Noname manuscript No.
(will be inserted by the editor)

Consensus Task Context Recommender to Guide Developers' Software Navigation

Layan Etaiwi · Pascal Sager · Yann-Gaël Guéhéneuc · Sylvie Hamel

Received: date / Accepted: date

Abstract Developers must complete change tasks on large software systems to keep them useful. When they are unfamiliar with the systems, they spend a considerable amount of time navigating through the source code, switching back and forth between files to understand the systems and locate parts relevant to their current tasks. This navigation can be difficult and consume a lot of the developers' effort and time. This navigation between files and their edition, until the completion of the tasks, generate task-specific contexts, which are sets of developers' interactions with the source code. These task contexts embody developers' time and effort.

We present and evaluate an approach that exploits tasks contexts to help guide new developers' software navigation more effectively in the context of a software family, *e.g.,* instances of SAP systems. Our novel approach, called Consensus Task Context Recommender (CTCR), recommends file(s)to-edit relevant to given tasks based on the tasks contexts obtained from previous developers who performed similar change tasks on different custom versions of the same system. Our approach uses a consensus algorithm, which takes as input a task context and recommends a consensus task context on which developers can rely to complete given similar change tasks that require editing common file(s).

To evaluate the efficiency of our approach, we perform three different evaluations. The first evaluation measures the accuracy of CTCR results. In the second evaluation, we assess to what extent CTCR can help developers by

Layan Etaiwi
Polytechnique Montréal, Montréal, Canada, E-mail: mashael.etaiwi@polymtl.ca

Pascal Sager
Zurich University of Applied Sciences, Winterthur, Switzerland, E-mail: sage@zhaw.ch

Yann-Gaël Guéhéneuc
Concordia University, Montréal, Canada, E-mail: yann-gael.gueheneuc@concordia.ca

Sylvie Hamel
Université de Montréal, Montréal, Canada, E-mail: hamelsyl@iro.umontreal.ca

conducting an observational comparative experiment in which two groups of developers performed evaluation change tasks with and without the results of CTCR. In the third and last evaluation, we compare CTCR to a state-of-the-art recommendation approach, MI Lee et al. (2015). Results with statistical significance report that CTCR can correctly recommend on average 72% of the files to be edited. Furthermore, our evaluation demonstrates that CTCR can increase developers' successful task completion rate. CTCR outperforms MI by an average of 40% higher recommendation accuracy.

Keywords Consensus Algorithm · Recommendation · Mylyn Interaction Traces · Task Context · Software Navigation · Maintenance

1 Introduction

Software companies,*e.g.,* SAP, understand the importance of tailor-made software systems that accommodate different needs, from clients' required features to technological frameworks. Accordingly, these companies develop highly customized software systems that meet their particular clients' demands better than off-the-shelf software systems and faster and more reliably than completely original systems. For example, many companies opt for adopting custom ERP software systems for managing their growing internal operations, each with its own custom version of the same ERP system.

Customised software systems grow in size and complexity, and as these systems grow, maintenance and development become more difficult, costly, and time consuming Soh et al. (2013a). Complexity, in particular, makes program comprehension challenging for most developers, specifically newcomers. Program comprehension has been reported as one of the developers' biggest problems LaToza et al. (2006). It involves tasks like reading large volume of documentation, navigating through the source-code, running the program, debugging use cases, analyzing UML diagrams and others. It may take up to 35% of the developers' time to navigate and understand source-code files for particular change tasks. Thus, it involves developers spending a valuable fraction of their time and effort exploring scattered pieces of code rather than completing their change tasks. For example, Eclipse bug report #261613 required code change in only two files, however it took the developer three days of navigating and understanding the code before making any changes Lee et al. (2015).

Some of the change tasks that developers perform on customised software systems, whether they are development, maintenance, or evolution tasks, are the same or very similar by virtue of clients having similar needs and using customised versions of the same software systems. Examples of such tasks are bugs created in the core code due to conflict of pushing new clients' customisation, exact or similar features that were created for some clients and then later requested by other clients, changes to the systems to adapt to new requirements, or implemented data designs that could be used for other clients.

Ramsauer et al. (2016). Consequently, we defined a (exact and similar) change task as follows:

Definition 1 A change task refers to either fixing bugs, improving performance, or implementing new features.

Definition 2 Exact or similar change tasks are tasks that require developers to interact with the same source-code file(s) to successfully perform the tasks.

The process of understanding a software system, locating and navigating through its source-code elements (*i.e.,* packages, files, classes, fields, methods, functions, *etc.*), and making modifications can generate events for each performed activity. When a developer completes a change task, she generates a set of events that is a developer's interaction trace (IT). When the same or similar change tasks are performed by multiple developers, each developer generates sets of events, *i.e.,* interaction traces, that overlap with the other developers'. The set of these interaction traces (ITs) from all developers creates a task context for that particular change task.

Definition 3 An interaction trace (IT) is composed of a set of events (*i.e.,* opening, searching, editing, *etc.*) performed by a developer on files while completing a change task.

Definition 4 A task context consists of a set of developers' interaction traces.

Task contexts have been used to study and build tools that can assist developers cope with the challenge of effectively and efficiently developing software systems. In particular, some works used developers' task contexts to present project elements with which developers frequently interacted to create exploration strategies and investigated how developers understand programs Kersten and Murphy (2006); Sahm and Maalej (2010); Soh et al. (2013b). Other works focused on mining interaction traces to suggest changes according to association rules Lee et al. (2015); Singer et al. (2005); Zimmermann et al. (2004), recommending navigation patterns DeLine et al. (2005), and clustering interactions to recommend clusters of textually related elements Lee and Kang (2013).

While these works help developers complete their tasks by recommending elements, none considered supporting customised software systems and they fail to provide developers with recommendations generated with high accuracy specifically for particular change tasks Lee and Kang (2011); Robbes et al. (2010). For example, when Lee and Kang (2011) evaluated their approach against Team Tracks DeLine et al. (2005), Team Tracks recommended three methods, none of which were required for completing the change task. These approaches suppose that developers come with at least a minimal knowledge of the systems and require them to start interacting with task-related elements before making their recommendations. Most of them also build their recommendation methods on association rules between elements and frequency,

which could ultimately lead to recommending unrelated elements if developers interacted with the wrong elements.

More importantly, we argue that it is unrealistic to assume that developers start interacting with relevant elements without experience with a system. For example, when a newcomer joins a project and is recommended with a set of relevant elements based on accumulated system interactions from more experienced colleagues, she should be able to navigate and understand the related parts of the system and complete her assigned tasks without the need for prior experience. Thus, we want to produce highly-relevant recommendations that can guide developers' navigation without requiring them to have prior system knowledge.

We propose consensus task context recommender (CTCR) to recommend file(s)-to-edit to newcomers based on an aggregated set of more experienced developers' system interactions. The purpose of our approach is to recommend files that are relevant to a given set of change tasks, Thus, our approach targets developers' tasks contexts for the same or similar tasks on customised systems as input data rather than interaction traces from systems as whole. By applying a consensus algorithm to tasks contexts, our approach creates consensus tasks contexts as recommendations. Each recommendation comprises a consensus set of file(s)-to-edit that are relevant and help perform new change tasks that are similar to the input tasks contexts.

Our general hypothesis is that CTCR can help those who are unfamiliar with their current system to guide their navigation through files in the context of a change task, complete their tasks successfully, with substantially less time and effort, minimal navigation to unrelated files, and ultimately increase their productivity.

To investigate this hypothesis, we conduct a series of evaluations. We determine the accuracy of our results by defining a ground truth data as a basis of comparison and using precision and recall metrics as performance measurements. We also evaluate to what extent gathered tasks contexts from previous developers can help newcomers navigating through software systems and quantitatively analyze success rate. We conduct an observational comparative experiment of 30 developers undertaking identical change tasks with and without CTCR recommendations. Lastly, we compare CTCR against MI Lee et al. (2015), an existing file-level recommendation approach.

- Quantitative results when comparing to the ground truth indicate that our approach can recommend file(s)-to-edit with average precision of 72%, recall of 61%, and F-measure of 60%.
- A detailed qualitative analysis of the experiment supported by video recordings reveals that developers with CTCR recommendations can complete their tasks in/with less than half of the time and effort needed by the control group, and with higher completion rate of 95%.
- The experiment shows that developers with CTCR recommendations have an increased performance at comprehending and navigating through sys-

tem elements. In contrast, the control group spends a considerable amount of time following unstructured navigation relying on guessing and glancing.
– The comparison demonstrates that CTCR returns higher recommendation accuracy than MI Lee et al. (2015).

The rest of the article is structured as follows: in the next section, we support our work with a motivating example. In Section 3, we describe our approach and an overview of the consensus algorithms. In Section 4, we implement a study to collect developers' ITs and apply the consensus algorithm to generate recommendations. empirical study setup. In Section 5, we apply three evaluation methods to investigate the success of the approach. In Section 6, we report and discuss results. In Section 7, we discuss limitations. Finally, in Section 8, we review the related work and, in Section 9, we conclude the article and discusses plans for future works.

2 Motivating Use Case

To illustrate the motivation and potential benefit of our approach, we consider the scenario of a new software developer, Alice, who has been recently hired as a software support engineer at our company. She has been assigned to the Environmental Analysis Software, which is one of the many large software systems that the company offers. Her role consists of enhancing the software, troubleshooting, and identifying solutions for technical issues.

As a large software company, we build customised software systems to help small and large businesses deal with rapid technology advances, and resolve their very specific needs. We encourage our developers to collect their interactions in the form of events with the system when performing any type of programming task.

Through our defect management tool, client Bob reported an access bug on one of the configuration pages. The bug is caused by an error in the default value of the auto start function.

As part of correcting this defect, Alice started investigating the source code prior to making any modifications. Using an IDE, she tried to find all the software elements that are related to implementing the configuration page, and then to inspect the functions that could be related to specifying the auto start value. The package explorer displays hundreds of files. She faces the daunting task of navigating through them and identifying related files. Eventually, after spending a significant amount of her time investigating the very large code base, exploring few related and many unrelated files, and reaching a dead-end failing to locate the file and the function related to the error, Alice decided to seek help from senior colleagues. Her colleagues shared with Alice collected events generated from fixing a related bug for another client. However, the number of events in their ITs is large and overwhelming for Alice to navigate and identify related files. Alice needs an approach that can help her understand the relevant part of the system better by providing the most relevant files to her task. She would benefit from an approach that aggregates her colleagues'

ITs, collected while completing the same or similar change tasks, and recommends her with one consensus task context that contains the file(s)-to-edit most relevant to the particular task that she is completing.

Newcomers working on subsequent tasks could query through the recommended consensus tasks contexts to identify which files could be related to resolving the task at hand, therefore enhancing program comprehension, reducing the time and effort required, and helping them be more productive.

3 Approach: Recommending Based on Task Context

We present in the following an overview of our approach, CTCR. Figure 1 illustrates how our approach can be incorporated into the motivating use case. Having a customised software system branched into different clients' instances, developers perform same or similar change tasks on each client's instance. Developers' interactions while completing these tasks are collected via a tool, *e.g.,* Mylyn. These interaction traces are then extracted, pre-processed, and formed into tasks contexts. The consensus algorithm is then applied to the task context of each change task to generate a consensus task context that contains a set of relevant file(s)-to-edit and that can be recommended to new developers to help them complete same or similar change tasks of other clients' instances.

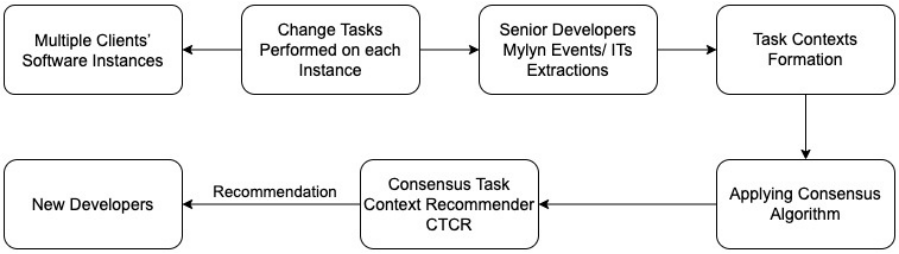


Fig. 1: Overview of the Approach

Following the motivating use case, Figure 2 exemplifies how a task context gets formed. A set of developers $\{D_1, D_2, D_3, D_4\}$ completed an access bug task T on the configuration page on different clients' system instances using Eclipse. Each activity performed on a system file for the completion of the task counts as an event $\{e_1, e_2, ..., e_n\}$. Mylyn collects and aggregates the sequence of events from each developer into interaction traces $\{IT_1, IT_2, IT_3, IT_4\}$. The set of developers' ITs forms a task context for change task T. This aggregation of developers' events into task context allow us generate a recommendation based on developers previous interactions with a system using a consensus algorithm that is able to produce a set of the most relevant files.

In the rest of this section, we discuss the history of consensus algorithms, define the algorithms and measures used in the following.

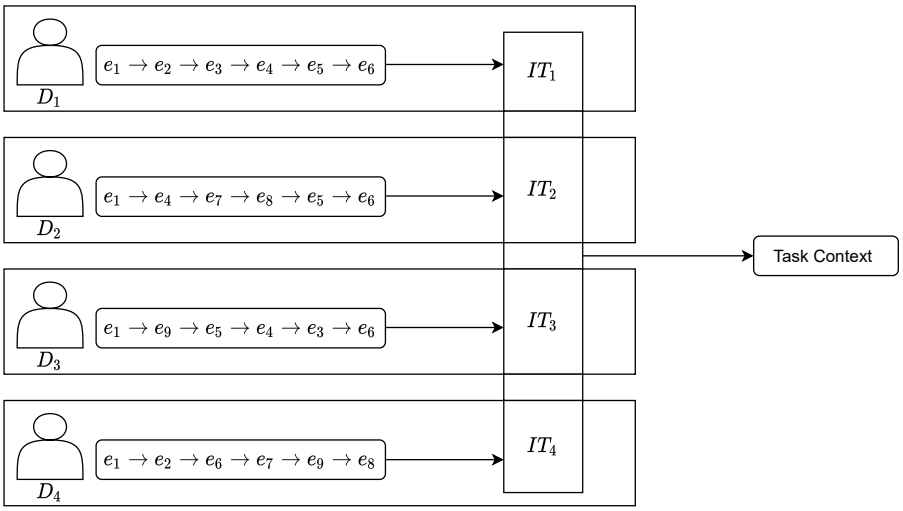


Fig. 2: Illustration of Task Context Formation

3.1 Overview of the Consensus Algorithms

Consensus algorithms have been around for a long time. They were first investigated about two centuries ago in the context of voting and elections, in which voters provide their preferences on a set of candidates and the algorithm needed to provide a single ranking of the candidates that would reflect the consensus of the voters' preferences Kemeny (1959).

Formally, from a set of N different ordering of the same n elements, each ordering being called a ranking of the n elements, the problem is to find one consensus ranking, *i.e.,* an ordering of the n elements that is the closest to all N rankings under a chosen distance Ali and Meilă (2012).

In real life applications, rankings can be incomplete. This happens when not all the n elements are ordered in every ranking. For example, in elections, a voter could have chosen not to take into account some of the candidates in her ranking. To deal with incomplete rankings, works proposed normalization techniques Brancotte et al. (2015). Before finding the consensus of a set of rankings, the projection technique keeps, for each ranking, only the common elements that exist in all rankings while the unification technique adds missing elements from each ranking at the end of them. Rankings can also be not strictly-ordered, when some elements are ranked at the same position, *i.e.,* are

tied. For example, in elections, a voter could have chosen to put more than one candidate in first position.

3.2 Measures

To measure the distance between two rankings, various measures have been suggested Critchlow (1985). Most works proposed the use of the generalized form of the Kendall-τ distance Brancotte et al. (2015); Cohen-Boulakia et al. (2011) when dealing with a set of incomplete, not strictly-ordered, rankings. The generalized Kendall-τ distance, G, between two rankings r and s, is:

$$G(r,s) = \#\{(i,j) : i < j \wedge$$
$$((r[i] < r[j] \wedge s[i] > s[j]) \vee (r[i] > r[j] \wedge s[i] < s[j]) \vee \textbf{(1)}$$
$$(r[i] \neq r[j] \wedge s[i] = s[j]) \vee (r[i] = r[j] \wedge s[i] \neq s[j]))\} \textbf{(2)}$$

It sums the number of times elements i and j appear in different orders in the two rankings (1), or (2) the number of times the two elements are tied (in one bucket) in one ranking but not in the other.

The generalized Kemeny score is the sum of the generalized Kendall-τ distance between a given ranking and all rankings in the set. Given a set of rankings with ties R, the generalized Kemeny score K is:

$$K(r, \mathcal{R}) = \sum_{s \in \mathcal{R}} G(r,s).$$

An optimal consensus ranking, denoted $r*$, for a set of rankings R is:

$$\forall r \in \mathcal{R}_n : K(r^*, \mathcal{R}) \leq K(r, \mathcal{R}).$$

3.3 Consensus Algorithms

Consensus algorithms have been applied in different domains such as bioinformatics Cohen-Boulakia et al. (2011), databases Fagin et al. (2004), artificial intelligence Pennock et al. (2000), as a means to bring forward interesting information coming from different rankings used in these domains.

In particular, consensus algorithms were applied with varying results in bioinformatics. Brancotte et al. (2015) studied 14 different consensus algorithms using the generalized Kendall-τ distance and classified them into score-based algorithms and positional-based algorithms. The foremost searches for a consensus by focusing on the disagreement between the order of the elements, while the positional-based algorithms focuses on the position of the elements in each ranking.

Brancotte et al. (2015) extensively compared and studied all the ranking algorithms with experiments on real, synthetic, and differently-sized datasets

from different fields. The outcomes of the experiments showed that the BioConsert algorithm Cohen-Boulakia et al. (2011) outperforms the other algorithms providing highest quality results on both real and synthetic datasets. KiwikSort Ailon et al. (2008) comes second after BioConcert, especially when the dataset is extremely large ($n > 30,000$).

While both algorithms are score-based algorithms, they differ in the way they construct the consensus ranking. BioConcert uses a local search. Given a set of rankings, it randomly selects one of them as a starting ranking and then continuously applies two operations until the generalized Kemeny score is stabilized. The two operations are (1) changeBucket: moves an element from one bucket and adds it into an existing bucket and (2) addBucket: moves an element from a bucket to put it in a new bucket Cohen-Boulakia et al. (2011).

KwikSort uses a divide-and-conquer approach. It randomly assigns one of the elements as a pivot and then recursively places the rest of the elements in two buckets after and before the pivot until a consensus ranking is reached.

4 Evaluation Setup

We perform three evaluations to assess the accuracy of the results of our approach and the extent to which CTCR can improve newcomers' productivity changing a software system. These evaluations are quantitative, qualitative, and a comparison to answer the following research questions:

RQ1 To what degree does CTCR recommend relevant files to given change tasks? We answer this question by building ground truth data and quantitatively comparing them with the results of CTCR using precision and recall measures (in Section 5.1).

RQ2 Given a change task, can CTCR help guide newcomers' navigation paths to relevant file(s)-to-edit and increase their productivity? To evaluate productivity and navigation behaviour, we conduct an observational comparative experiment of 30 developers performing evaluation change tasks with and without the recommendations of the CTCR (in Section 5.2).
in

RQ3 How does CTCR compare to MI (Mining Programmer Interaction Histories) Lee et al. (2015) in recommending relevant file(s)-to-edit for specific change tasks? We answer this question by comparing the results of our approach with MI recommendation results under the same set of conditions (Section 5.3).

To perform the evaluations and answer these RQs, we collect ITs from performing change tasks and generate the recommendations. Thus, we must first choose a subject system (in Section 4.1), which is the same system that will be used for the evaluation (in Section 5). Then we choose change tasks (in Section 4.2), recruit participants (in Section 4.3), and select tools for collecting the ITs (in Section 4.4). We then can collect and pre-process participants' interaction traces (in Sections 4.5 and 4.6). Finally, we apply the consensus

algorithms on these ITs to obtain recommendations (in Section 4.7)). To build the ground truth data, we use Mylyn ITs that are attached to the chosen change tasks tickets (in Section 4.8).

4.1 Subject System

Among a population of Java systems, we chose an Eclipse-based plugin, PDE (Plug-in Development Environment), as subject system. PDE[1] offers tools to create, develop, test, debug, build, and deploy Eclipse plug-ins, fragments, features, and update sites. It comprises approximately 2M LOC, and 4,000 classes scattered across 64 sub-projects. We use PDE because (1) it is open source, which we can use its source code freely, (2) its base code is big enough to exemplify real systems, (3) it has been used in many software engineering research studies, and (4) Mylyn ITs are attached to most of its fixed bug and completed feature request tickets, which we will use later for creating the study change tasks. PDE consists of many sub-projects and we chose to consider only the PDE-UI sub-projects. Participants will interact with PDE-UI files to complete change tasks.

4.2 Change Tasks

To define a set of change tasks for participants to perform, we explored completed tickets related to the PDE-UI in the Eclipse Bugzilla[2]; Web based bugs, and request tracking system. We browsed many tickets by reading their descriptions, replicating the bugs on the system when possible, and implementing the proposed solutions.

Some of the tickets were impossible to replicate because their description was not detailed enough; some were lengthy, requiring a few hours to complete; while some required minutes to an hour. For a ticket to be selected as a candidate change task, it had to be marked as completed or fixed, contain a detailed description and a complete solution patch, be reasonably difficult, and complemented by Mylyn ITs.

To measure the time needed and difficulty of a set of candidate tickets, we hired a Ph.D. student with approximately five years of industry experience as evaluator. The evaluator went through the candidate tickets and assessed their complexity level and how much time each ticket needed to be completed. Then, we randomly chose two moderately difficulty tickets that took no more than 45 minutes to complete as our change tasks.

Soh et al. (2018) performed a similar study on Eclipse PDE-UI, invited four participants to perform a change task on the system while video recording their screens, and collected their Mylyn ITs. We take advantage of the change

[1] `https://www.eclipse.org/pde/`

[2] `https://bugs.eclipse.org/bugs/`

task used in Soh et al. (2018), adapt it as our third change task and use the collected ITs as part of our dataset.

Table 1 presents detailed descriptions of the chosen change tasks.

Bugzilla Ticket #	**Task**	**Description**
304028	Task 1: Feature properties dialog window has no title	Click on the contents tab of a product configuration page, select one of the features, and then click on the properties button. The properties dialog window has no title.
229024	Task 2: A tab on the overview page shows "?" Instead of API Information	On the overview page of an extension point schema, one of the tabs' names is a question mark. The name instead should be "API Information". PDE here is not recognizing APIINFO as an attribute.
265931	Task 3: Autostart values are not persisted correctly on the plug-in	Add a plug-in and set autostart to "true". Save the file. Open the file in a text editor, and see how the value of the "autostart" attribute is still set to false.

Table 1: Change Tasks Used in the Study and their Descriptions

4.3 Participants

Considering the size and complexity of PDE-UI, we recruited only participants with experience with Java and the Eclipse IDE to guarantee the reasonable successful completion of the assigned tasks and to collect participants' ITs.

We began the recruitment process by sending out emails to the contact list of some research groups in the department of Computer and Software Engineering at Concordia University and Polytechnique Montréal. The emails contain a link[3] to an online form collecting information about their gender, level of education, and their years of Java and Eclipse IDE experience.

15 participants filled out the form, seven of them have over three years of Java experience, while the remaining eight participants have less than three years. All participants have at least one year experience with the Eclipse IDE. We selected the seven Java experienced developers to participate and complete the two change tasks. Among these seven participants (referred to as $P1$,..,$P7$), two are female, one is a postdoctoral researcher, three are doctoral candidates in software engineering, three are enrolled in a Master program in computer engineering, and all have 1 to 5 years of professional development experience. All of our participants were newcomers to the system, they never had worked with/on Eclipse PDE.

[3] `https://docs.google.com/forms/d/e/1FAIpQLSep_AN46h8AYBKHDUelcyfm7P-Zi-v7_IVAuEq_6T8i-jPdgg/viewform?usp=sf_link`

We sent an invitation email to each individual participant containing a brief description about the experiment and schedules for performing the tasks. To avoid time conflicts, we scheduled each participant on a different date.

4.4 Interaction Traces Tools

Integrated development environments (IDEs) support developers' activities on software systems. Numerous IDEs exist for various programming languages. However, the most used Java IDEs are Eclipse, IntelliJ IDEA, and NetBeans. In this work, we use Eclipse IDE[4].

Developers' Interaction traces (ITs) with software systems are collected by task management and monitoring tools, such as Mylyn[5], Blaze Fritz et al. (2014), FeebBaG Amann et al. (2016), or DFlow Minelli et al. (2014). Blaze and FeedBag are Visual Studio extensions, while DFlow is a Pharo extension. Therefore, we chose to use events generated by Mylyn because (1) Mylyn is an Eclipse extension and (2) it is the monitoring tool most used by developers and researchers Soh et al. (2018).

Mylyn is an Eclipse plugin that monitors and collects developers' interaction events with system elements while performing a change task. It starts collecting events after developers create and activate a Mylyn task for the change task on which they are working. It stops gathering events once the developers deactivate the Mylyn task. Then, it aggregates the collection of events in the form of an IT. ITs can be compressed, encoded, and exported in XML format.

Mylyn ITs consist of consecutively-performed events with system elements to accomplish a task. There are eight different kinds of events: Selection, Edit, Command, Attention, Manipulation, Prediction, Preference, and Propagation[6]. Selection, Edit, and Command are directly triggered by the developers, while the others are indirect events, triggered by Mylyn. We only consider direct events.

Mylyn captures nine attributes for each event, out of which we use the four following: (1) StructureHandle: a unique identifier of the project elements being worked on; (2) Kind: type of event; (3) StartDate: when the event started; (4) EndDate: when the event ended;

An example of two consecutive events is shown in Table 2.

StartDate	EndDate	StructureHandle	Kind
2018-08-08 11:43:44.97 EST	2018-08-08 11:46:09.716 EST	FeatureSection.java	Selection
2018-08-08 11:46:46.918 EST	2018-08-08 11:53:39.320 EST	FeatureSection.handleProperties();	Edit

Table 2: an Example of Mylyn Events

[4] https://www.eclipse.org/

[5] http://eclipse.org/mylyn/

[6] https://wiki.eclipse.org/Mylyn/Integrator_Reference

4.5 Interaction Traces Collection

We collected participants' ITs with the system by asking each participant to perform the change tasks on PDE-UI in a laboratory at a specific time, on the same computer, under the same settings and using the same procedure. We thus could control the ITs collection and ensures that participants were not distracted or interrupted.

The participants performed their change tasks on a desktop computer running Windows 10 with dual 28" flat monitors. The source code of the PDE system along with the change tasks are imported into an Eclipse IDE v4.10.0 workspace with the Mylyn plugin installed.

Before they began performing their change tasks, we created, in the IDE, for each participant, a Mylyn task for each change task. As shown in Figure 3, on the left side, the PDE system is imported to the IDE and Mylyn tasks are created and ready to be activated under Task List on the right side.

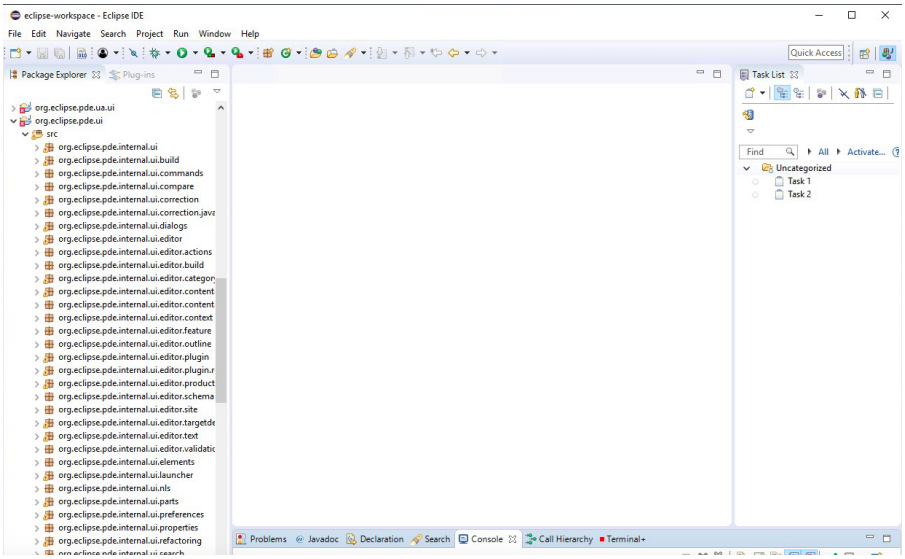


Fig. 3: A Screen Capture of the IDE Before the Start of a Change Task

Then, we explained to each participant the purpose of the work, directed them to the desktop station, and informed them that they would perform two change tasks. Each task was given up to 45 minutes to complete with up to 15 extra minutes if needed. We instructed participants that there was no right or wrong solution to each task and advised to try to complete the change tasks successfully. Participants had the choice to stop their participation at any time for any reason. We stayed in the laboratory to assist in case of a technical problem. However, we told participants that they could not ask programming questions related to the completion of the change tasks.

We gave participants a sheet of paper describing each task, and providing detailed instructions on how to replicate the bug to detect the current behaviour before they start making changes to the source code; a copy of the distributed sheet is available online on our companion Web site[7]. To provide the participants with a hint of what they had to do and type of tasks, we created a demo change task. We gave the participants 20 minutes to replicate the bug described in the demo task, and they were not required to provide a solution for the task.

Once participants were ready to start the first task, we asked them to activate the related Mylyn task and start navigating their ways through the source code. When the Mylyn task was activated, Mylyn started collecting events. After the successful completion of the change task, participants deactivated the related Mylyn task to stop the collection of events. Successfully, all participants could complete the change tasks. On average, they spent 37 minutes on the first task, while they carried out the second task in 33 minutes.

We then exported all Mylyn events from Eclipse IDE in XML format. Obtained events from the completion of Change Task 1 and 2 by seven participants, together with events related to Change Task 3 from Soh et al. (2018) by four participants sum a total of 2,390 events. Figure 4 represents a sample of an extracted Mylyn event.

```
<?xml version="1.0" encoding="UTF-8"?>
<InteractionHistory Id="local-2" Version="1">
  <InteractionEvent
    Delta="null"
    EndDate="2018-08-08 12:03:46.865 EDT"
    Interest="22.0"
    Kind="edit"
    Navigation="null"
    OriginId="org.eclipse.jdt.ui.CompilationUnitEditor"
    StartDate="2018-08-08 11:57:15.801 EDT"
    StructureHandle="=org.eclipse.pde.ui/src&amp;lt;org.eclipse.pde.internal.
    ui.editor.product{FeatureSection.java[FeatureSection~handleProperties"
    StructureKind="java"
    NumEvents="22"
    CreationCount="161"/>
  <InteractionEvent
```

Fig. 4: Sample of Mylyn Event in XML Format

4.6 Interaction Traces Pre-processing

We pre-processed each exported Mylyn event to extract participants' selection and edit activities and system elements on which the activities occur.

[7] https://www.ptidej.net/downloads/replications/emse22a/

This phase goes through multiple steps and starts by converting the extracted Mylyn XML files into CSV files.

As described in Section 4.4, there are eight types of events. Edit events are released when developers either select or edit text in a file in Eclipse IDE, while selection events are triggered when developers open a file. Any Mylyn triggered events are therefore removed from all CSV files.

Mylyn specifies the system elements on which events were performed in the StructureHandle attribute. System elements are divided in the StructureHandle into: project name, package, file, class, attribute or method, and the others Soh et al. (2013a). Figure 5 shows the parts of the StructureHandle against a real StructureHandle taken from one of the participants' Mylyn events, while Table 3 identifies the parts of the StructureHandle.

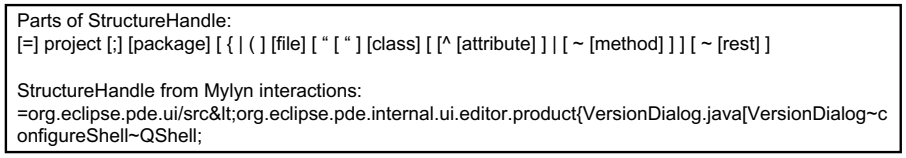


Fig. 5: Parts of Mylyn StructureHandle.

Part Name	**Matching part form StructureHandle**
Project	org.eclipse.pde.ui.src
Package	org.eclipse.pde.internal.ui.editor.product
File	VersionDialog.java
Class	VersionDialog
Method	configureShell
Rest	QShell

Table 3: Identification of the Parts of the StructureHandle in Figure 5

The paths to elements in the StructureHandle contain special characters that created noise and made it difficult to obtain the actual full paths. We implemented a tool that uses regular expressions to identify the parts of StructureHandle and either remove or replace these special characters with dots. The tool outputs a readable CompleteName for each StructureHandle that contains no special characters. Figure 6 compares a path to a system element as exported in the StructureHandle versus the path CompleteName after the removal of special characters.

The studied system contains some JAR files that were not related to the completion of any change tasks. Given that JAR files are irrelevant and rather could add noise to the tasks contexts, all participants' events related to JAR files were therefore removed from all Mylyn ITs.

StructureHandle:
=org.eclipse.pde.ui/src<org.eclipse.pde.internal.ui.editor.product{VersionDialog.java[VersionDialog~configureShell~QShell;

CompleteName:
org.eclipse.pde.ui.src.org.eclipse.pde.internal.ui.editor.product.VersionDialog.java.VersionDialog.configureShell.QShell

Fig. 6: StructureHandle vs. CompleteName after Special Characters Removal.

According to Lee et al. (2015) and Sanchez et al. (2015), any selection and edit events with 0-duration should be considered noise, related to developers mouse-clicking in a file. Considering that the purpose of this work is to recommend to developers a consensus task context that encompasses the most relative file(s)-to-edit, we removed all 0-duration events.

In the next step of pre-processing, we compared each event StructureHandle along with its type (*i.e.,* selection or edit) among all participants' events for each change task individually. Any event containing the same StructureHandle path and type was given the same unique ID. For example, if participant $P2$ made an edit on a method in a particular Java file with a specific StructureHandle, and participant $P4$ performed the same edit, then both of these events were given the same ID number. Figure 7 compares two screenshots taken from the interaction files of participant $P2$ and $P4$ for Change Task 1. Events 26 and 27 were performed by the two participants on exactly the same StructureHandle, hold the same type, and accordingly are assigned the same ID number.

Participant P2		
ID	**StructureHandle**	**Kind**
26	=org.eclipse.pde.ui/src<org.eclipse.pde.internal.ui.editor.product{VersionDialog.java[VersionDialog~VersionDialog~QShell;~Z~QString;	Selection
27	=org.eclipse.pde.ui/src<org.eclipse.pde.internal.ui.editor.product{VersionDialog.java[VersionDialog~VersionDialog~QShell;~Z~QString;	Edit
141	=org.eclipse.pde.ui/src<org.eclipse.pde.internal.ui{PDEPlugin.java	Selection
Participant P4		
ID	**StructureHandle**	**Kind**
86	=org.eclipse.pde.ui/src<org.eclipse.pde.internal.ui{PDEUIMessages.java[PDEUIMessages^VersionDialog_title	Selection
26	=org.eclipse.pde.ui/src<org.eclipse.pde.internal.ui.editor.product{VersionDialog.java[VersionDialog~VersionDialog~QShell;~Z~QString;	Selection
27	=org.eclipse.pde.ui/src<org.eclipse.pde.internal.ui.editor.product{VersionDialog.java[VersionDialog~VersionDialog~QShell;~Z~QString;	Edit

Fig. 7: Illustration of Common Events between Participants Hold the Same ID Number.

Mylyn events relate to two levels: method-level events and file-level events. Method-level events occur in/on classes, fields, and methods. File-level events occur on Java files, such as opening or editing a file. Considering that our approach aims to recommend file(s)-to-edit, we therefore keep only file level events and remove those on method level.

All collected and pre-processed participants' events that are used to generate CTCR recommendations are available online[7].

4.7 Applying Consensus Algorithms to the Tasks Contexts

Considering that the BioConcert and KwikSort algorithms provide best quality results (see Section 3.3) and the number of rankings (interaction traces) are less than 100 in our dataset, we choose to apply the two algorithms to generate consensus tasks contexts. We later compare the results from both algorithms to determine if one of the algorithms can possibly provide higher quality results in the case of our dataset.

The rankings, *i.e.,* participants' interaction traces in each task context, in our dataset are incomplete rankings, thus we apply the unification normalization technique to complete the rankings. We do not use the projection technique as it leads to the removal of events that could be relevant. The unification technique adds a bucket at the end of each participant's IT that contains events that appear in other ITs but not in this particular IT. For example, assume we have interaction traces of participants $P1$ and $P2$:

$$IT1 = [[16],[8],[5],[6],[7],[22]]$$
$$IT2 = [[18],[16],[19],[20],[5],[22]]$$

The application of the unification process produces the following ITs:

$$IT1 = [[16],[8],[5],[6],[7],[22],[18,19,20]]$$
$$IT2 = [[18],[16],[19],[20],[5],[22],[8,6,7]]$$

Table 4 shows consensus task contexts after applying both algorithms on some participants' task context for a change task T. The way both algorithms process inputs and construct the consensus is relatively similar. Specifically, they order the significant relevant files in a way that minimizes the disagreement between the set of input ITs and groups the less relevant files in a single bucket at the end of the consensus.

After examining the files in the last buckets of all the results, we observed that these files are definitely irrelevant to the successful completion of the task because all are selection events performed by one or two participants as part of code comprehension. Therefore, we chose to always ignore the last bucket in our approach.

BioConcert	[[4], [5], [6], [9, 10], [12], [1, 2, 3, 7, 8, 11, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34]]
Kwiksort	[[4], [5], [6], [29, 1, 2, 3, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 30, 31, 32, 33, 34]]

Table 4: CTCR after Applying BioConcert and Kwiksort to the Task Context of Task T.

Comparing the consensus results of applying BioConcert and Kwiksort shows that they are almost identical with minor differences. BioConcert outperforms all the other consensus algorithms in quality, therefore we chose to use BioConcert in our approach CTCR.

Table 5 translates the results of CTCR from the BioConcert algorithm of Task T into real participants' events. The rest of the results of the BioConcert algorithm along with translations are available in project repository[7].

ID	Action	
4	Selection	org.eclipse.pde.ui
5	Selection	org.eclipse.pde.ui.src.org.eclipse.pde.internal.ui.editor.product.PluginConfigurationSection.java
6	Edit	org.eclipse.pde.ui.src.org.eclipse.pde.internal.ui.editor.product.PluginConfigurationSection.java
9	Selection	org.eclipse.pde.core.src.org.eclipse.pde.internal.core.product.PluginConfiguration.java
10	Edit	org.eclipse.pde.core.src.org.eclipse.pde.internal.core.product.PluginConfiguration.java
12	Selection	org.eclipse.pde.ui.src.org.eclipse.pde.ui.launcher.PluginsTab.java

Table 5: Translation of recommended CTC from Applying BioConcert into Real Developers' Events.

4.8 Ground Truth

We create ground truth data to compare them with the results of applying the consensus algorithm to interaction traces to measure accuracy. A ground truth is the ideal set of files with which a developer should interact to complete a particular change task.

We derived the ground truth data by using Mylyn ITs attached to the Bugzilla tickets used to create the change tasks (Section 4.2). We applied the same pre-processing steps that we adopted in Section 4.6 to clean and extract only edit and selection events from the attached ITs. Table 6 presents the ground truth data of Change Task 3. The remaining ground truth data for the other two tasks are available online[7]

File Name
PluginConfigurationSection.java
IPluginConfiguration.java
Product.java
PluginConfiguration.java

Table 6: Ground Truth Data Created from Change Task 3 Ticket.

5 Evaluations

We now explain how we perform the three evaluations to answer our RQs.

5.1 *RQ1*: To what degree does CTCR recommend relevant files to given change tasks?

We evaluate the quality of the results of CTCR to determine whether or not the consensus algorithm can recommend accurate results by comparing the recommendations obtained in Section 4.7 against the created ground truth data as explained in Section 4.8.

To measure the accuracy of the recommendations and answer RQ1, we use three quantitative measures: precision, recall, and F-measure, which are commonly used for evaluating the quality of the results of recommendation systems Lee and Kang (2013). We calculate precision P, recall R, and F-measure F as follows Avazpour et al. (2014):

Precision P represents the proportion of recommended elements that are correct. The higher the precision, the more accurate the elements recommended by CTCR.

$$P = \frac{TP}{TP + FP}$$

Recall R represents the proportion of recommended elements that are actually met. The higher the recall, the more elements that are actually recommended by CTCR.

$$R = \frac{TP}{TP + FN}$$

F-measure (F) represents the accuracy of the recommendation. The higher the F-measure, the more accurate the results of CTCR.

$$F = \frac{2 \times P \times R}{P + R}$$

with TP (true positives) the recommended elements that are relevant, FP (false positives) the recommended elements that are not relevant, and FN (false negatives) the relevant elements that are not recommended.

5.2 *RQ2*: Given a change task, can CTCR help guide newcomers' navigation paths to relevant file(s)-to-edit and increase their productivity?

We answer this RQ by conducting a qualitative evaluation with 30 experienced developers performing a set of evaluation change tasks.

Setup. We use the same system, PDE-UI, as in Section 4.1 to perform this evaluation.

Due to the COVID-19 pandemic, this evaluation changed from a laboratory setting experiment to a remote observational experiment. We installed the PDE system on a laboratory computer. After comparing the image quality of a few remote desktop services under different internet speed and bandwidth, we chose Microsoft Remote Desktop service as it was the only service that did not require a fast internet connection to maintain a high quality image and data transfer. Thus, we requested developers to install Microsoft Remote Desktop service on their computers and we granted them a full remote access control to the laboratory computer to perform the evaluation change tasks. Furthermore, we captured video recordings of the developers' screen during the experiment using VLC Media Player.

Evaluation Change Tasks. We chose evaluation change tasks that are similar to the ones in Section 4.2 and that have Mylyn ITs attached to them. To obtain such tasks, we examined tickets on the Eclipse Bugzilla tracking system in three phases.

In a first phase, we created a search query to return tickets that meet the following criteria: (1) PDE product tickets, (2) UI component tickets, (3) status is set to resolved, (4) resolution is set to fixed, and (5) attachments contain a patch file. The patch file is needed to help us examine a proposed fix for each ticket and evaluate the tickets' complexity in the last phases.

In a second phase, we analyzed the extracted tickets and kept the tickets that shared the same context as the one in Section 4.2.

In a third phase, two of the authors went through the candidate tickets randomly, read the description and attached patches, and categorized the complexity of the tickets into easy, moderate and difficult. For equitable evaluation, we targeted moderate complexity tickets that require source-code modification in one to three files. We also asked two Ph.D. students, with some professional experience, and unfamiliar with the subject system to perform some of the selected tickets and assess their complexity and if they could complete them within 30 minutes

Finally, we considered six evaluation change tasks, which are described in Table 7.

Developers. We contacted developers to perform the evaluation change tasks and measure their productivity. To invite developers to participate in the experiment, we followed the same recruitment process as in Section 4.3.

We sent out invitation emails to software engineering research groups from four universities (Concordia University, Polytechnique Montréal, Zürich University, and Zürich University of Applied Sciences). The email provided them with a registration form to gather relevant educational and programming information. We used this information to select developers with different educational levels and programming experience to guarantee the generalisability of our approach and obtain results with a variety of problem-solving methods.

Bugzilla Ticket #	**Task**	**Description**
269618	Automatic wildcard on plug-ins	When searching for a plug-in via a string, you will have to input ** around the string. Fix the behaviour to accept wildcard strings without the **.
144533	Unnecessary white space on configuration tab	Remove the unnecessary white space on the configuration tab.
88003	Select all property	Add "All" property to the plug-ins view.
261878	Prompt to save changes on Plug-ins	When on the plug-ins page, it prompts you to save changes while you have not done any changes.
171767	Large font on main tab	When increasing the dialog font size, part of the main tab disappears.
101516	Sort alphabetically property	It would be helpful to be able to sort the extensions listed in the extensions section of the plug-in XML editor alphabetically.

Table 7: Evaluation Tasks Description.

We selected 34 developers (referred to as $D1$,..,$D34$), out of whom four abandoned the experiment. Of these 30 developers, three were senior undergraduate students, 17 were M.Sc. students, and ten were Ph.D. students. 53% of the developers had programming of over 5 years, with an average of 3 years of Java programming experience. All developers had industrial programming experience (six of them with more than 5 years, while the remaining 24 had between 1 to 5 years). All developers reported using different IDEs and being unfamiliar with the subject system.

Procedure. We applied the Between-Subjects experiment design Erlebacher (1977) in which there is a control group and an experimental group, and each participant experiences only one level of a single independent variable. Our independent variable is the recommendations with two levels: with and without. We split the 30 developers into a control group and an experimental group (15 developers in each group). For each group we made sure that developers' education and professional experience varied. In the control group, we requested developers to perform the evaluation tasks without the recommendations of CTCR, while we provided developers in the experimental group with the recommendations.

To ensure that the set of evaluation tasks are performed by developers with different experience levels, we split the six tasks into two sets (A and B with three evaluation change tasks in each set). Therefore, each control and experimental group was further divided into two sections, with each section performing a different set of evaluation tasks. Figure 8 illustrates this division.

The evaluation experiment took about of two months to be completed. We scheduled each developer on a particular day of their choice. Before the start of the experiment, we emailed the tasks description file and audio-called the developers via the Zoom conferencing software. The purpose of the call was to

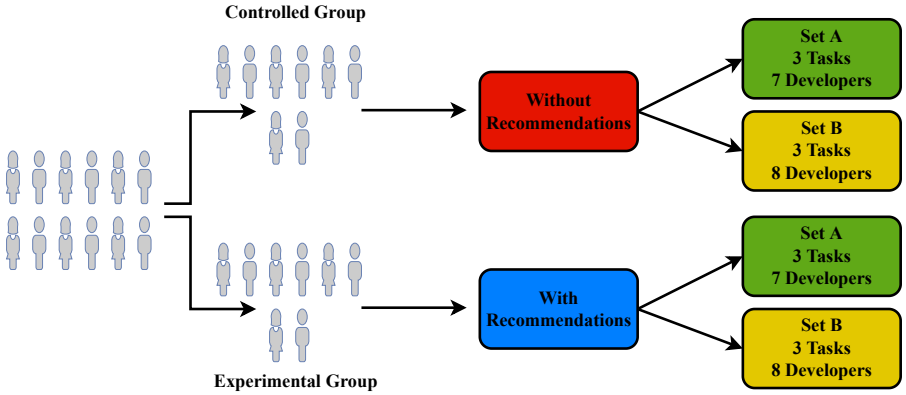


Fig. 8: Divisions of the Developers into Groups and Sections.

give a short presentation about the experiment and allow the developers to ask any questions that might arise during the experiment. However we remained on mute through the entire time of the experiment to avoid any distraction and unmuted only for answering questions.

In the short presentation, we explained the concept of the study, the purpose of the experiment, gave instructions on how to complete the experiment, and informed the developers that during the experiment they were only permitted to ask clarifying questions about the tasks. Further, the task description document explained the bug in each task, listed steps on how to replicate the issue, and gave instructions to access remotely the laboratory desktop where the subject system was installed. For developers in the second group of the experiment, their task description documents recommended the consensus tasks contexts along with each task.

Once a developer was remotely connected, we gave each of them up to 10 minutes to explore the system, get familiar with Eclipse and the system structure. In addition, we provided them with a practice task to familiarize them with the nature of the evaluation tasks. Before the start of the experiment, we asked the developers to perform the tasks in the same sequential order, and try their best locating relevant file(s)-to-edit to fix the bug. We did not ask them to make any code modification.

According to the task complexity assessment that was performed by two Ph.D. students, each task required a maximum of 30 minutes to complete. Therefore, we allocated 30 minutes of time for each task in the first group of developers, while we assigned 20 minutes for each tasks for those developers in the second group. We allowed developers to ask for five extra minutes if needed, and told them not to be concerned if they did not accomplish a task and move to the following one.

When developers completed locating file(s)-to-edit for each evaluation task, we requested them to fill their answers in an answer sheet. We then stopped

the video recording, and disconnected the remote access. Afterwards, we interviewed the developers and sent them a post-experiment questionnaire to gain their insight about the approach and the experiment in general, which will be discussed in the following section.

Measures. To study whether the CTCR affects developers' productivity when performing change tasks, we evaluated the success level of each developer by measuring time and completion. Time is meant to capture the total time each developer took to complete each evaluation change task, while completion confirms whether the task was completed successfully or not. To capture time, one of the authors collected the total time spent on each task by watching the video recordings of all developers. Task completion was inspected by the same author reading through developers' answer sheets, which were used to identify the file(s)-to-edit for the successful completion of the tasks. Blank answer sheets indicated that the developer could not define the set of file(s)-to-edit and therefore the task was not completed. Finally, we compared the results between the two groups for the same evaluation task.

In addition to measuring the ability to complete the evaluation tasks, we investigated the effect of the recommendations on developers' behaviour and navigation paths. We carefully watched and analyzed the video recordings of the 30 developers to study their navigation patterns and compared the patterns between the two groups. During the observation, we paused the videos whenever necessary to take notes and followed the mouse pointer's route on the screen to determine with what files the developers interacted. In particular, we observed the navigation steps each developer took to get a good understanding of their general navigation behaviour.

We assembled all the patterns of behaviour that were observed and summarized the most interesting observations from each group separately. The analysis help us define the kind of actions developers do in completing the assigned tasks and whether providing a set of recommendations helps improve their navigation behaviour and limit the number of consulted irrelevant system elements.

After the experimental group completed the experiment, we interviewed the developers to get their opinion about the experiment and CTCR approach in general. We also sent them an online post-experiment questionnaire with a series of exploratory questions. The questionnaire helped assess the importance of the recommendations, get developers' opinion about the perceived improvement in their performance, and gather any feedback that could help improve our approach.

The questionnaire consisted of nine questions: from assessing the difficulty of dealing with unfamiliar systems, to difficulty locating related files using the help from the approach, to the relevance of the recommended files, and to whether or not the approach helped improve their performance completing the evaluation tasks. Six of the nine questions were rating scale questions, two were yes/no questions, while the remaining two were open-ended questions. The questionnaire is presented in Table 8.

Questions	Type of Answer
Q1: When working on unfamiliar software systems, do you have difficulty knowing where to start?	(1-5)
Q2: Was the given time enough to perform each task?	(1-5)
Q3: How difficult was locating files related to the tasks at hand using the list of recommended files?	(1-5)
Q4: Having to deal with unfamiliar software system, do you think recommending files related to the tasks at hand eased program comprehension?	(1-5)
Q5: How would you rate the overall relevance of the recommended files to the actual files that needed code change to complete the tasks?	(1-5)
Q6: How would you rate the impact of the recommendation approach on the time needed to locate files/ source code? Do you think that saved you some time?	(1-5)
Q7: Do you think you could rely on the recommended files to help you complete the tasks?	Yes/No
Q8: Is there anything you dislike about the proposed recommendation approach?	Open-Ended
Q9: Is there anything would you like to suggest to improve the recommendation approach?	Open-Ended
1 = strongly agree; 2 = agree; 3 = neutral; 4 = disagree; 5 = strongly disagree	

Table 8: Post-Experiment Questionnaire.

5.3 *RQ3*: How does CTCR compare to MI (Mining Programmer Interaction Histories) Lee et al. (2015) in recommending relevant file(s)-to-edit for specific change tasks?

We extensively searched for existing file-level recommendation tools. Results of our search identified the following tools: NavTracks Singer et al. (2005), MI Lee et al. (2015), Mining Change History Ying et al. (2004), and NaCIN Majid and Robillard (2005). After carefully inspecting the four tools, we decided to base our comparison on MI because it is conceptually closest to CTCR. MI is a state-of-art approach that mines developers' interaction traces (edits and views), generates association rules using a provided context, and recommends file(s)-to-edit. A context is then a query formed from the current developer's interaction with a given task at the time of recommendation Lee et al. (2015).

Our hypothesis was that CTCR recommends more relevant file(s)-to-edit than the state-of-the-art because it forms its recommendations based on the consensus of the same or similar change tasks' interactions.

Context in MI is a core component that triggers recommendation. Given a developer's events from a change task, multiple contexts are formed from the last events (edit and selection types). MI defines a v-e sized sliding window that holds a set number of selection (v) and edit (e) events. The sized sliding window moves from the first to the last event. As the sliding window gets updated, the context is updated with the last events. MI introduces several methods for creating a context:

– *MI-EA* merges selection and edit events using *AND* operation and generates recommendation at edit events.

- *MI-EO* merges selection and edit events using *OR* operation and generates recommendation at edit events.
- *MI-VA* merges selection and edit events using *AND* operation and generates recommendation at selection and edit events.
- *MI-VO* merges selection and edit events using *OR* operation and generates recommendation at selection and edit events.
- *MI-VOA* a combination of *MI-VA* and *MI-VO*, merges selection and edit events using both *AND* and *OR* operations, and generates recommendation at selection and edit events.

Procedure. We simulated file(s)-to-edit recommendations in MI using the interactions generated form the three change tasks in Section 4. According to Lee et al. (2015), applying different context creating methods, generates different recommendation results. Further, their results showed that methods with the AND operation yield higher recommendation accuracy. Therefore, we used methods with the AND operation: MI-EA, MI-VA, and MI-VOA.

Regarding the size of the sliding window, the authors of MI spread the *v-e* sized sliding window between 1 and 10. Considering that the size of our dataset was smaller than the dataset used in the MI evaluation experiment, applying MI to our dataset with a sliding window size greater than 4 did not generate any recommendations. Hence, we varied the *v-e* sliding window size from 1 to 4. In general, developers viewed 79 files for every one they edited Lee et al. (2015). Therefore, we set the number of *(v)* to 4 and the number of *(e)* to be less than 4. For each *v-e* value, we ran the simulation repeatedly over all the participants' interaction traces.

Measures. We evaluated MI recommendation results against CTCR recommendations and assessed which approach recommended more relevant file(s)-to-edit using precision, recall, and F-measure. We used the sets of ground truths created in Section 4.8 as a baseline for the comparison. The formula of the three measures was presented in Section 5.1.

6 Results and Discussions

We now present the results from the evaluations, analyse observations, and discuss their implications.

6.1 *RQ1*: To what degree does CTCR recommend relevant files to given change tasks?

Table 9 presents the precision, recall, and F-measure values that result from comparing the accuracy of the results of CTCR to the ground truths from the three change tasks.

	Precision	Recall	F-measure
Task 1	1	1	1
Task 2	.17	.33	.15
Task 3	1	.5	.66

Table 9: Precision, Recall and F-measure Values of the Three Change Tasks.

The precision and recall values of CTCR results for Change Task 1 and 3 are encouraging. Results from both tasks generate a precision value of 1, meaning 100% of the time CTCR produces recommendations that are accurate and specifically correspond to the files needed to complete these change tasks by the participants. Furthermore, the recall rates show that 100% and 50% of the recommended files were in the ground truth data. High precision and recall rates lead to high F-measure rates, which result in accurate CTCR recommended file(s)-to-edit. The precision, recall, and F-measure values of the CTCR results for Change Task 2 are least satisfactory, however still 33% of the recommended files are relevant and overlap with the files in the ground truth (with a recall of 33%).

To investigate the reasons behind the lower values for Change Task 2, which stem from false negatives, we examine thoroughly the set of files in the ground truth and compare them to the result of CTCR. The ground truth includes three files (`DocSection.java`, `SchemaFormOutlinePage.java`, and `DocumentSection.java`), while CTCR recommended seven files. The bug in Change Task 2 required a code change in only a single file (`DocSection.java`), and the other two classes are irrelevant.

We investigate the files further to determine if there are methods that are called among the three files all together, and we identify no shared methods. Therefore, we assume that the ticket owner navigated these unrelated files for other purposes while fixing the bug.

Considering that these two files are irrelevant to the change task, none of the participants made any kind of interactions on them while performing the task. Consequently, CTCR did not recommend these files and instead recommended other files based on the navigation of all the participants who completed the task. Thus, we argue that CTCR provides more relevant files than available in the ground truth, files that are necessary for participants to understand the change tasks and perform the correct changes. To assess the relevancy of recommended files to change tasks, we plan in future work to perform an experiment in which we ask participants to rate the relevancy of recommended files.

Results from our approach statistically answered the first research question that considers the quality performance of CTCR. Overall, CTCR results achieved high average precision (72%), recall (61%), and F-measure (60%).

RQ1: CTCR achieves high precision, recall, and F-measure and recommend accurate and relevant file(s)-to-edit.

6.2 *RQ2*: Given a change task, can CTCR help guide newcomers' navigation paths to relevant file(s)-to-edit and increase their productivity?

Developers Success Level. Figures 9a and 9b compare the average time (in minutes) spent on each evaluation task from set A and B by the control group (1) (developers who completed the task without CTCR recommendations) to the experimental group (2) (developers given CTCR recommendations), while Figures 9c and 9d present the numbers of developers from each group who completed each evaluation task in the two sets.

(a) Average Spent Time on Tasks in Set A

(b) Average Spent Time on Tasks in Set B

(c) # of Completed Tasks in Set A

(d) # of Completed Tasks in Set B

Fig. 9: Average Spent Time and # of Completed Tasks in Both Sets by the Two Groups.

Looking at the average completion time of each evaluation task, we observe that the control group spent on average about twice as much time as needed by the experimental group. In task set A, developers with recommended file(s)-to-edit needed 46%, 68%, and 47% less time to perform each task, respectively, in comparison to developers in the control group. Similarly, those developers in the experimental group could complete the tasks in set B in 52%, 47%, and 30% less time than those in the control group.

Developers from both groups spent higher average time performing Evolution Task 1 in the two sets. Notwithstanding that the developers were given a practice task before the start of the experiment to familiarize themselves with

the experiment, navigation through random classes was a significant component of Evolution Task 1.

For Evolution Task 2, we observe that developers from the experimental group needed much lower average time in both sets because developers learned through the practice task and Evolution Task 1. Evolution Task 3 required higher average time than Evolution Task 2 in both sets because, while Evolution Tasks 1 and 2 are related to bugs and require developers to identify a single file-to-edit, Evolution Task 3 is a feature request that involves identifying three files-to-edit, hence demands more navigation time.

Regarding completion factor, Figures 9c and 9d show that, in both task sets, results from the experimental group present higher task completion rate than the first group. Examining completion rate of tasks in Set A reveals that among the seven developers in the experimental group, the completion rate was 100% for the three evolution tasks. Conservatively, only one out of the seven developers in the control group completed Evolution Task 1 and four completed Evolution Tasks 2 and 3. For Set B, the overall completion rate of the experimental group is higher in comparison to the control group. Particularly, files related to the completion of Evolution Tasks 1 and 2 were identified successfully by all the eight developers in the experimental group, while six of them identified the files related to Evolution Task 3.

None of the developers in the control group had any success with Evolution Tasks 1 and 3, while five of them completed Evolution Task 2. We expected this very low completion rate considering that developers had no prior system related knowledge. We noticed that only two developers from the experimental group could not identify the files related to Evolution Task 3. We hypothesize the lack of completion by the two developers was due to the type of Evolution Task 3 and navigation effort it required. To get more insight about the reason of not completing the task, we discuss it further with the two developers in the post-experiment interview later in this section in "User-experience and Feedback".

Although CTCR helped diminish the average time and navigation effort, there is still a disparity between the actual total time spent by each developer in the experimental group. Developer's efficiency and experience impact the amount of time and effort when dealing with an unfamiliar software system. Developers without CTCR however spent a substantial amount of time understanding and exploring unrelated files to find relevant files to their current tasks, which affected their productivity negatively.

> CTCR recommendations increase developers' productivity by recommending relevant files and reducing navigation effort and time.

CTCR Effect on Developers Navigational Behaviour. Developers typically explore systems using a variety of approaches. However, two distinct navigation

behaviours were primarily used by developers in the control group. In the first observed behaviour, after following the steps of replicating the bug, some developers made use of the built-in Eclipse search dialog to search for keywords related to the task at hand, based on fields names that appear on the window, for example. Then, they spent a considerable amount of time continually navigating through the returned results, visiting each result, switching between files, and glancing over the source code trying to find any relevant methods.

In the second observed behaviour, developers did not follow any search strategies. They explored the system via unstructured exploration that included scrolling up/down the package explorer. They skimmed through the names of files to judge their relevance. If they believed a name seemed relevant, they accessed the file and scrolled over the file elements to identify any relevant source code.

The experimental group, on the contrary, followed a same navigation approach. All developers started performing the tasks by navigating to every recommended file and reading through the source code, before defining file(s) related to the bug. Some developers went even one step further and specified the source code that should be edited to fix the bug or implement the feature request.

To further highlight the impact of the different navigation behaviours, we analyzed the observations and identified that the two navigation behaviours by the control group were inefficient. The main goal by all developers was to define a set of entry points, *i.e.,* a basic set of files with which they might begin their investigation. In the first navigation behaviour, we noticed that developers wrongly chose search keywords that led the search engine to return irrelevant results. Even when they chose more search keywords, the search returned a large number of files due to the size of the project. Consequently, developers had to spend time sifting through irrelevant search results.

In the second behaviour, developers began a broad search to filter out irrelevant files by arbitrarily browsing through the files in the package explorer. This behaviour resulted in a frequent switch between multiple files. Due to the size of the search space and unfamiliarity of the system, most developers in the control group found themselves engaged in an increasing effort and time exploring significantly unrelated files, rounds of searches that yielded no relevant results, and hence inability to locate file(s)-to-edit.

With the experimental group, our observation reveals that developers depended heavily on using the search dialog function to search for keywords related to fixing the bug at hand in the recommended files only to determine the needed file(s) to complete the tasks. Considering that the CTCR recommends relevant files and the searched keywords could possibly appear in most of the files, developers used their own judgment and programming experience and spent limited effort comprehending methods that they believed to be relevant to the tasks. When developers were required to locate file(s)-to-edit that were not part of the set of recommendations, in the case of Evolution Task 3, they followed the relevant methods' cross-reference to locate these files. From these observations, we found that the experimental group could apply a more struc-

tured navigation, guide their attention and effort to understanding relevant system elements, avoid investigating irrelevant files, and efficiently determine more related files.

> CTCR recommendations can guide new developers to exhibit a structured navigation behaviour that can increase their productivity.

User-experience and Feedback. We collected developers' answers to the interview/questionnaire questions, compared, and summarised them. Questionnaire results are reported in Table 10.

	1	**2**	**3**	**4**	**5**
Q1	Not at all 1 - 6.7%	2 - 13.3%	2 - 13.3%	4 - 26.7%	Very Difficult 6 - 40%
Q2	Not Enough 0 - 0%	1 - 6.7%	2 - 13.3%	5 - 33.3%	Very Enough 7 - 46.7%
Q3	Not at all 3 - 20%	6 - 40%	2 - 13.3%	4 - 26.7%	Very Difficult 0 - 0%
Q4	Not at all 0 - 0%	0 - 0%	0 - 0%	4 - 26.7%	Absolutely 11 - 73.3%
Q5	Not Related 0 - 0%	0 - 0%	3 - 20%	3 - 20%	Very Related 9 - 60%
Q6	No Time Saved 0 - 0%	0 - 0%	0 - 0%	4 - 26.7%	Saved Time 11 - 73.3%
Q7	Yes 14 - 93.3%	No 1 - 6.7%			

Table 10: Post-Experiment Questionnaire Answers.

When asked about the difficulties in finding an entry point, 67% of the developers stated that it is very difficult while 33% found it fairly easy to locate an entry point.

Most developers (12) considered that the time given to conduct each task was appropriate. We asked the developers to rate the difficulty of completing the given tasks using CTCR recommendations. Several developers (11) strongly agreed that completing the tasks using CTCR recommendations was not at all difficult, while the remaining four seemed to have difficulty. These answers confirm that a few developers could not successfully complete the tasks.

All developers strongly agreed that CTCR helped them understand the parts of the system that are related to the given tasks. Beside system comprehension, developers appeared to be extremely satisfied when asked about the relevancy of the CTCR recommended files to the given tasks: 80% stated the recommendations were very relevant. All developers expressed a positive impression of how CTCR helped them spend less time navigating through system elements because CTCR provided them a few entry points to start with.

93% of the developers confirmed that they could completely rely on CTCR recommendations to help them perform similar change tasks.

During the interview, we asked each developer to share their thoughts on the experiment in general, any obstacles they encountered, how CTCR promoted their productivity, and any general feedback. One developer stated that, when dealing with a change task, he needs to employ a set of steps, such as comprehending the structure of the system, identifying entry points, locating related source code, applying the change, and testing. Providing him with recommendations from other similar change tasks helped speed the process of locating the part of the system that is related to his task and exploring other files that he would not have considered. Similarly to this developer, other developers said that they treated the set of recommendations as entry points to the system which saved them from randomly hunting through the package explorer.

However, the two developers who did not complete one of their assigned evaluation tasks still found completing the tasks and navigating through the files challenging even with having CTCR recommendations. They reported that even though CTCR limited their search space and directed their navigation, being a newcomer to the system made it daunting to skim through the files and identify the ones to edit. Specifically, dealing with the system became overwhelming due to the growing number of files in the package explorer. We asked these developers if they believe that is potentially due to the lack of practical Java programming experience. Even though these developers indicated in the pre-experiment survey that they have some years of Java programming experience, during the interview they confirmed that the experience is more of educational experience rather than hands-on experience and they are more Python developers.

Nearly all developers were satisfied with the CTCR recommendations and the navigation guidance that they provide.

RQ2: CTCR can help minimize developers' time and effort completing change tasks and guide their navigation into a more structured navigation behaviour.

6.3 *RQ3*: How does CTCR compare to MI (Mining Programmer Interaction Histories) Lee et al. (2015) in recommending relevant file(s)-to-edit for specific change tasks?

We now assess how our approach compares to the state-of-the-art approach. We report and compare results of applying MI to our dataset using *MI-EA*, *MI-VA*, and *MI-VOA* context formation methods with different values of *v-e* sliding window. To answer the research question statistically, we compute precision, recall, and F-measures for MI over the set of ground truths. The values from the results of the three methods of MI are then analyzed. Based on the results, we measure the effectiveness of CTCR by comparing the statistical values of CTCR to MI.

We observe that simulated file(s)-to-edit recommendations on MI vary using the three recommended context formation methods and various v and e values. Running MI on our dataset with *v-e* values greater than 4 returns no recommendations. The three selected context formation methods make recommendations at either edit only events or select and edit events together. Given the average complexity of our change tasks and the moderate effort required by the participants to complete the tasks, the number of generated edit events from completing each task are not significant enough for the methods to provide recommendations when the sliding window is greater than 4. Therefore, we ran MI on interaction traces from Change Task 1 and 2 using three sets of *v-e* sliding window (4-2, 4-1, and 3-1). In contrast, MI on interaction traces from Change Task 3 produced results with only one set of *v-e* sliding window (2-1).

The *MI-EA* and *MI-VA* methods produced the exact recommendations from the three change tasks even with varied *v-e* sliding windows. A combination of two factors led to this result: both methods use the AND operation to merge events, while one recommends only at edit events and the other at selection and edit, however the difference in trigger point does not make an impact when the total number of edit events is relatively small.

Table 11 presents the recommendation results of applying *MI-EA* to interaction traces of Change Task 1 under the three sets of sliding window along with CTCR recommendation of the same task. Unsurprisingly, as the value of the sliding window decreases, MI makes recommendations on more edit events and hence recommends a higher number of files-to-edit. We notice that when (v=4, e=2), *MI-EA* recommendations do not intersect with CTCR recommendations nor ground truths, while the opposite is true in the case of (v=4, e=1) and (v=3, e=1). This observation suggests that the quality of the recommendations by MI depends greatly on the value of the sliding window and there is no a specific set of values that will guarantee high quality results as the number of events is always changing. CTCR provides quality results without the need of defining a sliding window.

Table 12 compares the recommendation results of *MI-EA* with *MI-VOA* methods under the same set of sliding windows. *MI-VOA* with different sliding-window values produced nearly the same recommendations of files-to-edit. We

***MI-EA* (v=4, e=2)**	***MI-EA* (v=4, e=1)**	***MI-EA* (v=3, e=1)**	**CTC**
FeatureSection.java PluginVersionPart.java PDELabelProvider.java	Pderesources.properties PDEUIMessages.java PluginVersionPart.java FeatureSection.java PDELabelProvider.java VersionDialog.java	Pderesources.properties PDEUIMessages.java FeatureSection.java VersionDialog.java	Pderesources.properties PDEUIMessages.java VersionDialog.java

Table 11: Simulation Results of *MI-EA* and CTCR Recommendations from Change Task 1.

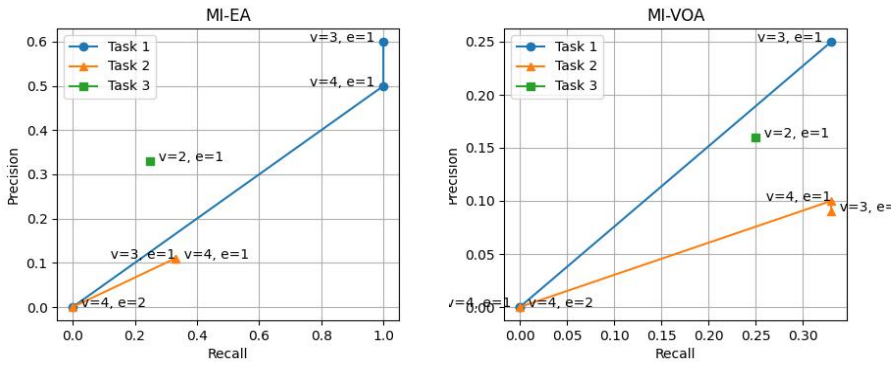
also found that none of the recommended files intersect with CTCR recommendations, except with (v=3, e=1). In addition to the previous observations, the use of the OR operation cannot provide high quality results with our dataset due to the small number of edit events, which we will assess further statistically. The rest of the results are available online[7].

***MI-EA* (v=4, e=2)**	***MI-VOA* (v=4, e=2)**
FeatureSection.java PluginVersionPart.java PDELabelProvider.java	FeatureSection.java PluginVersionPart.java PDELabelProvider.java Utilities.java
***MI-EA* (v=4, e=1)**	***MI-VOA* (v=4, e=1)**
Pderesources.properties PDEUIMessages.java PluginVersionPart.java FeatureSection.java PDELabelProvider.java VersionDialog.java	FeatureSection.java PluginVersionPart.java PDELabelProvider.java Utilities.java
***MI-EA* (v=3, e=1)**	***MI-VOA* (v=3, e=1)**
Pderesources.properties PDEUIMessages.java PluginVersionPart.java FeatureSection.java VersionDialog.java	FeatureSection.java PluginVersionPart.java PDELabelProvider.java Utilities.java Pderesources.properties

Table 12: Results of MI-EA Against MI-VOA from Change Task 1.

Given that *MI-EA* provided better quality recommendations over *MI-VOA* in the case of our dataset, we further investigate the accuracy statistically by comparing precision, recall, and F-measures values of the results of both methods. We computed the measure values using the set of results from both methods and the set of ground truth. Figure 10 presents the resulting precision and recall curves of the recommendation results from *MI-EA* and *MI-VOA* under different (v-e) sliding windows. The figure shows recommendations from Change Task 1 using *MI-EA* consistently achieved higher precision and recall than using *MI-VOA* when (v=4, e=1) and (v=3, e=1). Analogously, *MI-EA* showed higher precision and exact recall values under the same sliding windows for Change Task 2. Comparing results of *MI-EA* to *MI-VOA* from Change Task 3, the former also showed higher precision and same recall values. In terms of F-measure, we took the average values from the three sets of (v-e) slid-

ing windows for each change task. *MI-EA* performed an average F-measure of 0.47, 0.1, and 0.13 for each task respectively. Whereas *MI-VOA* performed lower F-measure from Change Task 1 and 2 (0.04 and 0.08) and a slightly higher value from Change Task 3 (.19). All in all, the average accuracy of *MI-EA* recommendations across all the three (v-e) sliding windows is consistently higher than the recommendations resulting from *MI-VOA* method. The lower accuracy is the result of the use of the combination of AND and OR operations in the process of generating context which triggers recommendations when both operations are met. *MI-VOA* was proven to provide high accuracy recommendations when the size of the dataset is large enough, however same performance can not be accomplished when the size of the dataset and number of edit events are relatively small.



(a) MI-EA Precision and Recall

(b) MI-VOA Precision and Recall

Fig. 10: Precision and Recall of Recommendations from MI-EA and MI-VOA.

To evaluate CTCR recommendations accuracy and relevancy, we use MI results as a comparison baseline. Comparing precision, recall, and F-measure values of the recommendations of *MI-EA* at each (v-e) sliding window for each change task, we observe that the values are very similar. Thus we take the average value of the three (v-e) sets of each measure to compare with CTCR recommendation results.

Figure 11 presents the precision and recall curves of the recommendation results of MI against CTCR. As shown, examining the results of all the three change tasks, CTCR recommends files-to-edit with precision values of 1, 0.17, and 1 and recall values of 1, 0.33, and 0.5, respectively. On average, MI yielded lower accuracy results with 0.36, 0.07, and 0.33 precision and 0.66, 0.22, and 0.25 recall, respectively. Consequently, CTCR significantly outperforms MI in terms of F-measure values. As shown in Figure 12, our approach shows F-measures values of 1, 0.15, and 0.66 for each change task respectively. Whereas MI performed average accuracy at 0.47, 0.1, and 0.13 of F-measure values.

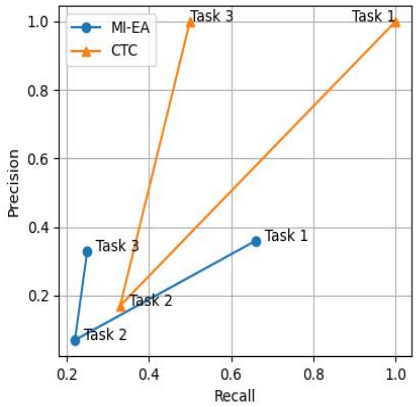


Fig. 11: MI-EA and CTCR Precision and Recall Curves

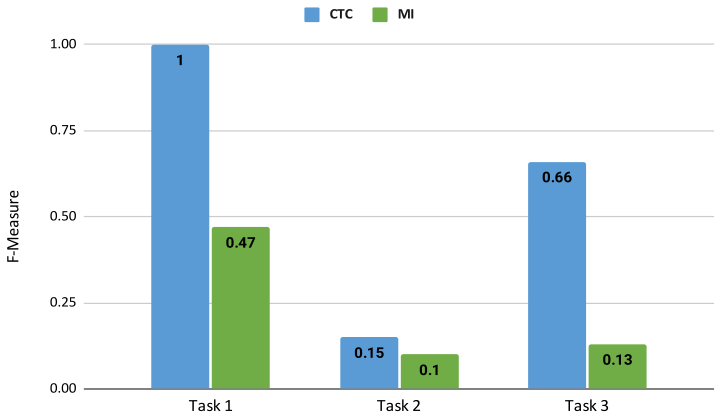


Fig. 12: CTCR and MI F-Measure Values

To better understand how CTCR provided higher recommendation accuracy than MI, we analyze how the approaches work along with the recommendation results. To illustrate, in Table 11, CTCR recommends `pderesources.properties`, `PDEUIMessages.java` and `VersionDialog.java` out of the interaction traces of Change Task 1, which precisely match the suggested files in the ground truth data.MI recommended three other files that are irrelevant to the context of this task. CTCR can recommend more relevant file(s)-to-edit with less noise than MI. The technique used requires MI to specify the size of the context that will trigger the recommendation. As the v-e value changes and the sliding window moves, the context gets constantly updated, which affects what files to consider and eventually yields irrelevant files. With CTCR,

there is no need to specify a particular set of context to trigger recommendation as the approach treats the whole set of interaction traces as one context and extracts the most relevant files based on the idea of the consensus.

Many recommendation tools Singer et al. (2005); Ying et al. (2004), including MI, require developers to start interacting with system elements before they start recommending file(s)-to-edit. Some of these tools base their recommendations on association rules. When a set of files are viewed and edited together, the method associates them together and recommends them if a future developer interacts with at least one of the files in the set. Yet, not all navigated together files are necessarily relevant to a given task. Thus, tools based on association rules recommend files that are not particularly related to the completion of the task at hand. These tools that require developers' interactions prior to recommendation are ideal when the developers are to some degree familiar with the software system and can navigate to a few entry points.

In comparison, CTCR does not build recommendations based on a small set of interactions that are associated together as one context. In essence, it combines and treats all developers' interaction histories as one task context, finds a set of consensus files among all the files, and recommends them to help completing other similar tasks. Ability to treat all interaction traces as one context for recommendations explain why CTCR suggests more relevant files than association-rule based approaches. To our knowledge, our approach is the first recommendation approach that recommends file(s)-to-edit based on the consensus algorithm and does not require developers to provide navigation hints prior to recommendation. CTCR guides the navigation of newcomers with no prior knowledge of the current system. Hence, it helps newcomers understanding and completing the tasks using the recommended files, and not on relying on random navigation to get the tool to start making recommendations.

RQ3: the comparison with MI showed that CTCR yields higher accuracy and relevance recommendations than MI.

7 Threats To Validity

Change Tasks. We based our choice of change tasks, their complexity, and required time to complete on our judgment and experience, which could be biased. We mitigated the threat of misjudgement by hiring an external experienced evaluator. We considered moderate complexity and tasks that require less than 45 minutes to complete. Complex tasks would take participants more time to complete and might lead to interruptions or participants quitting the study.

Time. The chosen change tasks require less than an hour to complete. Thus, these tasks might not reflect the full spectrum of tasks performed by developers. To limit the effect of this choice, we used three different change tasks from a large open source system in the design of the ITs collection (in Section 4.2), completed by participants with various educational and industrial backgrounds, and using a common IDE and programming language, Eclipse and Java.

Mylyn Noise. This threat is related to the tool used to collect participants' events, Mylyn Eclipse plugin. Mylyn introduces some noise, such as time-related noise, edit-related noise, duplicated events, or missing events. We implemented a pre-processing approach to reduce the impact of noise on the results of our evaluations. Despite the presence of remaining noise, our approach could deliver high quality results.

Generalizability. External threats pertain to the possibility to generalize our results. We evaluated CTCR on a limited number of developers' interaction traces and on one software system written in Java. An evaluation study on a much larger dataset and–or systems written in different programming languages could yield different results. In the future, we plan to assess the generalizability of CTCR on open and closed software systems written in different languages and on more interaction traces.

Remote Experiment. Due to the COVID-19 pandemic, we had to change the observational comparative experiment (in Section 5.2) from a laboratory experiment to a remote experiment. We could not control interruptions, which could impair developers' navigation behaviour and productivity. We could only ask developers to perform the experiment in a quiet environment, record their screens, and audio-call them using Zoom conferencing software.

Reliability. We make all data used in this study available online in a public repository for replication purposes[7]. To increase the reliability of our results, we employed multiple measures: precision, recall, and F-measure for quantitative evaluation; an observational comparative experiment with video observation analysis, post-experiment interviews, and questionnaire for the qualitative evaluation; and, a comparison with an existing approach.

Measures. Considering that our approach produces a set of consensus file(s)-to-edit with which developers must interact to complete a particular task, precision and recall measures could underestimate the accuracy of our results. Indeed, we computed precision and recall based on ground truths that contain files with which Bugzilla ticket owners interacted while fixing the bug. Some of these files may be actually unrelated to the ticket. However, we kept these files in the ground truths to be conservative and not risk tainting the ground truths with our own biases.

Observation Bias. We based the qualitative findings of developers' behaviour in the observational comparative experiment (in Section 5.2) on observation and interpretation of video recordings of developers performing some evaluation change tasks. We could have been biased and provided wrong interpretations. To ensure correct findings, one author watched the videos and noted the different behaviours, followed by another author who cross-validated the findings. The findings from the two authors were very identical.

8 Related Work

Previous research related to our work can be divided into four areas: research using developers' interaction traces to support software engineering activities; studies using different sources of data for building recommendation systems; building recommendation systems using interaction traces; and, research studying developers' navigation behaviour.

8.1 Use of Interaction Traces for Software Engineering Activities

Researchers studied and analyzed developers' ITs to ease software engineering daily activities. Soh et al. (2013b) mined developers' ITs to understand how their exploration strategies when performing maintenance tasks can affect time and effort spent. They classified developers' exploration strategies into referenced exploration (*i.e.,* revisitation of entities) and unreferenced exploration (*i.e.,* equal frequency of visiting entities).

Similarly, Ying and Robillard (2011) analyzed developers' interaction histories and characterized their editing styles into edit-first, edit-last, and edit-throughout. Their observation revealed that enhancement tasks are likely to be associated with edit-last or edit-throughout. Sanchez et al. (2015) studied ITs to investigate the correlation between work fragmentation (*i.e.,* interruption) and developers' productivity. Results showed that interruption can lead to a lower productivity level.

To reveal latent facts about the development process, Zou et al. (2007) investigated interaction coupling in interaction histories and found that restructuring is more costly than other maintenance activities.

Lastly, Parnin and Rugaber (2009) used interaction histories to discuss coping mechanisms that developers can follow to resume their work after having been interrupted.

All these works focused on the use of interaction traces to help developers enhance the quality of software activities. Although our work shares the same purpose of enhancing software activities quality, we focus on the use of ITs for building a recommendation system.

8.2 Recommendation Systems

To find system elements relevant to a task, developers have used a variety of tools, ranging from grep to program databases Teitelman and Masinter (1981). Recent research studies used different sources of data to help build recommendation systems to improve program comprehension and navigation.

HeatMaps Rothlisberger et al. (2009) computes a Degree-of-Interest (DOI) value based on navigation history, change logs, and execution data. They present artifacts in the IDE to colors ranging from red ("hot") to blue ("cold") according to the DOI value.

Hipikat Cubranic and Murphy (2003) recommends relevant artifacts based on a group memory that is formed using source code versions, bugs, electronic communication, and web documents. By applying a clustering technique to software revision history,

Robillard and Dagenais (2010) could retrieve source code relevant to tasks from clusters of change sets that contain common system elements based on defined filtering heuristics.

Both Zimmermann et al. (2004) and Ying et al. (2004) applied association rules to CVS data to recommend newcomers with system entities to edit.

While these approaches use system related data to generate recommendations, we focus on using developers' interaction histories data as a base for recommending file(s)-to-edit.

8.3 Interaction Traces Based Recommendation Systems

Several works used developers' interaction traces to recommend relevant system elements. Team Track DeLine et al. (2005) helps new developers better understand and navigate code by providing them with pieces of code to visit. It mines consecutive visits between methods in developers' interaction histories to find the next method to visit.

Likewise, by mining developers' interaction histories, NavTracks Singer et al. (2005) forms a relationship between system files as a developer browses them, and then recommends files that are relevant to the currently browsed ones.

Meanwhile, Kersten and Murphy (2006) used interaction histories to build task contexts. Their approach recommends system elements according to the frequency and recency of elements in the context.

NavClus Lee and Kang (2013) clusters sequences of navigation from interaction histories. It uses association rules to recommend system elements that are relevant to the developer's current navigation.

Robbes and Lanza (2010) used change history data to propose a code completion tool that helps developers complete their change tasks.

Switch! Sahm and Maalej (2010) was proposed to recommend system artifacts. It bases its recommendation on association between the context of the current task and interaction histories.

Although these studies use developers' interaction traces to build recommendations, they assume that developers come with some knowledge of the systems and require them to start interacting with the systems before providing any recommendation. In addition, recommendations are based on association rules, which might lead to recommending unrelated elements if the developers interacted with the wrong elements. In contrast, our approach builds recommendations without any prior interaction from the developers. In addition, when we compared our approach to MI, CTCR can recommend file(s)-to-edit that are more relevant to the given change tasks than what MI recommended.

8.4 Study of Developers' Behaviour

There are many works on developers' navigation behaviours, factors that impact their behaviours, and strategies to understand source code.

Ko et al. (2006) conducted an exploratory study to understand how developers decide what is relevant information to their tasks and how they keep track of this information. Their study involved 10 developers performing maintenance tasks on an unfamiliar software system. They found that developers spend a significant time searching for relevant information which often ends in failed searches.

Robillard et al. (2004) performed a study of five developers performing a change task to investigate factors that contribute to effective navigation behaviour.

To determine what specific questions developers ask when performing programming tasks, Sillito et al. (2006) conducted a laboratory study with 25 developers. They identified 44 types of questions developers ask during change tasks.

Similarly, to understand how developers perform feature location tasks, Wang et al. (2011) invited 38 students to perform six feature location tasks on unfamiliar systems. The study results enabled them to build a conceptual framework that consists of a collection of phases, patterns and actions.

Meanwhile, Starke et al. (2009) focused their exploratory study on investigating how developers search through source code and skim through results. They observed that developers do not inspect results closely if they believe that the results are irrelevant and prefer to perform another search.

While our qualitative results support some of the observations reported in these works, our observational experiment was based on real change tasks and focused on observing the behaviours of the experimental group of developers against the controlled group.

9 Conclusion

In large, customised software systems, the successful completion of change tasks requires developers to investigate elements that are scattered across the

systems. Finding and understanding the subset of elements part of a change task is complex and requires developers' time and effort.

We proposed an approach called consensus task context recommender, CTCR, which builds recommendations using a consensus algorithm. It uses developers' previous interaction traces collected while completing change tasks on different clients' system instances as tasks contexts and use them as input to a consensus algorithm to generate recommendations. CTCR can recommend relevant file(s)-to-edit to help newcomers complete similar change tasks with minimal effort and time.

We evaluated our approach using a series of evaluations: quantitative, qualitative, and comparison. In the quantitative evaluation, we measured quantitatively the accuracy of the recommendations against ground truths. Measures showed that CTCR can recommend accurate and relevant file(s)-to-edit with average precision of 72%, recall of 61%, and F-measure of 60%.

In the qualitative evaluation, we carried out an observational comparative experiments to measure the extent to which recommendations could increase newcomers' productivity. Results demonstrated that CTCR could increase developers' productivity by helping them achieve a high task success rate in less time and follow a more structured navigation behaviour.

Lastly, we compared our approach to a state-of-the-art approach, MI Lee et al. (2015). Results showed that CTCR can achieve higher recommendation accuracy and relevancy than that of MI with average F-measure value of 60% and 20% respectively.

We concluded that CTCR can help newcomers by recommending relevant file(s)-to-edit and that consensus algorithms are efficient tools to compute such recommendations.

In the future, we plan (1) to investigate the possibility of optimizing the quality of Mylyn collected events to reduce noise and atomising the preprocessing step; (2) to confirm our results on a larger dataset, more complex change tasks, and extend the evaluation process to involve real software companies; (3) to enhance developers' navigation experience by developing an IDE plugin that can use stored data to automatically generate recommendations and highlight the recommended files in the package explorer without requiring developers to explicitly locate these files; (4) to search and develop a ranking algorithm that can rank the recommended files according to their relevance to the task at hand; and, (5) to apply our approach at the method level to investigate whether the approach could recommend method(s)-to-edit.

Declarations

Conflict of interest The authors declare that they have no known competing interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Ailon N, Charikar M, Newman A (2008) Aggregating inconsistent information: Ranking and clustering. J ACM 55(5), DOI 10.1145/1411509.1411513

Ali A, Meilă M (2012) Experiments with kemeny ranking: What works when? Mathematical Social Sciences 64(1):28–40, DOI 10.1016/j.mathsocsci.2011.08.008, computational Foundations of Social Choice

Amann S, Proksch S, Nadi S (2016) Feedbag: An interaction tracker for visual studio. In: 2016 IEEE 24th International Conference on Program Comprehension (ICPC), pp 1–3, DOI 10.1109/ICPC.2016.7503741

Avazpour I, Pitakrat T, Grunske L, Grundy J (2014) Dimensions and Metrics for Evaluating Recommendation Systems. In: Robillard MP, Maalej W, Walker RJ, Zimmermann T (eds) Recommendation Systems in Software Engineering, Springer Berlin Heidelberg, pp 245–273, DOI 10.1007/978-3-642-45135-5_10

Brancotte B, Yang B, Blin G, Cohen-Boulakia S, Denise A, Hamel S (2015) Rank aggregation with ties: Experiments and analysis. Proc VLDB Endow 8(11):1202–1213, DOI 10.14778/2809974.2809982

Cohen-Boulakia S, Denise A, Hamel S (2011) Using medians to generate consensus rankings for biological data. In: Proceedings of the 23rd International Conference on Scientific and Statistical Database Management, Springer-Verlag, Berlin, Heidelberg, SSDBM'11, p 73–90

Critchlow DE (1985) Metric methods for analyzing partially ranked data, vol 34. Springer Science & Business Media

Cubranic D, Murphy G (2003) Hipikat: recommending pertinent software development artifacts. In: 25th International Conference on Software Engineering, 2003. Proceedings., pp 408–418, DOI 10.1109/ICSE.2003.1201219

DeLine R, Czerwinski M, Robertson G (2005) Easing program comprehension by sharing navigation data. In: 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), pp 241–248, DOI 10.1109/VLHCC.2005.32

Erlebacher A (1977) Design and analysis of experiments contrasting the within-and between-subjects manipulation of the independent variable. Psychological Bulletin 84(2):212, DOI 10.1037/0033-2909.84.2.212

Fagin R, Kumar R, Mahdian M, Sivakumar D, Vee E (2004) Comparing and aggregating rankings with ties. In: Proceedings of the Twenty-Third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, Association for Computing Machinery, New York, NY, USA, PODS '04, p 47–58, DOI 10.1145/1055558.1055568

Fritz T, Shepherd DC, Kevic K, Snipes W, Bräunlich C (2014) Developers' code context models for change tasks. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, FSE 2014, p 7–18, DOI 10.1145/2635868.2635905

Kemeny JG (1959) Mathematics without numbers. Daedalus 88(4):577–591

Kersten M, Murphy GC (2006) Using task context to improve programmer productivity. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, SIGSOFT '06/FSE-14, p 1–11, DOI 10.1145/1181775.1181777

Ko AJ, Myers BA, Coblenz MJ, Aung HH (2006) An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. IEEE Trans Softw Eng 32(12):971–987, DOI 10.1109/TSE.2006.116

LaToza TD, Venolia G, DeLine R (2006) Maintaining mental models: A study of developer work habits. In: Proceedings of the 28th International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '06, p 492–501, DOI 10.1145/1134285.1134355

Lee S, Kang S (2011) Clustering and recommending collections of code relevant to tasks. In: 2011 27th IEEE International Conference on Software Maintenance (ICSM), pp 536–539, DOI 10.1109/ICSM.2011.6080826

Lee S, Kang S (2013) Clustering navigation sequences to create contexts for guiding code navigation. J Syst Softw 86(8):2154–2165, DOI 10.1016/j.jss.2013.03.103

Lee S, Kang S, Kim S, Staats M (2015) The impact of view histories on edit recommendations. IEEE Transactions on Software Engineering 41(3):314–330, DOI 10.1109/TSE.2014.2362138

Majid I, Robillard MP (2005) Nacin: an eclipse plug-in for program navigation-based concern inference. In: Proceedings of the 2005 OOPSLA workshop on Eclipse Technology eXchange, ETX 2005, San Diego, California, USA, October 16-17, 2005, ACM, pp 70–74, DOI 10.1145/1117696.1117711

Minelli R, Mocci A, Lanza M, Kobayashi T (2014) Quantifying program comprehension with interaction data. In: 2014 14th International Conference on Quality Software, pp 276–285, DOI 10.1109/QSIC.2014.11

Parnin C, Rugaber S (2009) Resumption strategies for interrupted programming tasks. In: 2009 IEEE 17th International Conference on Program Comprehension, pp 80–89, DOI 10.1109/ICPC.2009.5090030

Pennock DM, Horvitz E, Giles CL (2000) Social choice theory and recommender systems: Analysis of the axiomatic foundations of collaborative filtering. In: Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence, AAAI Press, p 729–734

Ramsauer R, Lohmann D, Mauerer W (2016) Observing custom software modifications: A quantitative approach of tracking the evolution of patch stacks. In: Proceedings of the 12th International Symposium on Open Collaboration, Association for Computing Machinery, New York, NY, USA, OpenSym '16, DOI 10.1145/2957792.2957810

Robbes R, Lanza M (2010) Improving code completion with program history. Automated Software Engineering 17(2):181–212, DOI 10.1007/s10515-010-0064-x

Robbes R, Pollet D, Lanza M (2010) Replaying ide interactions to evaluate and improve change prediction approaches. 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010) pp 161–170, DOI 10.1109/MSR.2010.5463278

Robillard M, Coelho W, Murphy G (2004) How effective developers investigate source code: an exploratory study. IEEE Transactions on Software Engineering 30(12):889–903, DOI 10.1109/TSE.2004.101

Robillard MP, Dagenais B (2010) Recommending change clusters to support software investigation: an empirical study. Journal of Software Maintenance and Evolution: Research and Practice 22(3):143–164, DOI https://doi.org/10.1002/smr.413

Rothlisberger D, Nierstrasz O, Ducasse S, Pollet D, Robbes R (2009) Supporting task-oriented navigation in ides with configurable heatmaps. In: 2009 IEEE 17th International Conference on Program Comprehension, pp 253–257, DOI 10.1109/ICPC.2009.5090052

Sahm A, Maalej W (2010) Switch! recommending artifacts needed next based on personal and shared context. In: Engels G, Luckey M, Pretschner A, Reussner RH (eds) Software Engineering 2010 - Workshopband (inkl. Doktorandensymposium),Fachtagung des GI-Fachbereichs Softwaretechnik, 22.-26.02.2010, Paderborn, GI, LNI, vol P-160, pp 473–484

Sanchez H, Robbes R, Gonzalez VM (2015) An empirical study of work fragmentation in software evolution tasks. In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp 251–260, DOI 10.1109/SANER.2015.7081835

Sillito J, Murphy GC, De Volder K (2006) Questions programmers ask during software evolution tasks. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Association for Computing Machinery, p 23–34, DOI 10.1145/1181775.1181779

Singer J, Elves R, Storey MA (2005) Navtracks: supporting navigation in software maintenance. In: 21st IEEE International Conference on Software Maintenance (ICSM'05), pp 325–334, DOI 10.1109/ICSM.2005.66

Soh Z, Khomh F, Guéhéneuc YG, Antoniol G (2013a) Towards understanding how developers spend their effort during maintenance activities. In: 2013 20th Working Conference on Reverse Engineering (WCRE), pp 152–161, DOI 10.1109/WCRE.2013.6671290

Soh Z, Khomh F, Guéhéneuc YG, Antoniol G, Adams B (2013b) On the effect of program exploration on maintenance tasks. In: 2013 20th Working Conference on Reverse Engineering (WCRE), pp 391–400, DOI 10.1109/WCRE.2013.6671314

Soh Z, Khomh F, Guéhéneuc YG, Antoniol G (2018) Noise in mylyn interaction traces and its impact on developers and recommendation systems. Empirical Software Engineering 23(2):645–692, DOI 10.1007/s10664-017-9529-x

Starke J, Luce C, Sillito J (2009) Searching and skimming: An exploratory study. In: 2009 IEEE International Conference on Software Maintenance, pp 157–166, DOI 10.1109/ICSM.2009.5306335

Teitelman W, Masinter L (1981) The interlisp programming environment. Computer 14(4):25–33, DOI 10.1109/C-M.1981.220410

Wang J, Peng X, Xing Z, Zhao W (2011) An exploratory study of feature location process: Distinct phases, recurring patterns, and elementary actions. In: 2011 27th IEEE International Conference on Software Maintenance (ICSM), pp 213–222, DOI 10.1109/ICSM.2011.6080788

Ying A, Murphy G, Ng R, Chu-Carroll M (2004) Predicting source code changes by mining change history. IEEE Transactions on Software Engineering 30(9):574–586, DOI 10.1109/TSE.2004.52

Ying AT, Robillard MP (2011) The influence of the task on programmer behaviour. In: 2011 IEEE 19th International Conference on Program Comprehension, pp 31–40, DOI 10.1109/ICPC.2011.35

Zimmermann T, Weibgerber P, Diehl S, Zeller A (2004) Mining version histories to guide software changes. In: Proceedings. 26th International Conference on Software Engineering, pp 563–572, DOI 10.1109/ICSE.2004.1317478

Zou L, Godfrey MW, Hassan AE (2007) Detecting interaction coupling from task interaction histories. In: 15th IEEE International Conference on Program Comprehension (ICPC '07), pp 135–144, DOI 10.1109/ICPC.2007.18