View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by PolyPublie

PolyPublie
Polytechnique Montréal

POLYTECHNIQUE MONTRÉAL
LE GÉNIE EN PREMIÈRE CLASSE

Titre: Title:	Towards the exploration strategies by mining Mylyns' interaction histories
Auteurs: Authors:	Zéphyrin Soh et Yann-Gaël Guéhéneuc
Date:	2013
Type:	Rapport / Report
Référence: Citation:	Soh, Z. & Guéhéneuc, Y.-G. (2013). *Towards the exploration strategies by mining Mylyns' interaction histories* (Rapport technique n° EPM-RT-2013-01).

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

URL de PolyPublie: PolyPublie URL:	http://publications.polymtl.ca/2799/
Version:	Version officielle de l'éditeur / Published version Non révisé par les pairs / Unrefereed
Conditions d'utilisation: Terms of Use:	Tous droits réservés / All rights reserved

Document publié chez l'éditeur officiel
Document issued by the official publisher

Maison d'édition: Publisher:	École Polytechnique de Montréal
URL officiel: Official URL:	http://publications.polymtl.ca/2799/
Mention légale: Legal notice:	

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal

http://publications.polymtl.ca

EPM–RT–2013-01

TOWARDS THE EXPLORATION STRAGEGIES BY MINING MYLYNS' INTERACTION HISTORIES

Zéphyrin Soh, Yann-Gaël Guéhéneuc
Département de Génie informatique et génie logiciel
École Polytechnique de Montréal

Mai 2013

Poly

ÉCOLE POLYTECHNIQUE MONTRÉAL

EPM-RT-2013-01

TOWARDS THE EXPLORATION STRAGEGIES BY MINING MYLYNS' INTERACTION HISTORIES

Zéphyrin Soh, Yann-Gaël Guéhéneuc
Département de génie informatique et génie logiciel
École Polytechnique de Montréal

Mai 2013

©2013
Zéphyrin Soh, Yann-Gaël Guéhéneuc
Tous droits réservés

Dépôt légal :
Bibliothèque nationale du Québec, 2010
Bibliothèque nationale du Canada, 2010

EPM-RT-2013-01
Towards the Exploration Strategies by Mining Mylyns' Interaction Histories
par : Zéphyrin Soh, Yann-Gaël Guéhéneuc
Département de génie informatique et génie logiciel
École Polytechnique de Montréal

Toute reproduction de ce document à des fins d'étude personnelle ou de recherche est autorisée à la condition que la citation ci-dessus y soit mentionnée.

Tout autre usage doit faire l'objet d'une autorisation écrite des auteurs. Les demandes peuvent être adressées directement aux auteurs (consulter le bottin sur le site http://www.polymtl.ca/) ou par l'entremise de la Bibliothèque :

École Polytechnique de Montréal
Bibliothèque – Service de fourniture de documents
Case postale 6079, Succursale «Centre-Ville»
Montréal (Québec)
Canada H3C 3A7

Téléphone : (514) 340-4846
Télécopie : (514) 340-4026
Courrier électronique : biblio.sfd@courriel.polymtl.ca

Ce rapport technique peut-être repéré par auteur et par titre dans le catalogue de la Bibliothèque :
http://www.polymtl.ca/biblio/catalogue.htm

Towards the Exploration Strategies by Mining Mylyns' Interaction Histories

Zéphyrin Soh, Yann-Gaël Guéhéneuc
Department of Computer and Software Engineering
École Polytechnique de Montréal, Canada
Email: {zephyrin.soh, yann-gael.gueheneuc}@polymtl.ca

February 25, 2013

Abstract

When developers perform a maintenance task, they always explore the program, *i.e.*, move from one program entity to another. However, even though maintenance is a crucial task, the exploration strategies (ES) used by developers to navigate through the program entities remain unstudied. This lack of study prevents us from understanding how developers explore a program and perform a change task, from recommending strategies to developers, and (ultimately) from critically evaluating a developer's exploration performance. As a first step towards understanding ES, we mined interaction histories (IH) gathered using the Eclipse Mylyn plugin from developers performing a change task on four open-source projects (ECF, Mylyn, PDE, and Eclipse Platform). An ES is defined and characterized by the way (how) the developers navigate through the program entities. Using the Gini inequality index on the number of revisits of program entities, we observe that ES can be either centralized (CES) or extended (EES). We automatically classified interaction histories as CES or EES and performed an empirical study to ascertain the effect of the ES on the task duration and effort. We found that, although an EES requires more exploration effort than a CES, an EES is less time consuming than a CES. Extensive work (number of days spent performing a task) typically imply a CES. Our results show that developers who follow an EES have a methodical investigation of source code while developers who follow a CES have an opportunistic exploration of source code.

Keywords: *Software Maintenance, Program Exploration, Interaction Histories, Exploration Strategies, Mylyn*

1 Introduction

Software systems must be maintained and evolved to fix bugs and adapt to new technologies and requirements changes. When developers perform a maintenance task, they always 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 [21]. Finding the right set of entities for a task is important for a successful completion of the task [18, 21]. Navigation information represents a valuable resource for program comprehension and would help better understand developers' programming strategies [15]. However, even though maintenance is a crucial task, to the best of our knowledge, the exploration strategies (ES) used by developers to navigate through the program entities remain unstudied. Understanding developers' exploration strategies can lead to the development of new exploration features for Integrated Development Environment (IDE) to assist

developers during the exploration of the source code [23]. The new exploration features could help reduce the developers' effort to find relevant entities in a program and consequently could improve their productivity [6]. The exploration strategies can also help to characterize developers expertise *i.e.*, the difference between exploration strategies can highlight the difference between developers' experience.

In this report, we analyse developers' interaction histories collected from four open-source projects using Mylyn: Eclipse Communication Framework (ECF)[1], Mylyn[2], Eclipse Plug-in Development Environment (PDE)[3], and Eclipse Platform[4]. Mylyn is an Eclipse plugin that captures all developers' interactions with the program entities when performing a task. ECF is a framework for building distributed servers, applications, and tools. Eclipse PDE provides tools to create, develop, test, debug, build and deploy Eclipse plug-ins. Eclipse Platform defines the set of frameworks and common services that collectively make up infrastructure required to support the use of Eclipse. We then answer the following four research questions:

RQ1) *Do developers follow specific exploration strategies when performing maintenance tasks?*
There are several way in which a developer can interact with entities while performing a maintenance task. This RQ aims to study how developers explore program entities. Two extreme cases are when the developers concentrate all (most) of his activity on a limited number of entities and when there is no privileged entity subset. In the latter case all entities are (almost) equally visited. In this report we are referring to these two extreme cases as the centralized exploration strategy (CES) and the extended exploration strategy (EES).

A CES is an exploration based on one (or a set of) program entity(ies), called "core entity(ies)". These are entities that a developer frequently revisits and concentrates upon. On the contrary, in an EES strategy, a developer visits program entities with almost the same frequency, *i.e.*, there is no set of preferred entities. Based on Robillard *et al.*'s findings that methodical developers do not reinvestigate methods as frequently as opportunistic developers [18], we argue that developers who follow an EES strategy perform a methodical investigation of the source code, while developers who follow a CES strategy perform an opportunistic exploration.

RQ2) *Do specific exploration strategies affect the maintenance time?*
A maintenance time is the time spent performing a maintenance task. We conjecture that the way developers explore the program entities can affect the time spent to perform a maintenance task. The goal of this RQ is to study whereas exploration strategies affect or not the maintenance time. An EES is on average 69.39% less time consuming than a CES. Our results is consistent with Robillard *et al.*'s findings [18], which state that methodical developers perform the task in half of the time compared to opportunistic developers. Moreover, opportunistic developers must guess and read the source code in details [18], which may explain the longer time spent with CES.

RQ3) *Do specific exploration strategies affect the exploration effort?*
We measure the exploration effort as the ability of developer to find the program entities to modify (see Section 3.3.1). We think that finding the program entities that need to be modify depends on how developers explore the program entities. An EES requires more exploration effort than a CES. While this observation is surprising, we think that a CES requires less exploration effort because opportunistic developers make their code modifications in one place [18]. We are

[1] http://www.eclipse.org/ecf/
[2] http://www.eclipse.org/mylyn/
[3] http://www.eclipse.org/pde/
[4] http://wiki.eclipse.org/Platform/

aware that other factors such as the architecture, the developer experience, style or preferences may play an important role. Here, in this first study we limit ourselves to verify if indeed CES and EES are somehow tied to different effort distribution.

RQ4) *Does extensive work results to a specific exploration strategy?*
We define an extensive work using the number of days developers work on a maintenance task. We think that the more days developers work on a task, the more they can follow a specific exploration strategy. The extensive work usually results to a centralized exploration strategy. Developers adopt CES for extensive works because they must refresh their knowledge of program entities.

The reminder of this technical report is organized as follows. Section 2 provides some background knowledge on the task life-cycle management framework Mylyn and the Gini inequality index used to identify developers' exploration strategies. Section 3 describes the design of our empirical study, including data collection and processing, the identification of exploration strategies, and our analysis approach. Sections 4 to 7 presents the results of our four research questions and Section 8 discusses the threats to their validity. We relate our study to previous work in Section 9. Section 10 summarizes our findings and highlights some avenues for future work.

2 Background

This section presents background knowledge on the Mylyn Plugin used to collect developers' interactions histories and the Gini Inequality Index used to identify the exploration strategies.

2.1 Mylyn Plugin

Mylyn is an Eclipse plugin that captures developers' interactions with program entities when performing a task. The definition of a task is the starting point to use Mylyn. The developers must define a task and activate the current task they are working on. Each developer's 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* [11]. 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 [4]. Interaction histories logs are stored in an XML format. Each interaction history log is identified by a unique *Id* (*i.e.*, the task identifier) and contains 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 affordance that tracks the event (*i.e.*, *OriginId*), and the program entity involved in the event (*i.e.*, *StructureHandle*). Figure 1 shows an interaction event extracted from the interaction history #89344 of the Eclipse Platform project.

```
<InteractionEvent StructureKind="java"
StructureHandle="=org.eclipse.ui.externaltools/External Tools
Base&lt;org.eclipse.ui.externaltools.internal.launchConfigurations
{ExternalToolsMainTab.java[ExternalToolsMainTab"
StartDate="2008-02-09 19:11:42.15 CET"
OriginId="org.eclipse.jdt.ui.CompilationUnitEditor"
Navigation="null" Kind="edit" Interest="2.0" EndDate="2008-
02-09 19:12:44.46 CET" Delta="null"/>
```

Figure 1: An example of interaction event from Eclipse Platform's IH #89344

Mylyn also records events that are not directly triggered by developers. However, in this study, we are interested in developers' exploration strategies, so we consider only developer's interaction events: *Selection*, *Edit*, *Command*, and *Preference*. Mylyn interaction history logs are compressed, encoded under the Base64 format, attached to change request reports, and stored in change request tracking systems.

2.2 Gini Inequality Index

We use the Gini inequality index to investigate exploration strategies. In econometrics, many inequality indices are used to measure the inequality of income in the society. The Gini inequality index is "*one of the most used indicators of social and economic conditions*" [25]. We choose the Gini inequality index because (1) it has been used in previous software engineering studies [8, 9, 24] and (2) the mathematical properties of the Gini inequality index presented by Mordal *et al.* [9] are conform to the metric that we use as income in Section 3.3.1. In this study, we are interested in the inequality 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, which is defined in Section 3.3.1, together with other metrics we used in this technical report.

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 *i.e.*, where only one person has all the income. Xu [25] presented many computational approaches for the Gini inequality index and mentioned that theses approaches are consistent with one another. These approaches include geometric, mean difference, covariance, and matrix approaches. Xu [25] mentioned that theses approaches are consistent with one another. As used in [8, 9], we use the mean difference approach (see Section 3.3.2) defined as "*the mean of the difference between every possible pair of individuals, divided by the mean size* μ".

3 Study Setup

To investigate exploration strategies (ES), we follow the steps below (summarized in Figure 2):

1. Data collection (Section 3.1)
2. Data parsing (Section 3.2)
3. Exploration strategies identification (Section 3.3)
    - Definition of the metrics (Section 3.3.1)
    - Identification process (Section 3.3.2)
4. Data Analysis (Section 3.4)
5. RQ1 (Section 4)
6. RQ2 (Section 5)
7. RQ3 (Section 6)
8. RQ4 (Section 7)

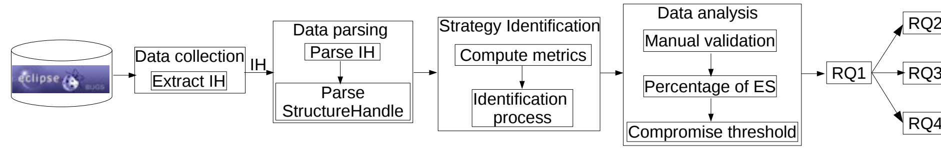


Figure 2: Overview of the approach

Table 1: Descriptive statistics of the data

	Projects				
	ECF	**Mylyn**	**PDE**	**Platform**	**All**
Number of bugs	138	1603	464	396	2601
Number of IH	158	2309	567	579	3613
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	1606	372	335	2375
IH <= 2 classes	27	285	204	45	561
IH class level	26	1273	131	275	1705
IH <= 2 files	25	276	183	37	521
IH file level	34	1316	180	293	1823

3.1 Data Collection

We downloaded 3,609 bug reports from Eclipse bug report system[5]. 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 downloaded and parsed 3,613 interaction histories. We clean our data set by removing 110 interaction histories that do not pertain to Java program entities (see Section 3.2), 153 interaction histories that have at least one event with a negative duration, and 973 interaction histories that have a null duration. We retain 2,375 interaction histories.

Because we are interested to exploration strategy, we think that an exploration strategy cannot be found with an interaction history in which only one or two program entities are involved. Therefore, for class (respectively file) levels, we removed 561 (respectively 521) interaction histories in which only one or two classes (respectively files) were involed. Overall, we kept 1,705 interaction histories for class level and 1,823 interaction histories for file level. Table 1 presents a description of the data set.

3.2 Data Parsing

We parse the interaction histories to extract useful data. As we are interested in the exploration of program entities, the name of a program entity on which an event occur is the most important piece of data which is identified by the *StructureHandle* attribute for each event (see Figure 1). A program element can be a resource (XML file, MANIFEST.MF file, properties file, HTML file, etc.) or a Java program entity (project, package, file, class, attribute, or method). We use the *StructureKind* attribute of interaction event to distinguish resource and Java *StructureHandles*. Figure 3 shows an example of resource *StructureHandle* (Figure 3a) and Java *StructureHandle* (Figure 3b). We consider only Java *StructureHandles* and we explain how we parse them in the following Section.

[5]https://bugs.eclipse.org/bugs/

```
/org.eclipse.pde.ui/icons/elcl16/restore_log.gif
/org.eclipse.pde.ui/build.properties
/org.eclipse.pde.ui/plugin.xml
```

(a) Resources (from PDE's IH #82914)

```
=org.eclipse.ecf.provider.jslp/src&lt;org.eclipse.ecf.internal.provider.jslp
{ServicePropertiesAdapter.java[ServicePropertiesAdapter
~ServicePropertiesAdapter~QList;
```

(b) Java (from ECF's IH #120570)

Figure 3: An example of *StructureHandle*

3.2.1 Identification of the Parts of Java *StructureHandle*

We found that a Java *StructureHandle* is structured in multiple parts: the project name, an optional package, file, class, attribute, or method name. We validate the structure of Java *StructureHandle* in two ways:

- We follow some IHs and for each interaction event, we navigate through the related source code and check the presence of each program entity found in the *StructureHandle*.
- We use the Mylyn plugin when we perform some maintenance tasks on the project that we used to parse the bugs data. Then, we explore the gathered interaction histories and compare with the exploration we did.

As Java *StructureHandle* is well-structured, we use a regular expression to identify all parts of *StructureHandle*. Regular expressions is already used by Bettenburg *et al.* [1] to identify the parts of stack trace in the bug reports. Figure 4 shows the structure of java *StructureHandle*. Due to the containment principle of Java, a *StructureHandle* cannot contain a package name without a project name; a file name without a project name and a package name, etc, and we do not need to add many overlapping optional marks ([]).

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

Figure 4: The structure of a Java *StructureHandle*

A Java *StructureHandle* sometimes starts with the character "=" followed by the name of the project. Sometimes the name of the project contains "/" and/or "\/". Since a project can have more than one source folder, the character "/" indicates the source folder while "\/" indicates the sub folders. We replace "/" by "." and "\/" by "/". Sometimes, the name of the project ends with "<". We clean the project name by removing both "=" (at the beginning) and "<" (at the end). Sometimes, the project also contains the jar file. For example in the PDE IH #82914, many jar files are in the project name, e.g., org.eclipse.mylyn.context.tests/C:\/eclipse-target-platform\/eclipse-SDK-I20071113-0800-win32\/plugins\/org.eclipse.pde.core_3.4.0.v20071113-0800.jar. Table 2 shows the parts of a Java *StructureHandle* presented in Figure 3b. After the project name, the rest of the *StructureHandle* is optional. The package name follows the character ";". Sometimes, the package name is empty. It indicates that (1) a program entity is on the root of a source folder or (2) a Java code containing a program entity is not in a source folder. For example, a developer creates a folder "test" (not a source folder), and "save" her source code (copy the "src" folder in "test"), then browses the code in the "test" folder. Anyway, in that case, the name of the package is already in the project name. So, we just put the name of the project and ignore the name of the package. The file name follows "{" or "(". If the file name follows "{", the file is a Java file, and if the file name follows "(", the file is a class file. The class name follows "[". Sometimes, there are internal classes or enumeration

types in the class. So, the class name can contain additional character "[" that separates the class name and the name of subclass/enumeration type. In case of the presence of subclass/enumeration type, we replace "[" by ".". After a class name is an attribute or a method. The attribute name follows the character "ˆ", and the method name follows the character "∼". We use [*] to materialize the rest of a *StructureHandle*. We explain below how we use the rest of a *StructureHandle*.

Part	**Part Name**
Project	org.eclipse.ecf.provider.jslp.src
Package	org.eclipse.ecf.internal.provider.jslp
File	ServicePropertiesAdapter.java
Class	ServicePropertiesAdapter
Attribute	
Method	ServicePropertiesAdapter
Rest	∼QList;

Table 2: Identification of the parts of *StructureHandle* in Figure 3b

3.2.2 Identification of the Type and Name of a Program Entity

As an event occurs on a program entity which can be a project, package, file, class, attribute, or method, the whole *StructureHandle* identify a program entity. Based on the containment principle, the type of the program entity on which the event occurs is identified by the presence of more internal part of *StructureHandle*. The *StructureHandle* in Figure 3b shows that the event occurs on a method. Figure 5 shows the examples of *StructureHandle* corresponding to different type of program entity, i.e., project (Figure 5a), package (Figure 5b), file (Figure 5c), class (Figure 5d), and attribute (Figure 5e).

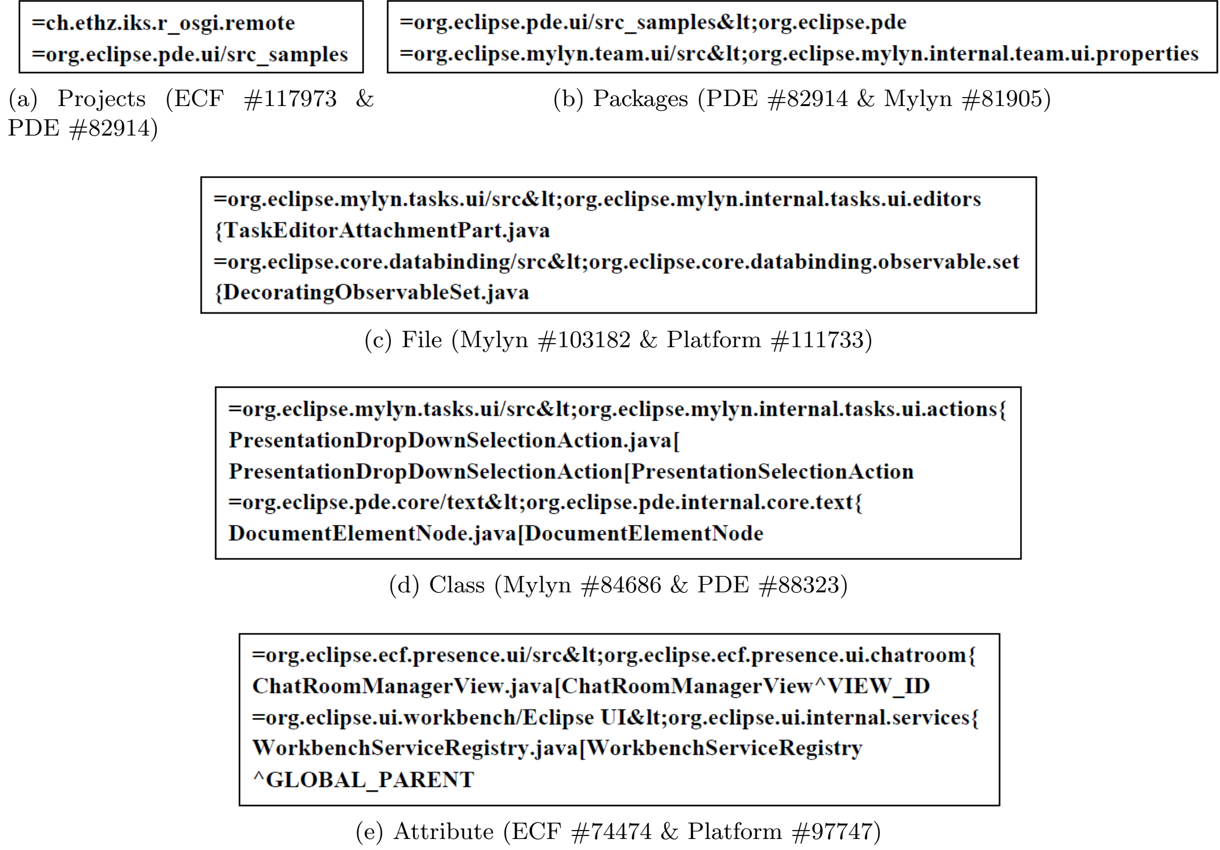
Having identified the type of a program entity and the parts of a *StructureHandle*, we obtain the complete name of the entity by using the containment principle (*e.g.*,, project.package.file.class.method).

Concerning the rest of the *StructureHandle*, when a developer selects an import declaration in a file, the event occurs on the file, but the name of the file can be followed by the name of the imported package, separated by a special character (*e.g.*, #). Well we can have the rest of *StructureHandle* for other type of program entity. At this state of the investigation, we limit the depth of the investigation. Therefore, when an event occurs on a project, package, file, class, or attribute, we consider the rest of the *StructureHandle* as a trash, and remove it to the name of the program entity. We define a regular expression with all the identified special characters (=;{([∼ˆ!|#) to remove the rest of the *StructureHandle*. For the event occurs on a method, since we can have different methods with a same name and different signatures, we use the rest of *StructureHandle* to distinguish different methods.

3.3 Exploration Strategies Identification

3.3.1 Definition of the Metrics

In the interaction history, an interaction event occurs on a program entity. In object-oriented paradigm, there are many type of program entity (file, class, method, attribute). We compute the metrics for each type of program entity. The computation of the metrics for one type of program element is defined as a level. So, we compute the metrics for file level, class level, attribute level, and method level. We also 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`.



(a) Projects (ECF #117973 & PDE #82914)

(b) Packages (PDE #82914 & Mylyn #81905)

(c) File (Mylyn #103182 & Platform #111733)

(d) Class (Mylyn #84686 & PDE #88323)

(e) Attribute (ECF #74474 & Platform #97747)

Figure 5: Examples of *StructureHandle* for different types of program entity

We introduce the following notations: Let L be a program entity level, *i.e.*, file, class, attribute, or method level.

- $e_n \in L$ (is a): e_n **is a** program entity at level L, *e.g.*, $Foo.java \in file$.
- $e_v \rhd e_l$ (occur on): an interaction event e_v **occurs on** the program entity e_n, *e.g.*, an event can occur on the method $foo()$.
- $e_v \in IH$ (is a): e_v **is an** interaction event in the interaction history IH.

We define the number of revisits used to identify the exploration strategy as followed:

- Number of revisits: $NumRevisit(e_n)$ defines the number of time the entity e_n is revisited, which is different to the number of events. Consider an interaction history with five interaction events that occurred on a set of three program entities $\{e_{n_1}, e_{n2}, e_{n3}\}$. If we suppose that the events occurred in the following order: $e_{n_1} \rightarrow e_{n_2} \rightarrow e_{n_2} \rightarrow e_{n_3} \rightarrow e_{n_1}$. 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.

We also compute the following metrics (if applicable) for each interaction event e_v, program entity e_n, level L, and interaction history IH:

- Level Duration: a cumulative duration spent on entities at level L.

$Duration(e_v) = EndDate(e_v) - StartDate(e_v)$

$Duration(e_l) = \sum_{e_v \rhd e_l} Duration(e_v)$

$Duration(L) = \sum_{e_l \in L} Duration(e_l) = \sum_{\substack{e_v \rhd e_l \\ e_l \in L}} Duration(e_v)$

- Overall Duration: a duration of interaction history (IH).

  $Duration(IH) = \sum_{e_v \in IH} Duration(e_v)$

- Number of events: $NumEvent(L)$ is the total number of user events occurring on entities at level L.

  $NumEvent(e_l) = \#e_v, e_v \in UE, e_v \rhd e_l$

  $NumEvent(L) = \sum_{e_l \in L} NumEvent(e_l)$

- Number of edit events: a total number of edit events, e_v is an edit event if $kind(e_v) =$ "*Edit*" (see Section 2.1)

  $NumEdit(e_v) = \begin{cases} 1 & \text{if } kind(e_v) = \text{“}Edit\text{”} \\ 0 & \text{if } kind(e_v) \text{ != “}Edit\text{”} \end{cases}$

  $NumEdit(e_l) = \sum_{e_v \rhd e_l} NumEdit(e_v)$

  $NumEdit(L) = \sum_{e_l \in L} NumEdit(e_l)$

- Exploration effort: we use an edit ratio to measure the exploration effort. An edit ratio is a number of edit divided by the number of events. It relates to the identification of the right program entities to edit.

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

3.3.2 Identification Process

We explain the process to identify exploration strategy below.

Income for Inequality Index

We use the Gini inequality index for identification of exploration strategy. As Gini inequality index works with incomes (see Section 2.2), we use the number of revisits of program entities as income to compute the Gini inequality index. We calculate the Gini inequality index as follows (n is the total number of visited entities):

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

We use the Gini inequality index to find if there exists any (un)equality between visited entities. We must define a threshold to determine whereas entities are equally or unequally revisited.

Exploration Strategies Identification and Thresholds

We plot all Gini values and we did not find a value (pic) that can be used as threshold *i.e.*, that split the interaction histories into two groups. Therefore, we use 10 threshold values ranging from 0.1 to 1 per step of 0.1. We identify the threshold as explain in Section 4. We identify exploration strategies 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 strategy is extended (EES).
- If the Gini is greater or equal to the threshold, it means that the revisits are concentrated on a few program entities, *i.e.*, core entities. Thus, we can split the visited entities into two groups, the more revisited group (core entities) and the less revisited one (periphery entities). We say that the exploration strategy is centralised (CES).

Cut-off Point

After the identification of exploration strategy, we should define the splitting process to identify the principal elements. The splitting process is performed only if the exploration strategy is centralized. We define a Cut-off Point (COP) to split the visited entities into two groups (core and periphery) as follows: we order the visited entities by *NumRevisit* from high to low values and we define the COP as the maximum difference between *NumRevisit* of two consecutive entities. Formally, if the visited entities are $e_{n_i}, i = 1, \ldots, m$, and we use i' to iterate on the ordered entities, *i.e.*, $NumRevisit(e_{n_{i'}}) \geq NumRevisit(e_{n_{i'+1}}), i' = 1, \ldots, m-1$, then:

$$COP = \max_{1 \leq i' < m} NumRevisit(e_{n_{i'}}) - NumRevisit(e_{n_{i'+1}})$$

After the identification of exploration strategy, we should define the splitting process to identify the principal elements. The splitting process is performed only if the exploration strategy is centralized. We define the Cut-off Point (COP) as the point to split the visited elements into two groups. The COP is identify as follow:

If we find two COPs, we use the COP that provide more principal entity. For example, if we have the following *NumRevisit* 16, 11, 9, 4, 2 with Gini values greater than the considered threshold, the maximum difference between consecutive values is 5. we obtain two COP= 5, between 16 and 11 and between 9 and 4. In such case, we choose the COP between 9 and 4 and the core entities are those with *NumRevisit* greater or equal to 9 (*i.e.*, 16, 11, and 9).

3.3.3 Confounding Factor

Developers explore the program by following different kind of relationships [23] and using key binding [10]. We conjecture that the architecture of the system can affect the exploration strategy. To address this confounding factor, we compute the number of common entities between each pair of interaction histories. The program entities involved in an IH, including the relations between them are the part of the system used to perform a task. Thus, the common parts between IHs is captured by the number of common entities. If it is true that developers are guided only by the architecture, then two IHs that were explored using different ES will have few common entities compare to IHs explored using the same ES.

3.4 Analysis Method

To answer our first research question on the identification of exploration strategies (*i.e.*, **RQ1**), we proceed in two steps: (1) we randomly sample the interaction histories and we manually validate the sample (oracle). The validation is done at class level; and (2) we choose a compromise threshold value with high precision and recall. When sampling, we ensure to cover a varied number of program entities and the different versions of the projects. We also avoid selecting multiple interaction histories from a same bug. We choose the sample size in order to achieve a 95%±10 confidence level. Because of the small number of data from the ECF project in our data set, we consider half of ECF interaction histories in our sample (instead of two as suggested by the sample size). Three students manually validated all CES and EES in the samples. We compute the precision and recall for each threshold. The subjects were enrolled in the PhD program in software engineering at the École Polytechnique de Montréal. They already completed at least two years and a half of their program and used Java in their research projects. We consider the percentage of CES and EES of the compromise threshold to answer **RQ1**.

To investigate the effect of exploration strategies on maintenance tasks (**RQ2**, **RQ3**, and **RQ4**), we perform an unpaired version of the non parametric Wilcoxon rank sum test. The Wilcoxon rank sum test is a non-parametric statistical test to assess whether two independent distributions have equally large values. 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.*, $\alpha = 0.05$).

4 Do developers follow specific exploration strategies when performing maintenance tasks?

We answer this question by following the method defined in Section 3.4. For the first step, three students manually validated the exploration strategies of a random set of interaction histories. Table 3 presents the size of the samples for each project.

Table 3: Validation samples, precision and recall values for the compromise Gini's threashold

		Products			
		ECF	**Mylyn**	**PDE**	**Platform**
Sample size		13	68	7	16
Undecided		0	1	1	1
Precision	**CES**	1	0.89	1	0.87
	EES	0.8	0.76	1	0.71
Recall	**CES**	0.60	0.60	1	0.77
	EES	1	0.94	1	0.88

To validate the exploration strategies, we generated the Graphviz [3] representation of the interaction histories. Then, we use Graphviz to generate PNG file. Grapviz[6] is an open-source graph visualization software. Sometimes, some generated Graphviz files encountered the errors when Graphviz try to generate PNG files. These errors were related to the restrictions[7] in the name of the Graphviz node or the graphs that are too large[8]. We remove these files before choose the random set of interaction histories.

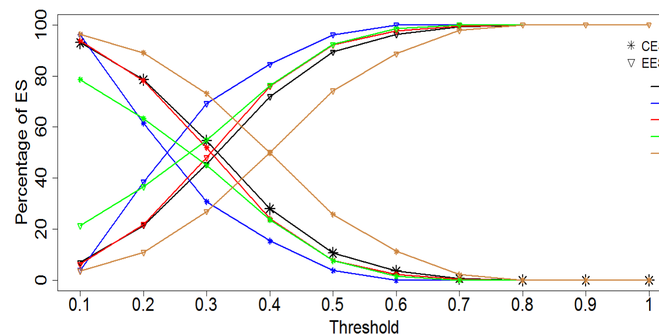
[6] http://www.graphviz.org/
[7] For example the character $ in the name of the .class file
[8] Error message "graph is too large for cairo-renderer bitmaps"

We explained to the subjects that a graph represents the exploration between program entities. The nodes are the program entities and the arrow between two nodes represents a developers' movement from one program entity to another. We asked the subjects to say if the exploration graphs are centralized, extended, or if they have a doubt. At the end of the validation, we asked each subject to explain their choice, using the following questions: (1) How did you judged that a graph was centralized or extended? (2) Why did you had a doubt on some graphs? Subjects comments were about counting the number of in-out arrows and the number of entities involved in the graphs. Their identification process is consistent with the definition of the Gini inequality index: when the number of program entities is high and some of them mainly revisited, the Gini inequality index is also high, characterizing a CES.

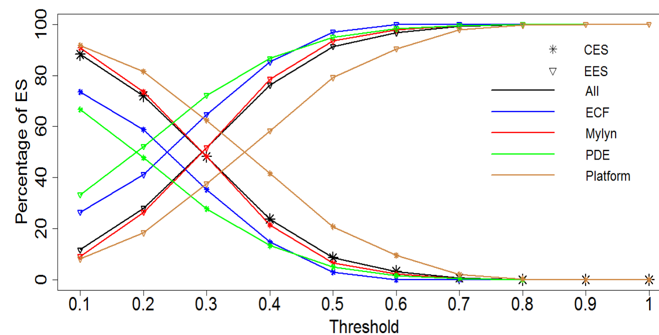
Based on the subjects' data, we decided that an exploration strategy is centralized (or extended) if at least two subjects identified the corresponding graph as centralized (or extended). The undecided cases where all subjects had different interpretations, or where at least two subjects had a doubt, were removed from the study. In total, we had three undecided cases (see Table 3).

For step two, we define a compromise Gini threshold. The percentage of exploration strategies obtained for threshold values ranging from 0.1 to 1 per step of 0.1 is presented in Figure 6 for class level (Figure 6a) and file level (Figure 6b). Because we want to maximize both precision and recall, we compute the F-Measure for the threshold values around 0.3 as follow:

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



(a) Class level



(b) File level

Figure 6: Percentage of exploration strategies per threshold

Figure 7 shows the distribution of the F-Measure for class level validation. It indicates that the

identification of exploration strategies is most accurate at 0.4 threshold. Therefore, we consider the value 0.4 as a compromise threshold both for class and file level in the reminder of this technical report.

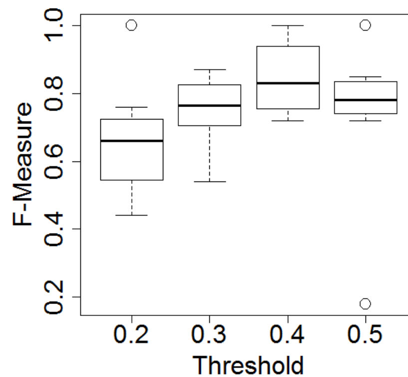


Figure 7: F-Measure per threshold for the validation sample

According to the compromise threshold, Tables 4 and 5 presents the percentage of exploration strategies for class and file level respectively (LD=Level Duration, OD=Overall Duration, EW=Extensive Work).

Table 4: Percentage of exploration strategies (ES) and p-values for class level

		ES		**p-values**			
		#	**%**	**LD**	**OD**	**EditRatio**	**EW**
ECF	**CES**	4	15.38	0.028	0.065	0.12	0.036
	EES	22	84.61				
Mylyn	**CES**	306	24.03	<2.2e-16	<2.2e-16	< 2.2e-16	<2.2e-16
	EES	967	75.96				
PDE	**CES**	31	23.66	2.6e-08	1.4e-07	4.1e-05	1.1e-08
	EES	100	76.33				
Platform	**CES**	137	49.81	1.5e-09	4.3e-07	9.9e-12	2.2e-07
	EES	138	50.18				
All	**CES**	478	28.03	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16
	EES	1227	71.96				

Figures 8 and 9 show examples of CES and EES for class level. Because the architecture of the system can affect the exploration strategy, we study the number of common entities for each pair of ES. We focus the study of the architecture effect on the class level because the object oriented paradigm allow the relations between classes and not between files. 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 distinguish the project, there is no statistical significant difference (p-value = 1). Therefore, architecture does not affect the ES.

> ***Observation 1**: Developers follow mostly the EES exploration strategy when performing a maintenance task.*

Table 5: Percentage of exploration strategies (ES) and p-values for file level

		ES		p-values			
		#	%	LD	OD	EditRatio	EW
ECF	**CES**	5	14.70	0.29	0.33	0.20	0.10
	EES	29	85.29				
Mylyn	**CES**	282	21.42	<2.2e-16	<2.2e-16	< 2.2e-16	<2.2e-16
	EES	1034	78.57				
PDE	**CES**	24	13.33	3.1e-07	6.3e-07	2.3e-07	4.03e-10
	EES	156	86.66				
Platform	**CES**	122	41.63	7.9e-09	2.08e-08	<2.2e-16	3.1e-8
	EES	171	58.36				
All	**CES**	433	23.75	< 2.2e-16	< 2.2e-16	< 2.2e-16	< 2.2e-16
	EES	1390	76.24				

The exploration strategy concept is consistent with Robillard *et al.* [18] findings that state that methodical developers do not reinvestigate methods as frequently as opportunistic developers. The number of revisits used to identify exploration strategies somehow measures/correlates to the reinvestigation frequency. Therefore, we argue that an EES is followed by methodical developers while a CES is followed by opportunistic developers. Moreover, Robillard *et al.* [18] argue that methodical developers seem to answer specific questions using focussed search, while opportunistic developers guess and read the source code in details. Because finding focus points and expanding them are some of the steps that developers follow during maintenance tasks [21], we think that the EES strategy is a linear pattern of focus point expansion, while the CES strategy is a star pattern of focus point expansion.

5 Do specific exploration strategies affect the maintenance time?

In **RQ1**, we found that developers follow different exploration strategies when performing maintenance tasks. We conjecture that these strategies 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 and file levels and on the whole task, whether the time spent by developers to perform a task is affected by their exploration strategies.

Tables 4 and 5 show that there is significant difference (at class/file level and overall) between the time spent when following CES and EES. In general, **the exploration strategy affects both the duration at class/file level and the overall duration**. When we study the effect of exploration strategies on the duration of each project individually, at class level, there is no statistical difference in overall duration between CES and EES for the ECF project. At file level, there is no statistical difference in file level duration and overall duration between CES and EES. We think that the case of the ECF project is possibly because there are only 26 (respectively 34) interaction histories at class (respectively file) level.

Without distinguishing the projects, we found that **the CES is the most time consuming strategy** for both class/file level durations and overall durations. The mean (and standard deviation) of class level durations for CES is 35,250 (121,324.7) sec. vs. 6,170 (24,061.21) sec. for EES. The mean (and standard deviation) of file level durations for CES is 42,260 (116,404.7) sec. vs. 14,440 (83,232.1) sec. for EES.

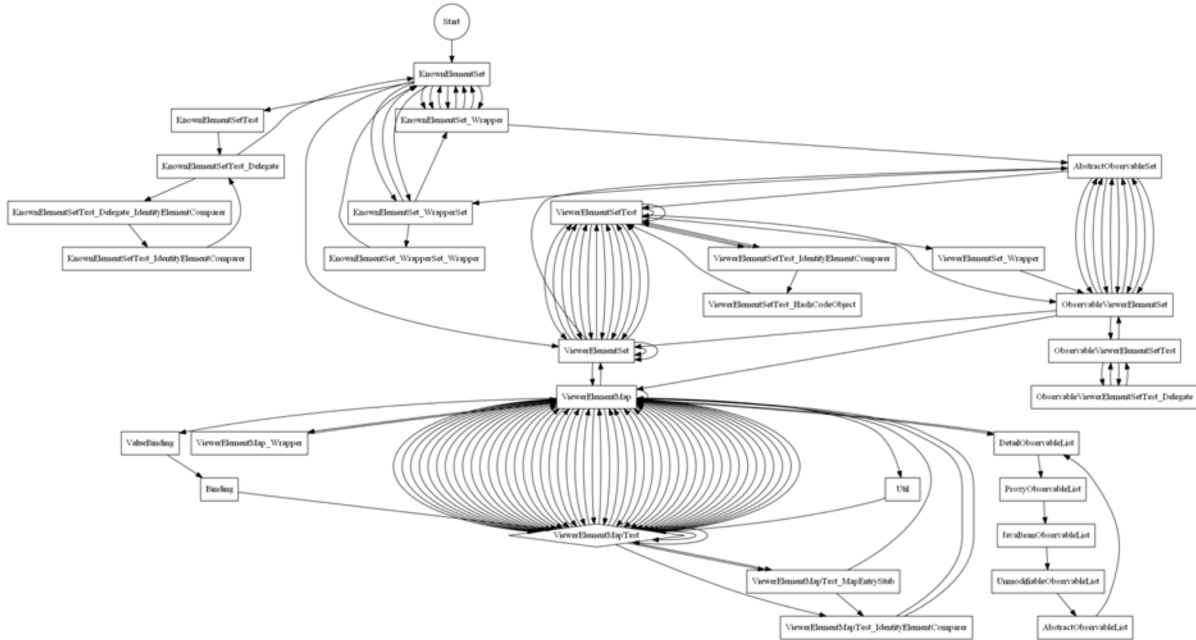


Figure 8: An example of CES (IH #90442 from Platform)

> ***Observation 2:*** *For the class (respectively file) level duration, the EES strategy is on average 82.49% (respectively 65.83%) less time consuming than the CES strategy.*

At class level, the mean (and standard deviation) of the overall duration for CES is 53, 790 (167, 804.7) sec. vs. 16, 460 (80, 629.34) sec. for EES. At file level, the mean (and standard deviation) of the overall duration for CES is 48,500 (132,372) sec. vs. 18,160 (100,692.7) sec. for EES.

> ***Observation 3:*** *For the overall duration, the EES strategy is on average 69.39% (respectively 62.55%) less time consuming than the CES strategy at class and file level respectively.*

Figure 10 shows the difference between the logarithm of the overall duration of CES and EES strategies for each of our studied systems.

Our result are in agreement with the findings of Robillard *et al.* [18], which states that the methodical investigation of a source code does not require more time than an opportunistic investigation. Moreover, methodical developers performs tasks two times faster than others [18]. Therefore, an EES that is less time consuming is likely to be related to a methodical investigation of the source code while a CES may be the result of an opportunistic investigation of the source code. The fact that opportunistic developers guess and read the source code in details could be a justification for the time spent. On the contrary, the less time spent when following an EES may be because developers who follow an EES look at explicit program entity(ies).

6 Do specific exploration strategies affect the exploration effort?

Similarly to **RQ2**, we think that the exploration strategy can affect developers' exploration effort. By definition (see Section 3.3.1), a low value of *EditRatio* indicates a small number of edit events and

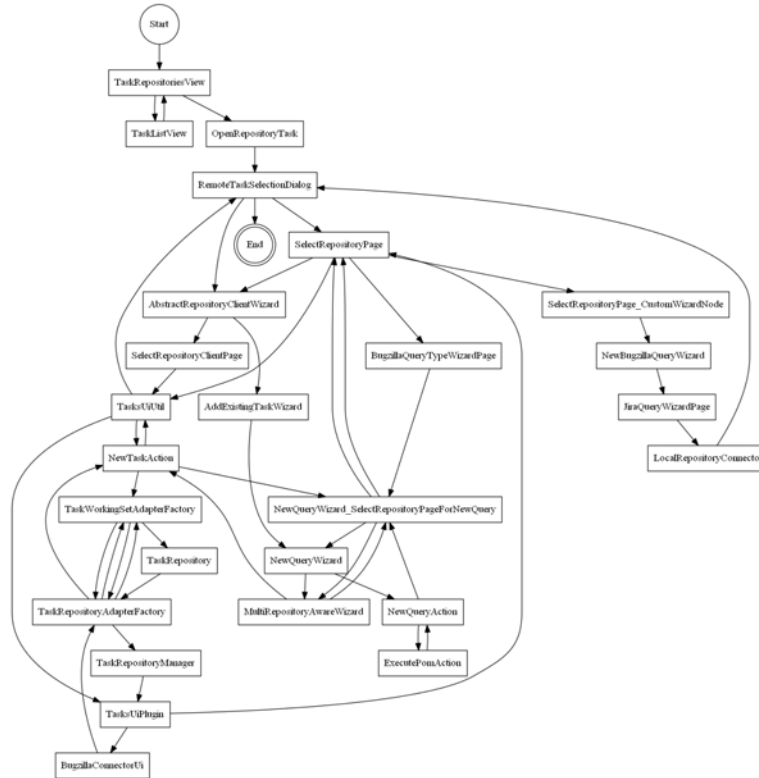


Figure 9: An example of EES (IH #72336 from Mylyn)

a high number of other events: the developer spent more exploration effort. When the *EditRatio* is high, the developer spent less exploration effort and performed edit events more frequently.

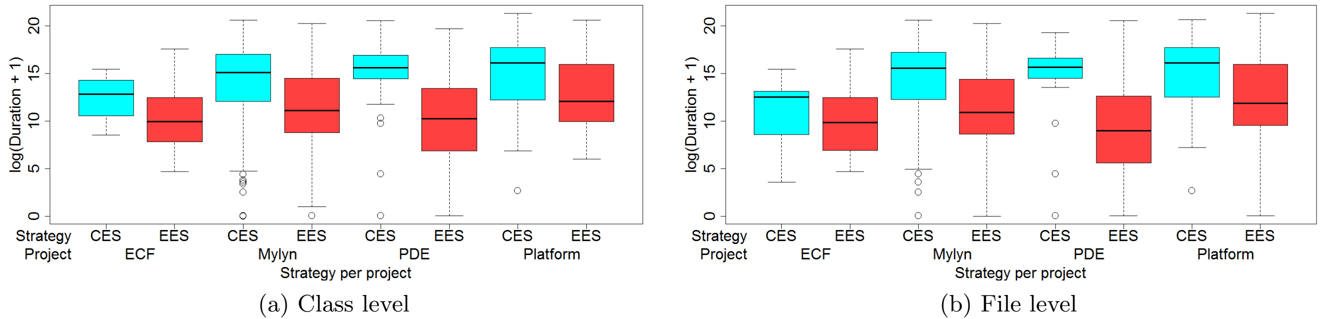
As shown in Tables 4 and 5, except for the ECF project, developers' exploration efforts are significantly different for CES and EES. By investigating the less costly exploration strategy in term of exploration effort, Figure 11 show that the exploration effort of EES is always smaller than that of CES for all projects.

Observation 4: *An EES strategy requires more exploration effort than a CES strategy.*

These results indicate that a methodical approach requires more exploration effort, which is surprising because methodical developers are successful developers compared to opportunistic developers [18]. However, the fact that opportunistic developers make their code modifications in one place [18] can justify the reduced exploration effort for a CES. Indeed, it is likely that opportunistic developers could 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 CES.

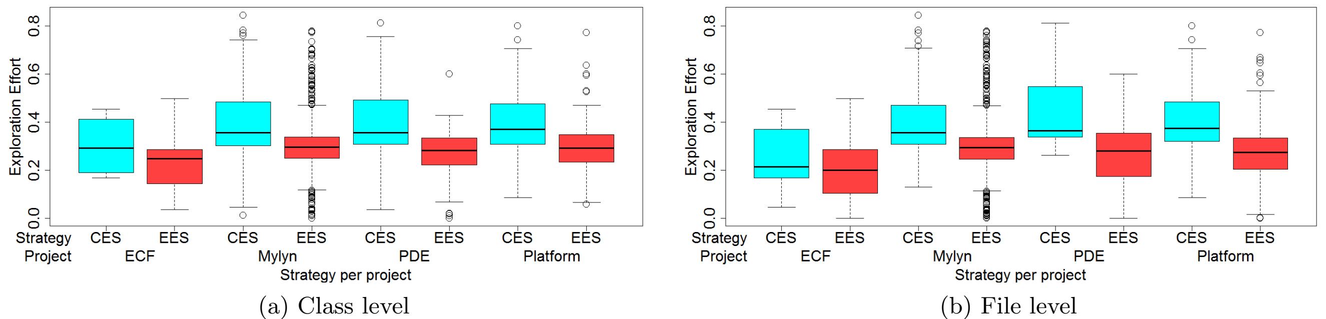
7 Does extensive work results to a specific exploration strategy?

Sometimes, and particularly 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. Hence, they can work



(a) Class level (b) File level

Figure 10: Overall duration per project



(a) Class level (b) File level

Figure 11: Difference on exploration effort

on a change request for several days. By extensive work, we mean that the number of days (from an interaction history) spent by a developer on a change request is greater than three days. In this study, working days are consecutive or not.

We use the Wilcoxon unpaired test and found that, according to the number of working days, there is a statistical significant difference between CES and EES at class level (see Table 4). There is no significant difference for ECF product at file level (see Table 5). When we study the percentage of interaction histories per number of working days, Tables 6 and 7 show 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 12) shows that the more days developers work on a change task, the more time they spend on program entities.

Table 6: Percentage of interaction history per number of working days for class level

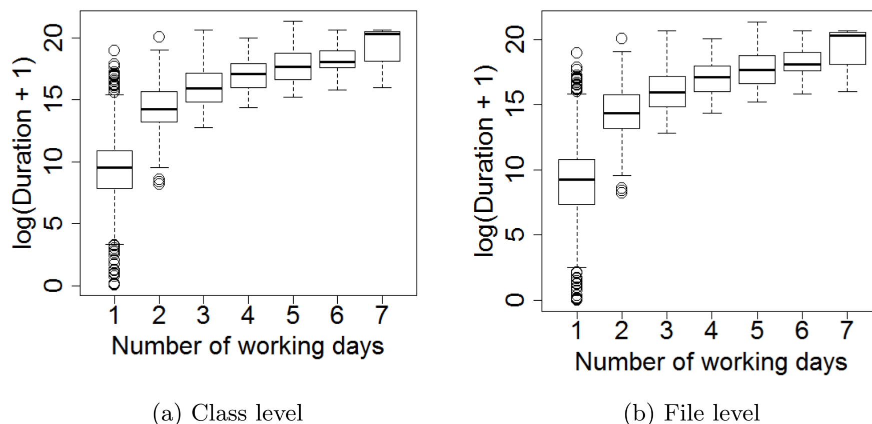
	Number of working days						
	1	**2**	**3**	**4**	**5**	**6**	**7**
ECF	69.23	15.38	15.38	0	0	0	0
Mylyn	54.43	22.54	11.07	5.34	4.24	1.72	0.62
PDE	56.48	20.61	12.21	6.87	3.05	0.76	0
Platform	38.90	22.18	12.36	6.54	10.54	2.18	7.27
All	52.31	22.22	11.43	5.57	5.10	1.70	1.64

According to **RQ2**, a CES is more time consuming. More time spent for more working days

Table 7: Percentage of interaction history per number of working days for file level

	Number of working days						
	1	**2**	**3**	**4**	**5**	**6**	**7**
ECF	67.64	17.64	14.70	0	0	0	0
Mylyn	55.24	22.49	10.71	5.16	4.10	1.67	0.60
PDE	67.22	16.66	8.33	5	2.22	0.55	0
Platform	41.63	21.84	11.60	6.14	9.89	2.04	6.82
All	54.47	21.72	10.69	5.21	4.77	1.59	1.53

indicates that a CES is probably the most followed strategy when a task spans multiple working days, as shown on Figure 13. Therefore, at class level (see Figure 13a) we conclude that when developers work on maintenance tasks for less than three days, more often, they follow an EES. On the contrary, when a maintenance task spans four or more days (*i.e.*, is extensive), developers follow the centralized exploration strategy frequently. The same result is find at file level but the work is consider as extensive when developers work for more than five or more days (see Figure 13b).



(a) Class level

(b) File level

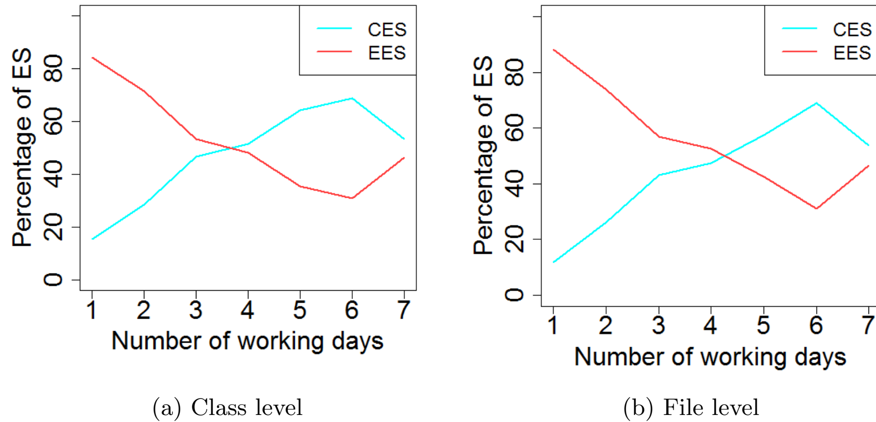
Figure 12: Distribution of duration per number of working days

Observation 5: *The extensive work usually results to a centralized exploration strategy.*

We think that two reasons can justify why more extensive works result into more CES. First, when the work is extensive, developers try to (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 new entities. This feature of Mylyn is likely to push developers to (re)explore entities already explored in previous working sessions.

8 Threats to Validity

As any empirical study, many threats can affect the validity of our findings. We discuss them in this section.



(a) Class level (b) File level

Figure 13: Percentage of exploration strategies per number of working days

8.1 Construct Validity

Construct validity threat is related to the identification of exploration strategies 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 compromise Gini threshold on a manual validation. A manual validation is subjective and depends on the way Graphviz displays the exploration graphs. We address this threat by using an odd number of subjects. Moreover, we had few undecided validation cases: subjectivity of the manual validation process did not affect our study. For graph displays, 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.

8.2 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 un-paired 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.

8.3 Internal Validity

Internal validity threat relates to the tools used to collect interaction histories and the choice of the projects. Many tools [12, 19] can collect developers' interactions with the IDE. We use Mylyn's interaction histories because (1) Mylyn is an industrial tool provided as an Eclipse's plug-in and (2)

all contributions to Mylyn must be made using Mylyn[9], *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.

8.4 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. More investigations should be done using (1) data collected with other tools and (2) other non-Eclipse projects.

8.5 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 architecture and complexity, may help to better model and understand the underlying phenomenon.

9 Related Works

Our work on exploration strategies is related to works on (1) program explorations, (2) mining of developer's interaction histories, and (3) software engineering studies using the Gini inequality index.

9.1 Program Exploration and Tools

Robillard *et al.* [18] performed an exploratory study of program investigation behaviors using five developers performing a maintenance task on a real life software system. Using recorded videos of the developers performing the task, they compared the investigation behaviors of successful and unsuccessful developers. They concluded that, while successful developers seems to investigate the source code methodically and answer specific questions using focussed searches, unsuccessful developers exhibits a more opportunistic approach. Our study differs from Robillard *et al.*'s study because we consider the interaction history of multiple developers, gathered by the Mylyn plug-in from four open-source software projects.

The Mylyn plug-in was developed by Kersten *et al.* [4, 5, 6] to capture developers' interactions with program entities when they perform a task using Eclipse IDE. Later on, Röthlisberger *et al.* implemented SmartGroups [19] to complement Mylyn with evolutionary and dynamic information. Similar to Mylyn, CodingTracker [12] 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

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

from multiple Eclipse projects, using Mylyn are publicly available in the version control system of Eclipse.

9.2 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.* [28] mined interaction history logs to study the impact of interaction couplings on maintenance activities. They conclude that restructuring activities are more costly than other maintenance activities. Parnin and Rugaber [14] used interaction logs to identify strategies used by developers to manage task interruptions. In another work [13], they proposed a technique to extract the usage context of a task when it has been interrupted. Coman and Sillito [2] proposed an approach to automatically infer task boundaries and split developers' sessions. Schneider *et al.* [20] investigated the benefits of tracking developers'local interactions history when developing in a distributed environment. Murphy *et al.* [10] 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 [26, 27]; Ying and Robillard [26] found that the kind of a task can impact the editing style of developers and Zhang *et al.* [27] identified four file editing patterns that they argue can affect the quality of a software system. Most previous studies on Mylyn interaction histories only considered the *kinds* of the Mylyn *events*. In this work, we investigates Mylyn *events* in more details by looking at the type of program entities on which an *events* occurred.

Concerning the features and tools built from interaction histories, Singer *et al.* [23] proposed the tool NavTrack that uses *selection* and *open* events on a file to discover hidden dependencies between files and make recommendations about files that are related to a file of interest. Robbes and Lanza [16, 17] also relied on change history logs to propose a code completion technique that can reduce programmers' scrolling efforts. They also proposed the Spyware visualization tool to graphically display the change history of developers' sessions [15]. Instead of building a tool, we investigates developers' exploration strategies. The results of our study can be used to develop new features to guide developers during their explorations of program entities.

9.3 Gini Inequality Index in Software Engineering Studies

Martínez-Torres *et al.* [8] studied the structure and the evolution of virtual development teams in OSS (Open Source Software) using the Gini Inequality index. They mined the mailing list of open-source development teams and applied the Gini inequality index on the number of mails shared by contributors. They found inequalities between contributors and argued that the OSS development community is organized in a core/periphery topology where the core is composed of regular contributors and the periphery contains free rider contributors. Vasa *et al.* [24] used the Gini inequality index to analyse software metrics. They found that the Gini inequality index of software metrics hover a bounded value space and that "*theses values are remarkably consistent as a project evolves over time*".

10 Conclusion and Future Work

When developers perform a maintenance task, they always explore the program entities. Studying the way in which exploration is performed can help to improve our knowledge on developers' comprehension process and characterize developers' expertise. As a first step towards understanding exploration strategies, we mined 1,705 Mylyn's interaction histories. We applied the Gini inequality index on the number of program entity revisits. We found that developers' follow both centralized exploration

strategy (CES) and extended exploration strategy (EES) when they perform a change task. Although an EES requires more exploration effort than a CES, an EES is on average 69.39% less time consuming than a CES. We also found that work items taking up more than three days typically imply a CES.

According to the source code investigation results presented by Robillard *et al.* [18], and the opportunistic program comprehension strategy defined by Littman *et al.* [7], we argue that (1) because an EES results to less time consuming (RQ2) and more exploration effort (RQ3), an EES is followed by methodical developers and (2) on the contrary, because a CES results to more time consuming (RQ2) and less exploration effort (RQ3), a CES is followed by opportunistic developers.

We think that the strategy followed by developers depends on the complexity of the task and the systems. We plan to investigate in our future work the impact of the complexity of the task/systems on ES. Also, we conjecture that an EES could be followed by experienced developers while CES is the inexperienced developers' strategy. We plan to look at this conjecture. In addition to the relation between exploration strategy and the complexity of the tasks/projects, and developers' expertise, we will study the structure of the core entity(ies), *i.e.*, by somehow characterizing core entity(ies); they may play a specific role and be related to best practices such as design patterns. Because we think that the way developers explore a program is related to the evolution of their mental model, we will investigate the relation between exploration strategies and the understanding of a program over time, *i.e.*, if the more the developers work on a project, the more their exploration become CES or EES. We will also refine the strategy at method level and study how developers move through different kind of program entities [22]. Finally, we plan to link the interaction history to bug fixing (source code repository) to see how much is the gap between edit events in interaction histories and committed edit events.

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 validation of exploration strategies.

References

[1] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Extracting structural information from bug reports. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, pages 27–30, 2008.

[2] I. Coman and A. Sillitti. Automated identification of tasks in development sessions. In *The 16th IEEE International Conference on Program Comprehension*, pages 212–217, 2008.

[3] 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.

[4] M. Kersten. *Focusing knowledge work with task context*. PhD thesis, The University of British Columbia, 2007.

[5] 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.

[6] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1–11, 2006.

[7] D. Littman, J. Pinto, S. Letosky, and E. Soloway. Mental models and software maintenance. In *Empirical Studies of Programmers*, pages 80–98, 1986.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] C. Parnin and C. Görg. Building usage contexts during program comprehension. In *14th IEEE International Conference on Program Comprehension*, pages 13–22, 2006.

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

[15] R. Robbes and M. Lanza. Characterizing and understanding development sessions. In *International Conference on Program Comprehension*, pages 155–166, 2007.

[16] R. Robbes and M. Lanza. How program history can improve code completion. In *International Conference on Automated Software Engineering*, pages 317–326, 2008.

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

[18] 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.

[19] D. Röthlisberger, O. Nierstrasz, and S. Ducasse. Smartgroups: Focusing on task-relevant source artifacts in IDEs. In *IEEE 19th International Conference on Program Comprehension*, pages 61–70, june 2011.

[20] K. A. Schneider, C. Gutwin, R. Penner, and D. Paquette. Mining a softare developers local interaction history. In *Proceedings of the 2004 international workshop on Mining Software Repositories*, 2004.

[21] 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.

[22] S. E. Sim, S. Ratanotayanon, and L. Cotran. Structure transition graphs: An ecg for program comprehension? In *International Conference on Program Comprehension*, pages 303–304, 2009.

[23] J. Singer, R. Elves, and M. A. Storey. Navtracks: Supporting naviga-tion in software maintenance. In *International Conference on Software Maintenance*, pages 325–334, 2005.

[24] R. Vasa, M. Lumpe, P. Branch, and O. Nierstrasz. Comparative analysis of evolving software systems using the gini coefficient. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 179–188, sept. 2009.

[25] 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.

[26] A. Ying and M. Robillard. The influence of the task on programmer behaviour. In *IEEE 19th International Conference on Program Comprehension*, pages 31–40, june 2011.

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

[28] L. Zou, M. Godfrey, and A. Hassan. Detecting interaction coupling from task interaction histories. In *15th IEEE International Conference on Program Comprehension*, pages 135–144, 2007.

L'École Polytechnique se spécialise dans la formation d'ingénieurs et la recherche en ingénierie depuis 1873

École Polytechnique de Montréal

École affiliée à l'Université de Montréal

Campus de l'Université de Montréal
C.P. 6079, succ. Centre-ville
Montréal (Québec)
Canada H3C 3A7

www.polymtl.ca