projects studied in this paper are also available online for the public.

VII. Conclusion and Future Works

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. In this paper, we mined 2,408 developers' interaction histories and 3,395 patches from four open-source software projects (ECF, Mylyn, PDE, Eclipse Platform) and examined the factors affecting developers' effort.

We matched the interactions to the patches (*i.e.*, identify a patch that is the result of an interaction) and found 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. We expect that the results of this study will pave the way for better program comprehension tools to guide developers during their explorations of software projects.

In the future, we plan to mine the source code repositories of the software projects analyzed in this paper to assess the structural relations between the files involved in the interaction/patch matchings. With this information, we will build prediction models of *significantly relevant files* in order to recommend relevant files to developers performing maintenance tasks on the projects.

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.

References

[1] R. D. Banker, S. M. Datar, and C. F. Kemerer. Factors affecting software maintenance productivity: An exploratory study. In *Proceedings of the International Conference on Information Systems*, pages 160–175, 1987.

[2] B. Boehm. Software engineering. *IEEE Trans. Computers*, 12(25):1226–1242, 1976.

[3] B. Boehm. Improving software productivity. *Computer*, 20(9):43–57, 1987.

[4] J. Feigenspan, C. Kastner, J. Liebig, S. Apel, and S. Hanenberg. Measuring programming experience. In *Proceedings ICPC*, pages 73–82, 2012.

[5] T. Fritz, J. Ou, G. C. Murphy, and E. Murphy-Hill. A degree-of-knowledge model to capture source code familiarity. In *Proceedings ICSE*, pages 385–394, 2010.

[6] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings ICSE*, pages 78–88, 2009.

[7] 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.

[8] 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.

[9] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals. Predicting the severity of a reported bug. In *Proceedings MSR*, pages 1–10, 2010.

[10] S. Lee and S. Kang. Clustering and recommending collections of code relevant to tasks. In *Proceedings ICSM*, pages 536–539, 2011.

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

[12] N. I. of Standards & Technology. The economic impacts of inadequate infrastructure for software testing, May 2002. US Dept of Commerce.

[13] L. D. Panjer. Predicting eclipse bug lifetimes. In *Proceedings MSR*, pages 29–, 2007.

[14] R. Robbes and D. Röthlisberger. Using developer interaction data to compare expertise metrics. In *Proceedings MSR*, pages 297–300, 2013.

[15] A. Serebrenik, B. Vasilescu, and M. v. d. Brand. Similar tasks, different effort: Why the same amount of functionality requires different development effort? In *BENEVOL*, 2011.

[16] E. Shihab, A. Ihara, Y. Kamei, W. Ibrahim, M. Ohira, B. Adams, A. Hassan, and K.-i. Matsumoto. Studying re-opened bugs in open source software. *Empirical Software Engineering*, 32(12):1–38, 2012.

[17] Z. Soh, F. Khomh, Y.-G. Guéhéneuc, G. Antoniol, and B. Adams. On the effect of program exploration on maintenance tasks. In *Working Conference on Reverse Engineering (WCRE)*, 2013. To appear.

[18] S. Zaman, B. Adams, and A. E. Hassan. Security versus performance bugs: a case study on firefox. In *Proceedings MSR*, pages 93–102, 2011.