Empirical Software Engineering (2024) 29:147
https://doi.org/10.1007/s10664-024-10528-7

Consensus task interaction trace recommender to guide developers' software navigation

Layan Etaiwi[1] · **Pascal Sager**[2,3] · **Yann-Gaël Guéhéneuc**[4] · **Sylvie Hamel**[5]

Accepted: 2 July 2024

Abstract
Developers must complete change tasks on large software systems for maintenance and development purposes. Having a custom software system with numerous instances that meet the growing client demand for features and functionalities increases the software complexity. Developers, especially newcomers, must spend a significant amount of time navigating through the source code and switching back and forth between files in order to understand such a system and find the parts relevant for performing current tasks. This navigation can be difficult, time-consuming and affect developers' productivity. To help guide developers' navigation towards successfully resolving tasks with minimal time and effort, we present a task-based recommendation approach that exploits aggregated developers' interaction traces. Our novel approach, Consensus Task Interaction Trace Recommender (CITR), recommends file(s)-to-edit that help perform a set of tasks based on a tasks-related set of interaction traces obtained from developers who performed similar change tasks on the same or different custom instances of the same system. Our approach uses a consensus algorithm, which takes as input task-related interaction traces and recommends a consensus task interaction trace that developers can use to complete given similar change tasks that require editing (a) common file(s). To evaluate the efficiency of our approach, we perform three different evaluations. The first evaluation measures the accuracy of CITR recommendations. In the second evaluation, we assess to what extent CITR can help developers by conducting an observational controlled experiment in which two groups of developers performed evaluation tasks with and without the recommendations of CITR. In the third and last evaluation, we compare CITR to a state-of-the-art recommendation approach, MI. Results report with statistical significance that CITR can correctly recommend on average 73% of the files to be edited. Furthermore, they show that CITR can increase developers' successful task completion rate. CITR outperforms MI by an average of 31% higher recommendation accuracy.

Keywords Consensus algorithm · Recommendation systems · Mylyn interaction traces · Task-related interaction traces · Software navigation · Maintenance

Communicated by: David Lo

Extended author information available on the last page of the article

Published online: 02 September 2024

1 Introduction

Software companies understand the importance of tailor-made software systems that accommodate different needs, from clients' specific required features to technological frameworks. Accordingly, these companies are more inclined to develop customised software systems that meet their particular clients' demands better than off-the-shelf software systems, and more reliable than completely original systems. For example, many clients adopt custom software systems, such as Web and mobile applications, e-commerce solutions, CRM (with a global market to reach $50 billion by 2025 CRM 2022) and ERP (a global market to reach $78.4 billion by 2026 Oracle 2022) systems, to maintain ownership of unique systems, seamless integration with their existing systems, and increased security and reliability.

To build client-specific software, developers extend an original software system in the desired direction by forking. Traditionally, forking is the practice of copying a shared codebase, under a new name, to create a logically independent software system that may never be merged into the root codebase (Hammouda et al. 2012). Through this practice, a software company can create independent forks of their original software system, customize each fork's functionalities or add new features in response to client requests.

One of the challenges of building custom software systems is meeting the increasing clients' requests for new features and functionalities. As clients' requests grow, so do the size and complexity of each software instance. Thus, maintaining and developing each client's instance become more and more difficult, costly, and time-consuming (Soh et al. 2013a).

Indeed, software complexity increases the mental effort needed by developers, specifically newcomers, to comprehend, maintain, and evolve the software. In fact, program comprehension has been reported as one of the developers' main challenges (LaToza et al. 2006). It involves reading large volumes of documentation, navigating through large codebases, running complex systems, debugging tangled use cases, etc. It may take up to 35% of the developers' time to navigate and understand source-code files for particular change tasks (Ko et al. 2006). Thus, it requires developers to spend 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[1] required code change in only two files but it took the developer three days of navigating and understanding the code before making these changes (Lee et al. 2014).

Some of the change tasks that developers perform on customised software instances, whether they are development, maintenance, or evolution tasks, are the same for each client or very similar by virtue of clients having similar needs and using customised versions of the same software systems. An example of exact tasks can be a bug found in the codebase of a client's instance, which must be fixed in all instances. An example of similar tasks can be features that were implemented for some clients and then later requested by other clients. Consequently, completing these types of tasks for each client requires developers to interact with the same or similar source-code file(s) (Ramsauer et al. 2016). Hence, we define 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 can be implemented on each client's software instance and hence require developers to interact with the same or similar source-code file(s) to successfully perform them.

[1] https://bugs.eclipse.org/bugs/show_bug.cgi?id=261613

As a software developer performs a change task, she spends time understanding the software system, interacting and navigating through its source-code elements (*i.e.,* packages, files, classes, fields, methods, functions, *etc.*), and making modifications. The process of completing the change task generates events for every activity that the developer performs. After completing the change task, the developer obtains a task interaction trace consisting of source-code elements and their relationships in the form of events. In this study, we only consider collecting file-level events, *i.e.,* activities performed on system files. When the same or similar change tasks are performed by multiple developers on different instances, each developer obtains task interaction trace from completing each task. Eventually, for each particular change task, developers form a Task-related Set of Interaction Traces (TSITs).

Definition 3 Developers' events are generated by developers' activities on source-code elements (*i.e.,* opening, searching, editing, *etc.*) while completing a change task.

Definition 4 A developer's interaction trace (IT) is a set of events obtained by developers after the completion of a change task.

Definition 5 Task-related Set of Interaction Traces (TSITs) is a set of developers' interaction traces after the completion of similar change tasks.

Developers' interactions have been used to study and build tools that can assist developers to cope with the challenge of effectively and efficiently developing and maintaining software systems. In particular, some works used developers' interaction histories to present source-code elements with which developers frequently interacted to create exploration strategies and investigate how developers understand programs (Kersten and Murphy 2006; Soh et al. 2013b; Sahm and Maalej 2010).

Some previous works collected and analysed developers' interactions for predicting code context that can be used for navigating directly to related source-code elements when completing a change task (Biegel et al. 2015; Robillard 2008; Wan et al. 2020). Other works focused on mining interaction traces to recommend files-to-edit based on association rules (Lee et al. 2014; Singer et al. 2005; Ying et al. 2004), recommend navigation patterns (DeLine et al. 2005), and cluster interactions to recommend textually related elements (Lee and Kang 2013).

While these works help developers complete their tasks by recommending source-code elements, none considered studying and supporting instances of software and their similarities, or providing task-specific recommendations. Rather, they only considered software in a isolation and provided recommendations for the entire software system. Moreover, some of these works fail to provide developers with accurate-enough recommendations (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. Furthermore, most of these works overlooked newly-hired developers. They assumed that developers have some understanding of the software systems and required them to start interacting with task-related elements to use these elements as input to the approaches before making recommendations.

Some of the approaches, particularly MI (Lee et al. 2014), NavTracks (Singer et al. 2005) and Mining Change Histories (Ying et al. 2004), built their recommendations using association rules between elements frequently edited or viewed together, which lead to recommending unrelated elements when developers interacted with the "wrong" elements, typically because they do not yet understand the software systems. Thus, we argue that it is unrealistic to assume that developers can start interacting with relevant elements without any prior

experience with the software system to provide recommendations. In this study, we want to help developers, particularly newly-hired developers, understand and navigate through the files to perform change tasks by recommending tasks-related interaction traces that contain a set of relevant files-to-edit without the need to interact with the system beforehand.

We propose Consensus Task Interaction Trace Recommender (CITR); a task-based recommendation approach that recommends file(s)-to-edit to developers based on an aggregated set of interaction traces. We consider in this study selection and edit types of events for a broad context of input interactions and higher recommendation accuracy (Lee et al. 2014). The purpose of our approach is to utilize the formation of task-related set of interaction traces to recommend files that are relevant to a given set of similar change tasks. Thus, our approach targets developers' interaction traces from previously completed same or similar tasks on custom software's instances as input data rather than interaction traces from the software as whole. By applying a consensus algorithm to developers' interaction traces, our approach creates a consensus task interaction trace as a recommendation. Each recommendation comprises a consensus set of relevant file(s)-to-edit and help perform new change tasks that are similar to the input task-related interaction traces.

Our general hypothesis is that CITR can guide developers, whether they are familiar with the software or have no prior knowledge, through files navigation, allowing them to complete their tasks successfully, with substantially less time and effort, minimal navigation to unrelated files, and ultimately help with program comprehension and increase their productivity.

To investigate this hypothesis, we conduct a series of evaluations. We determine the accuracy of the recommendation results by defining a ground truth and using precision and recall metrics. In the second evaluation, we conduct an observational controlled experiment of 50 developers undertaking identical evaluation change tasks with and without CITR recommendations and quantitatively analyze their tasks completion success rate. We finally compare CITR against MI (Lee et al. 2014), an existing file-level recommendation approach.

- When comparing our results to the ground truth data, quantitative results indicate that our approach can recommend file(s)-to-edit with average precision of 74%, recall of 68%, and F-measure of 68%.
- A detailed qualitative analysis of the experiment supported by video recordings reveals that developers with CITR recommendations can complete their tasks in less than half of the time and effort needed by the control group and with a higher completion rate of 94%.
- The controlled experiment shows that developers with CITR recommendations understand and navigate through system elements better than those without. Developers in the control group performed unstructured navigation and relied on guessing and glancing.
- The comparison demonstrates that the recommendations returned by CITR have a higher accuracy than those of MI (Lee et al. 2014).

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. In Section 4, we review the related work. An overview of the consensus algorithms is discussed in Section 5. In Section 6, we implement a study to collect developers' ITs and apply the consensus algorithm to generate recommendations. In Section 7, we apply three evaluation methods to investigate the success of the approach. In Section 8, we report and discuss results. In Section 9, we discuss limitations. Finally, in Section 10, 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 tasks.

Through our defect management tool, client Bob reported a launch bug on the configuration page. He reported that when he adds a new plug-in to the system, the plug-in configuration page does not launch automatically. 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 her 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 a task-based approach that aggregates her colleagues' ITs, collected while completing the same or similar change tasks, and recommends her with one consensus task interaction trace that contains the file(s)-to-edit most relevant to the particular task that she is completing.

Developers working on subsequent tasks could query through the recommended consensus interaction traces 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

We present in the following an overview of the concept and steps of our approach, Consensus Interaction Trace Recommender (CITR). Figure 1 illustrates how our approach can be incorporated into the use case (Section 2). Having a customised software system forked into different clients' instances, developers regularly perform the same or similar change tasks on each client's instance. These developers use an event collection tool *e.g.,* Mylyn to collect their events with the software elements while completing tasks. In this study we consider both types of events; selection and edit. Including both types of events provides more context about related files to the task, allowing for a recommendation that can help completing a broader range of similar tasks. In comparison to using only edit events, using selection and edit events consistently increases the accuracy of file-level recommendations (Lee et al. 2014).

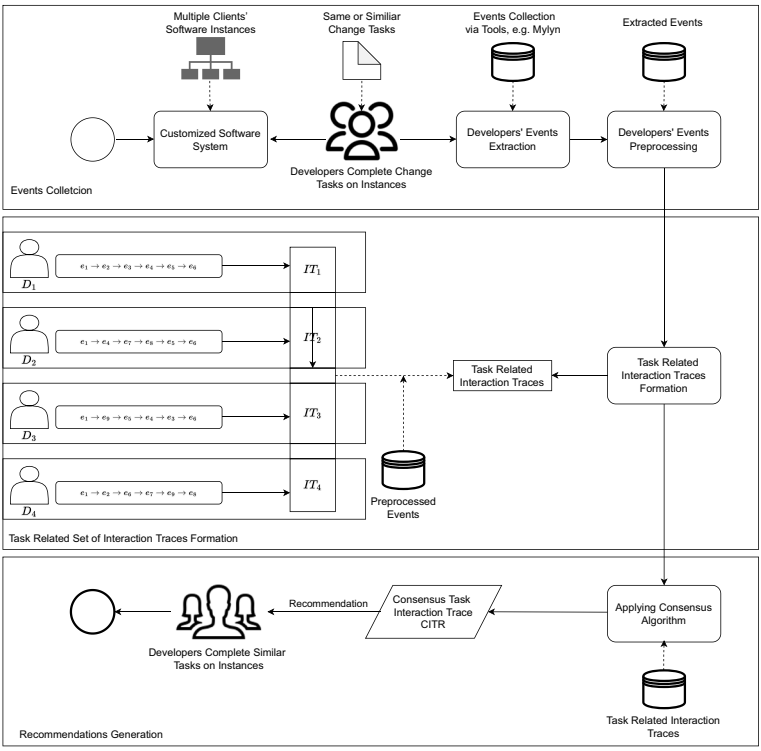


Fig. 1 Overview of the approach concept

Collected events from performing each tasks are extracted and go through multiple pre-processing steps for noise removal. Pre-processing includes steps like eliminating Mylyn trigger events, JAR files, and noise events. Once these events are pre-cprocessed, they are formed into an interaction trace (IT) for each developer from every completed task. The set of developers' interaction traces (ITs) from each completed task creates a Task-related Set of Interaction Traces (TSITs). Specifically TSITs contains a set of interaction traces from all developers who completed the particular change task on the same software instance or multiple different instances. Lastly, a consensus algorithm is then applied to the task-related set of interaction traces to generate a Consensus Task Interaction Trace. CITR contains a set of relevant file(s)-to-edit and that can be recommended to other developers to help them complete tasks that are the same or similar to the input tasks on other clients' instances.

To better illustrate the task-related set of interaction traces formation phase, we exemplify the use case in Figure 1. A set of developers $\{D_1, D_2, D_3, D_4\}$ complete the launch bug, task T, on the configuration page on different clients' instances using Eclipse. The developers preform a sequence of events $\{e_1, e_2, ..., e_n\}$ on the system files for the completion of the task. Mylyn collects the sequence of events from each developer. Events are then extracted to form an interaction trace for each developer $\{IT_1, IT_2, IT_3, IT_4\}$. The set of developers' formed interaction traces forms task-related set of interaction traces (TSITs) for change task T. This aggregation of developers' ITs into task-related set of interaction traces enables us to generate a recommendation based on developers' interaction histories with a system by employing a consensus algorithm that is able to produce a set of the most relevant files.

Applying the Consensus Algorithm step is the core of our recommendation approach. The algorithm takes as an input a task-related set of developers' interaction traces. It measures the distance between every two input interaction traces using a predefined measure. Finally, it generates a consensus task interaction trace that is closest to the input interaction traces and consists of files that are most relevant to the task at hand. In the next section, we discuss the history of the consensus algorithms, measures used, and how the algorithms function.

4 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.

4.1 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.

4.2 Recommendation systems

To find system elements relevant to a task that developers is trying to perform, 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, navigation, and eventually productivity.

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

Hipikat (Cubranic and Murphy 2003) on the other hand forms a group memory using source code versions, bugs, electronic communication, and web documents. It then recommends artifacts relevant to tasks based on inferring links between the archived artifacts in a group memory.

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.

4.3 Interaction traces based recommendation systems

Several works used developers' interaction traces to recommend or preditc 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 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.

Code context prediction was proposed by Wan et al. (2020). Based on learned abstract topological patterns from Mylyn interaction histories, the approach predicts code context as developers perform a development task.

Similarly, (Robillard 2008) proposed a similar approach called Suade. The approach predicts elements based on topology of a graph of structural dependencies for the software system. It takes as input a fuzzy set of elements that a developer interacted with and predicts a fuzzy set of elements that are of potential interest by calculating specificity and reinforcement

Although these studies use developers' interaction traces to build recommendation or prediction, they assume that developers come with some knowledge of the systems. Hence, they require developers to start interacting with the systems, use these interactions as seeds for the approach 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 and does not base the recommendation on association rules. Rather than considering the entire software system, our proposed approach generates task-specific recommendations. Thus, when we compared our approach to MI,

CITR can recommend file(s)-to-edit that are more relevant to the given change tasks than what MI recommended.

4.4 Studies of developer activities and behaviour

There are many works on developers' navigation behaviours and programming activities, factors that impact them, 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.

In line with other studies on developer behavior, (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 potentially ask.

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.

On the other hand, (Chattopadhyay et al. 2019) observed and recorded 10 developers programming activities while performing development tasks to study how developers structure their development effort and whether context impact the structure. They observed that developers organize their development work into a series of episodes with different patterns while trying to maintain context between episodes.

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.

5 Background on the consensus algorithms

5.1 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.

5.2 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:

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

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:

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

5.3 Consensus algorithms

Consensus algorithms have been applied in different domains such as bio-informatics (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 bio-informatics. Brancotte et al. (2015) studied 14 different algorithms using the generalized Kendall-τ distance and classified them into score-based 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.

6 Study setup

We carry out in the next Section three evaluation methods to assess the accuracy of the results of our recommendation approach and the extent to which CITR can improve developers' productivity maintaining and developing a software system. These evaluations are quantitative, qualitative, and a comparison to answer the following research questions:

RQ1 We answer this question by building ground truth data and quantitatively compare them with the results of CITR using precision and recall measures (in Section 7.1).

RQ2 To evaluate productivity and navigation behaviour, we conduct an observational controlled experiment of 50 developers performing evaluation change tasks with and without the recommendations of the CITR and compare their behaviours (in Section 7.2).

RQ3 We answer this question by comparing the results of our approach with MI recommendation results under the same set of conditions (Section 7.3).

To generate the recommendations, carry out the evaluations and answer these RQs, we conduct an experiment of participants performing change tasks to collect their events (data). Collected events are then extracted, pro-processed, and used as training data to generate recommendations.

Unlike previous studies, such as Zimmermann et al. (2004) and Lee et al. (2014), that extracted and used archived developers' events as training data to build recommendations, we collect our data through a participant-involved experiment. As previously stated, these approaches generate recommendations for the entire system. Therefore, the input data must be large and include all existing developers' events with the system. On the contrary, because we build recommendations at the task level in this study, the input data should be limited and come only from tasks that are similar to the tasks for which we are building recommendations.

To set up the experiment, we first choose a subject system (in Section 6.1), which is the same system that will be used for the evaluation (in Section 7). Then we define change tasks (in Section 6.2), recruit participants (in Section 6.3), and select tools for collecting the events (in Section 6.4). Next, we invite the participants to conduct the experiment by performing the change tasks and collect their events with the software elements (in Sections 6.5). After collecting and extracting the events, we pre-process them in Section 6.6. Finally, we form task-related set of interaction traces (TSITs) and apply the consensus algorithms on these TSITs to obtain recommendations (in Section 6.7)).

6.1 Subject system

Among a population of Java systems, we chose an Eclipse-based plug-in, PDE (Plug-in Development Environment), as subject system. PDE[2] 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 three components and we chose to consider only the PDE-UI component. Participants will interact with PDE-UI files to complete change tasks.

6.2 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[3]; Web based bugs tracking system. We queried for tickets that were marked as resolved and have a solution patch attached to them. Following that, we looked through these tickets at random, reading their descriptions, replicating the issues they described when it was possible, and putting the suggested fixes into action.

Some of the tickets were impossible to replicate because their descriptions were insufficiently detailed; 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 complexity and time needed to complete the candidate tickets, we hired a Ph.D. student with approximately five years of industry experience as evaluator. The evaluator performed the candidate tickets, noted the duration needed to complete each ticket and measured the complexity. We used three factors to measure the level of task complexity: time, navigation effort, and bug location. Bug location refers to whether the bug lies in a `core class` or a `module class`. `Core classes` implement the `core` functionalities of the system, highly coupled, and more complex to comprehend, like classes found in the *Launcher package* that handle the launching of plug-ins and features. Meanwhile, `module classes` implement additional features on top of `core`, cohesive, loosely coupled, and thus easier to comprehend, such as classes found in the `Feature package` or `Product package`. The evaluator marked as "easy" tickets that required a few minutes to complete, minimal navigation, and that were module-related. He rated as "moderate" tickets that required 20 minutes to an hour to complete, more cognitive effort to understand, and that were module-related. Finally, he marked as "difficult" tickets that required 20 minutes to an hour to complete, navigation through several files, and that were core-related. Lastly, he classified as "complex" any ticket that took him longer than one hour to complete or that he was unable to complete.

To the best of our knowledge, there is no theory that suggests the optimal length of the experiment task. However, to prevent participant fatigue, which might lead to interruptions or abandoning the experiment, we chose to set the maximum required task completion time to 1 hour. We enforced this choice by randomly selecting three moderate and two difficult

[2] https://www.eclipse.org/pde/

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

tickets as our change tasks, which should not take longer than 45 minutes to complete, with an additional 15 minutes of buffer time.

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 events. We take advantage of the change task used in Soh et al. (2018), adapt it as our sixth change task and use the collected events as part of our dataset. Table 1 presents detailed descriptions of the chosen change tasks.

6.3 Participants

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

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[4] to an online form collecting information about their gender, level of education, and their years of Java and Eclipse IDE experience.

We recruited 23 participants who filled in the form. Out of these 23 participants, 15 have over three years of Java experience, while the remaining eight participants have less than three years. They all have at least one year experience with the Eclipse IDE. We selected the 15 Java experienced developers to participate and complete the five change tasks. Among these 15 participants (referred to as $P1, ..., P15$), three are female, two are postdoctoral researchers, eight are doctoral candidates in software engineering, three are enrolled in a Master program in computer engineering, two are professional software developers, and all have 1 to 5 years of professional development experience. All were new to the system: none had worked on Eclipse PDE.

We sent an invitation email to each individual participant containing a brief description of the experiment. To avoid time conflicts, we scheduled each participant on a different date.

6.4 Events collection 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[5].

Developers' events with software systems are collected by task management and monitoring tools, such as Mylyn[6], 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 that is commonly used in research studies (Soh et al. 2018).

Mylyn is an Eclipse plug-in 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

[4] Online Form

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

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

Table 1 Change tasks used in the study and their descriptions

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.
240737	Task 4: Configuration Page is missing a title	the configuration page of a product project is missing a title. On the same page, the default value of the autostart is set to 2 instead of 0.
61894	Task 5: Add a "Sort Alphabetically" button	Add a sort button that sorts added plug-ins Alphabetically to the "Included Plug-ins", "Included Features", and "Dependencies" pages of a feature project.
188904	Task 6: Add a "Validate Plug-ins" button	On the Run Configuration settings page, add a "validate" button that validates plug-ins before adding them.

gathering events once the developers deactivate the Mylyn task. Then, it aggregates the collection of events, compresses, encodes, and exports them in XML format.

Mylyn events 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[7]. 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.

6.5 Events collection

We collected participants' events 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

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

Table 2 An example of Mylyn events

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

same settings, and using the same procedure. We thus could control the events collection and ensure 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 plug-in 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 2, on the left side, the PDE system is imported to the IDE and Mylyn tasks are created and ready to be activated under the Task List on the right side.

We divided the 15 participants into two groups. The first group contained seven participants, each of whom was asked to complete Change Tasks 1 and 2. The remaining eight participants made up the second group, which preformed Change Tasks 4, 5, and 6. We then explained to each participant the purpose of the work, directed them to the desktop station, and informed them that they would perform two/three 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 them to try to complete the 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

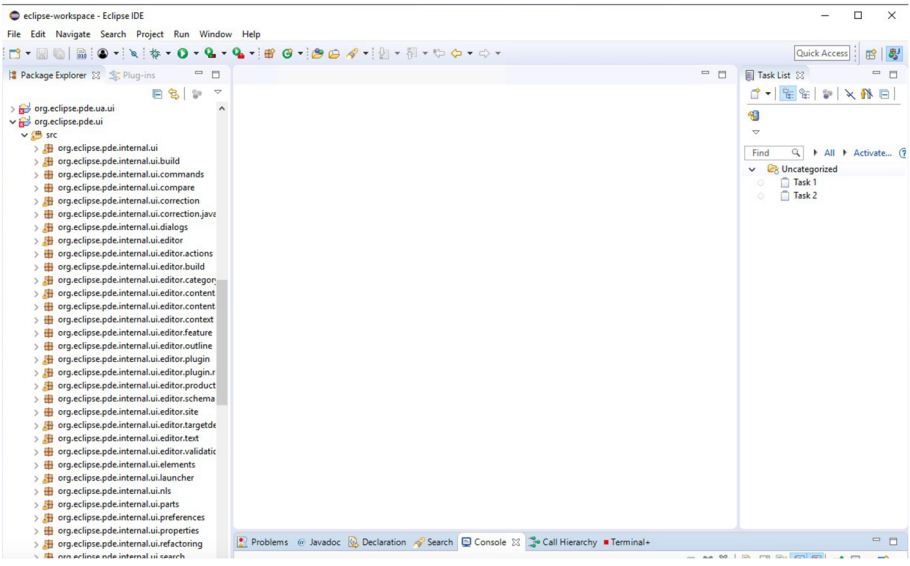


Fig. 2 A screen capture of the IDE before the start of a change task

```
<?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. 3 Sample of Mylyn event in XML format

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[8]. To provide the participants with a hint of what they have 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.

We then exported all Mylyn events from Eclipse IDE in XML format. The events obtained from the completion of the five Change Tasks by the 15 participants, together with events related to Change Task 3 from (Soh et al. 2018) by four participants, make a total of 5,550 events. Figure 3 represents a sample of an extracted Mylyn event.

6.6 Events pre-processing

We pre-processed each exported Mylyn event to extract participants' selection and edit activities and system elements on which the activities occur. This phase goes through multiple steps and starts by converting the extracted Mylyn XML files into CSV files.

As described in Section 6.4, 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 4 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.

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

Parts of StructureHandle:
[=] project [;] [package] [{ | ( ] [file] [" ["] [class] [[^ [attribute]] | [~ [method]]] [~ [rest]]

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

Fig. 4 Parts of Mylyn StructureHandle

The paths to elements in the StructureHandle contain special characters that create noise and make 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 5 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 events.

According to Lee et al. (2014) 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 6 compares two screenshots taken from the interactions 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.

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 CITR recommendations are available online[8].

Table 3 Identification of the parts of the StructureHandle in Figure 4

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`

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. 5 StructureHandle vs. CompleteName after special characters removal

6.7 Task-related interaction traces formation and generating recommendations

Each participant's set of events from completing one of the change tasks is grouped to form a participant's interaction trace (IT) and we labeled them as ($IT1$, ITs, ...$IT15$). Each set of participants' ITs from performing a task forms a task related set of interaction traces (TSITs). Given that the study experiment involved six change tasks, thus we were able to generate six TSITs.

Considering that the BioConcert and KwikSort algorithms provide best quality results (see Section 5.3) 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 interaction traces. 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 TSITs 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:

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

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. 6 Illustration of common events between participants hold the same ID number

Table 4 CITR after applying BioConcert and Kwiksort to TSITs of task T

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 shows consensus task interaction trace after applying both algorithms on task related set of interaction traces 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.

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 adopt BioConcert results as our approach recommendations.

Table 5 translates the results of CITR 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 the replication package[8].

7 Evaluations

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

Table 5 Translation of recommended consensus interaction trace from applying BioConcert into real participants' events

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`

7.1 *RQ1*: To what degree does CITR recommend relevant files to given change tasks?

We evaluate the quality of the results of CITR to determine whether or not the consensus algorithm can recommend accurate results by comparing the recommendations obtained in Section 6.7 against ground truth data.

A ground truth data is the ideal set of files with which a developer should interact with some, if not all, in order to complete a specific change task. We derived ground truth data for the six change tasks (Subsection 6.2) by using the Mylyn events that are attached to the Bugzilla tickets we used to create the tasks. The PDE developers use Mylyn to collect, extract and attach their interactions with the software while resolving the tickets. We extracted these attached Mylyn events, pre-processes these events to remove any possible noise by following the same pre-processing steps that we adopted in Subsection 6.6. Following that, we extracted a set of files from the set of pre-processed events that the PDE developer interacted with to resolve the ticket, whether it is a selection type of event or an edit type. These sets of files represent the ground truth data for each of the change tasks, which we use to compare to the results of the approach recommendations. Table 6 presents the ground truth data created for Change Task 3. The remaining ground truth data for the other five tasks are available online[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 CITR.

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

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

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

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

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

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.

Table 6 Ground truth data created for change task 3

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

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

We answer this RQ qualitatively by conducting a Between-Subjects experiment with 50 developers performing a set of evaluation change tasks. The following details the experiment.

Setup We use the same system, PDE-UI, as in Section 6.1 to perform this evaluation experiment as it is the subject system for this study.

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 choose evaluation tasks that developers in this experiment perform to evaluate whether CITR recommendations can help them complete the tasks and increase their productivity. Specifically, we choose evaluation tasks that are similar in context to the change tasks that were used to generate the recommendations (Subsection 6.2).

To obtain evaluation tasks, we examined tickets on the Eclipse Bugzilla tracking system in three phases. In the 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) attachment contains a patch file. The patch file is needed to help us examine the proposed fix for each ticket and evaluate the tickets' complexity in the last phases.

In the second phase, we extracted the tickets descriptions. We read through the descriptions and looked for keywords to identify tickets that share the same context as the six change tasks. Eclipse is comprised of a range of products, with PDE being one of the them. PDE is made up of serveral components. We used the PDE-UI as subject system. PDE-UI divides into `core` and several `modules`, including product, feature, plug-in, context, and others. We considered tickets that reported bugs occurring in one `module` as sharing contextual similarities. For example, Change Task 1 pertains to an issue present in the plug-in `module`. Thus, we searched for tickets containing the keyword plug-in, reviewed their descriptions, and confirmed their relevance to the plug-in `module`. From all the extracted tickets, we found many tickets that have similar context with the change tasks and categorised them into `core` and `modules`.

In the third phase, two authors randomly chose tickets from the second phase, performed them, assessed their complexity based on four categories: easy, moderate, difficult, and complex; description of these categories can be found in Section 6.2. For equitable evaluation, we targeted moderate and difficult tickets that require moderate file navigation effort, call for one to three files of source-code modification and a maximum of 30 minutes to complete. We also invited two Ph.D. students with prior professional experience but no background in the subject system to perform the tickets, confirm their complexity, and if they could complete them within 30 to 45 minutes.

Finally, with the assistance of the Ph.D. students, we considered 12 evaluation tasks that were similar in context to the change tasks. Out of the twelve tasks, three are considered difficult, while the remaining nine are moderate. We describe them 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 6.3.

Table 7 Evaluation tasks description

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	While browsing the plug-ins page, you will receive a prompt to save any changes, even if you have not made any modifications.
171767	Large font on main tab	Increasing the dialog font size results in a portion of the main tab disappearing.
101516	Sort alphabetically property	It would be helpful to have the option to alphabetically sort the listed extensions in the plug-in XML editor's extensions section.
269107	Some controls are enabled incorrectly	The browse buttons on the Configuration and Launching tabs should be greyed out if they are not selected.
88566	Tab description is missing	Description of the Included plug-ins tab is missing.
204404	Unable to use checkbox	Toggling the "Only Show Selected" checkbox in the run configuration window does not trigger any changes. The checkbox should only show selected plug-ins.
223727	Configuration section overview is missing	Incorporate a section overview into the Configuration section of a product project.
38536	Up and Down buttons are missing	The Included Features and Build tabs should contain Up and Down buttons, which enable the user to navigate through records.
312156	Malfunctioned button	The "Select Feature" button on the plug-ins tab should allow for the addition of more features, but clicking it does not result in any action.

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.

We recruited 56 developers (referred to as $D1$, .., $D56$), out of whom six abandoned before we scheduled them for the experiment. Out of the 50 remaining developers, five were senior undergraduate students, 24 M.Sc. students, and 21 Ph.D. students. 52% of the developers had programming experience of over 5 years, with an average of 3 years of Java programming experience. All developers had industrial programming experience (11 of them with more than 5 years, while the remaining 39 between 1 to 5 years). All developers reported using different IDEs but being unfamiliar with the subject system. We provide detailed participant demographics in Table 8.

Procedure We applied the Between-Subjects design (Erlebacher 1977) in which there is a control group and an experimental group, and each developer experiences only one level of a single independent variable. Our independent variable is the recommendations with two levels: with and without. We split the 50 developers into a control group and an experimental

Table 8 Demographics of selected developers

Education	
Bachelor	5
Masters	24
Doctorate	21
Programming Experience	
5 Years or More	26
3-5 Years	12
1-3 Years	12
0-1 year	0
Java Experience	
5 Years or More	5
3-5 Years	5
1-3 Years	20
0-1 year	20
Eclipse IDE Experience	
5 Years or More	6
3-5 Years	5
1-3 Years	12
0-1 year	27
Professional Experience	
5 Years or More	11
3-5 Years	9
1-3 Years	15
0-1 year	15

group (25 developers in each group). We ensured that developers' education and professional experience varied in each group. In the control group, we requested developers to perform the evaluation tasks without the recommendations of CITR, while we provided developers in the experimental group with the recommendations.

To ensure that the set of evaluation tasks were performed by developers with different experience levels, we split the 12 tasks into two sets A and B, with six evaluation change tasks in each set. Therefore, we further divided the control and experimental groups into two sub-groups, with each sub-groups performing a different set of evaluation tasks. Figure 7 illustrates this division. Furthermore, we chose to have the two groups of developers conduct the experiment on the same instance of the PDE-UI rather than on different customised instances. Carrying out the experiment on the same instance should eliminate the risk of significant result variation between developers and allow us to compare the outcomes of the defined measures between the two groups on an equal footing.

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 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 remotely access the laboratory desktop where the subject system was installed. For developers in the second group of the experiment, their

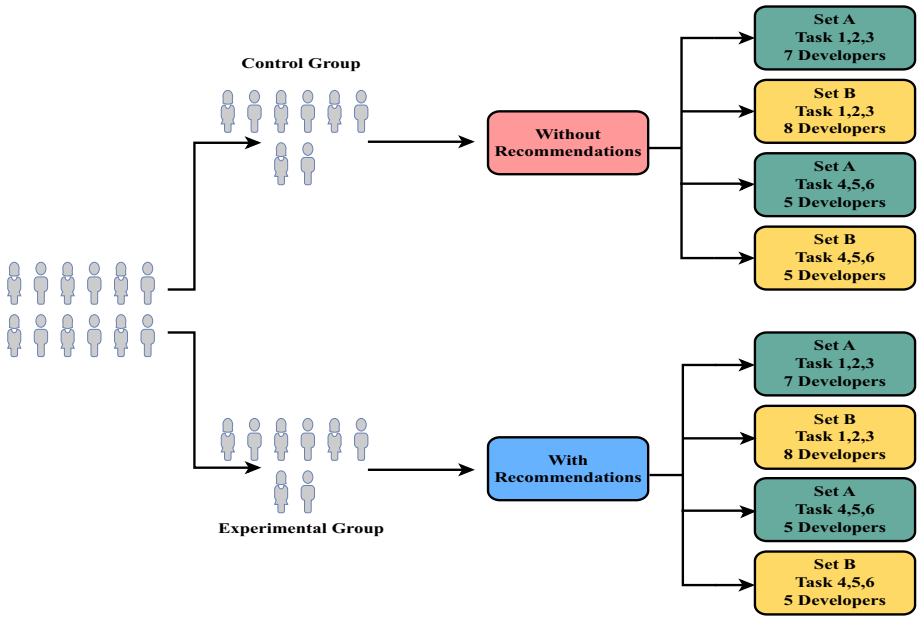


Fig. 7 Divisions of the developers into groups and sub-groups

task description document recommended the consensus task interaction traces along with each task.

Once the developer is 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 familiarise 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 some of the authors and 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 control group of developers. As time is one of the evaluation factors when evaluating the success of our approach, we decided to limit the time allotted to the experimental group to 20 minutes to eliminate the possibility that developers performing unnecessary navigation knowing that they have a set of recommended files with more than enough time. In both groups, we gave developers the option to request an additional five to fifteen minutes if necessary,reassuring them not to worry if they were unable to complete a task and advance to the next 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 CITR 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 tasks.

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 analysed the video recordings of the 50 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 what files the developers interacted with. 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 summarised the most interesting observations from each group separately. The analysis helps us define the kind of actions developers do in completing the assigned tasks and whether providing a set of recommendations helps improve their navigation 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 CITR approach in general. We also sent them an online post-experiment questionnaire with a series of exploratory questions. The question-

naire 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 eight 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. Five of the eight questions were rating scale questions, one was yes/no questions, while the remaining two were open-ended questions. The questionnaire is presented in Table 9.

7.3 *RQ3*: How does CITR compare to MI (Mining Programmer Interaction Histories) Lee et al. (2014) 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. 2014), 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 CITR. 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 a query formed from the current developer's interaction with a given task at the time of recommendation (Lee et al. 2014).

Our hypothesis is that CITR recommends more relevant file(s)-to-edit than the state-of-the-art because it forms recommendations from interaction traces related to the same or similar change tasks rather than interactions from the entire system.

Table 9 Post-experiment questionnaire

Questions	Type of Answer
Q1: When working on unfamiliar software systems, do you have difficulty knowing where to start?	(1-5)
Q2: How difficult was locating files related to the tasks at hand using the list of recommended files?	(1-5)
Q3: Having to deal with unfamiliar software system, do you think recommending files eased locating files related to the task and program comprehension?	(1-5)
Q4: 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)
Q5: 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)
Q6: Do you think you could rely on the recommended files to help you complete the tasks?	Yes/No
Q7: Is there anything you dislike about the proposed recommendation approach?	Open-Ended
Q8: 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	

Context in MI is a core component that triggers recommendation. Given a developer's set of events from interacting with a system, multiple contexts are formed from the last events. MI defines a *v-e* sized sliding window that holds a set number of view (*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 then finds association rules by checking whether interaction traces contain the events in the current context and recommends files from these ITs that satisfy the condition. MI introduces several methods for creating a context:

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

Procedure We simulated file(s)-to-edit recommendations in MI using the interactions generated from the six change tasks described in Section 6. According to Lee et al. (2014), different context creation methods generate 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, in particular MI-EA and MI-VOA, since they demonstrated better performance.

Regarding the size of the sliding window, the authors of MI suggested a *v-e* sized sliding window between 1 and 10. Considering that we are applying MI to ITs from completing change tasks rather than ITs from an entire system and considering 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 6. Hence, we created multiple contexts by setting the number of $(v = n)$, decreasing n from 6 to 1, and the number of $(e = n)$, decreasing n from 5 to 0. Thus, we set the *v-e* sliding window between 6 and 0 for Change Task 2, 5 and 6, between 5 and 0 for Change Task 4, and between 4 and 0 for Change Task 1 and 3. For each *v-e* value, we run the simulation repeatedly over all the participants' interaction traces.

Measures We evaluated MI recommendation results against CITR recommendations and assessed which approach recommended more relevant file(s)-to-edit using precision, recall, and F-measure. We used the sets of ground truth data created in Subsection 7.1 as a baseline for the comparison. The formula of the three measures is presented in same sub-section.

8 Results and discussions

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

8.1 *RQ1*: To what degree does CITR recommend relevant files to given change tasks?

Table 10 presents the precision, recall, and F-measure values that result from comparing the accuracy of the results of CITR to the ground truth data from the six change tasks. The accuracy of recommendations is considered acceptable when more than half of the recommended files are correct. Meaning, precision and recall values are .5 or higher.

The precision and recall values of the CITR recommendations generated from developers' interaction traces of Change Tasks 1, 3, 4, 5, and 6 are encouraging. Results from Change Tasks 1 and 3 generate a precision value of 1.00; CITR produces recommendations that are accurate and specifically correspond to the files needed to complete these change tasks 100% of the time. Furthermore, the recall rates show that 100% and 50% of the recommended files were in the ground truth. In general, Change Tasks 4, 5, and 6 have over 50% precision and recall rates. The precision, recall, and F-measure values of the CITR recommendations for Change Task 2 are least satisfactory. However 33% of the recommended files are still relevant and overlap with the files in the ground truth, with a recall of 33%.

To investigate the reasons behind the resulting lower values from recommendations generated from Change Task 2, which stem from false negatives, we examine thoroughly the set of files in the ground truth data and compare them to the result of CITR. The ground truth data includes three files `DocSection.java`, `SchemaFormOutlinePage.java`, and `DocumentSection.j- ava`, while CITR recommended six files. Resolving the bug in Change Task 2 required a code modification in only a single file "`DocSection.java`" and the other two classes are irrelevant.

We investigate the other two 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 and edited these unrelated files for other purposes *e.g.,* fixing another bug, without switching off Mylyn. Thus these two files were collected by Mylyn.

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, CITR 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 CITR provides more relevant files than the ones in the ground truth data, 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 answered the first research question that considers the quality performance of CITR. Overall, CITR results achieved relatively high average precision (73%), recall (66%), and F-measure (67%).

Table 10 Precision, Recall and F-measure values of the results accuracy

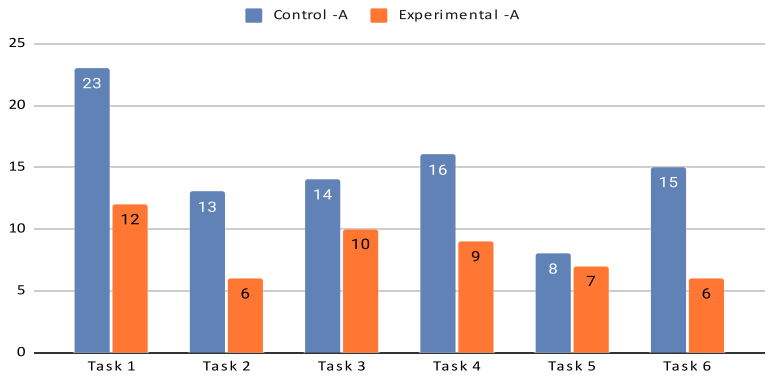
	Precision	Recall	F-measure
Task 1	1.00	1.00	1.00
Task 2	0.17	0.33	0.15
Task 3	1.00	0.50	0.66
Task 4	0.80	0.67	0.73
Task 5	0.75	0.75	0.75
Task 6	0.71	0.83	0.77

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

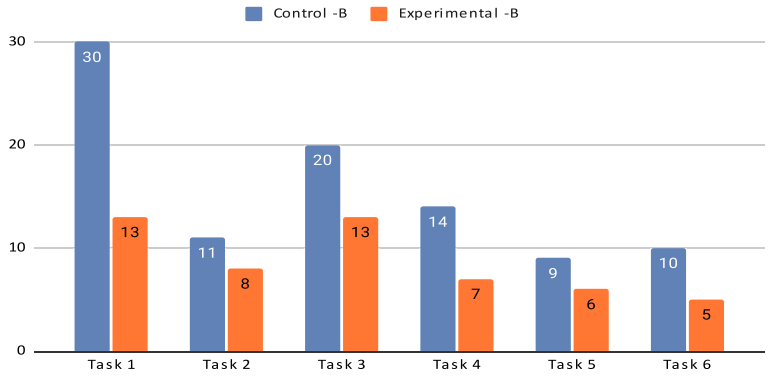
8.2 *RQ2*: Given a change task, can CITR help guide developers' navigation paths to relevant file(s)-to-edit and increase their productivity?

Developers success level Figure 8a and b compare the average time (in minutes) spent on each evaluation task from Set A and B by the control group to the experimental group.

The average completion time for each evaluation task shows that the control group spent more time than the experimental group. For the majority of tasks, developers in the experimental group completed the tasks successfully in half the time of the control group. In Task Set A, developers with recommended file(s)-to-edit spent on average 12, 6, 10, 9, 7, and 6 minutes, respectively, less average compared to developers in the control group. Similarly, developers could complete the tasks in Task Set B in an average of 13, 8, 13, 7, 6, and 5



(a) Average Time Needed to Complete Tasks in Set A



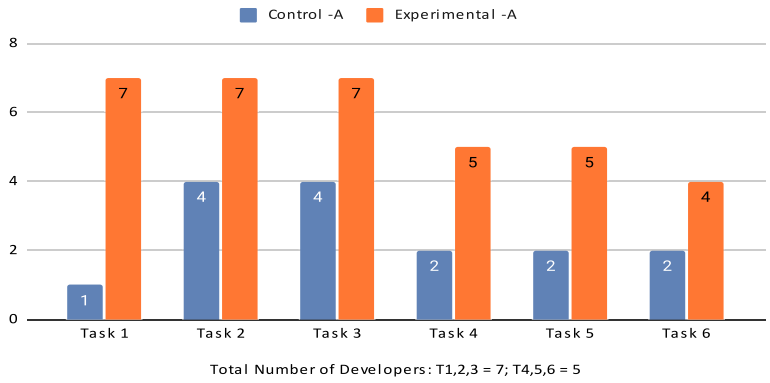
(b) Average Time Needed to Complete Tasks in Set B

Fig. 8 Average time needed to complete tasks in both sets by the two groups

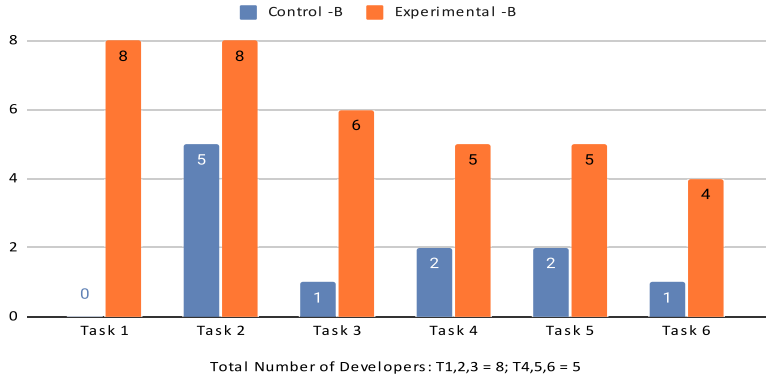
minutes. Detailed tasks completion time by developers from both groups is available in the replication package[8].

The data shows that the two groups had the longest average completion time for Task 1 in Task Set A. Although the developers were given a practice task before the start of the experiment, they still had to navigate through random classes while performing Evolution Task 1. In contrast, the remaining tasks were completed in a shorter amount of time as the developers had already gained familiarity with the system from the practice task and Evolution Task 1. Remarkably, developers from both groups spent a nearly identical average time to accomplish Task 5 in Task Set A. The task complexity level and the developers' competence may explain this observation.

In Task Set B, we observe that the control group had an average completion time of 30 minutes for Task 1, indicating that all developer used all the given time to complete the task. While the experimental group spent the longest amount of time on the completion of Task 1, when compared to the other tasks. The long average spent time on Task 1 is possibly due to the difficult level of complexity of this task and the previously mentioned lack of familiarity with the system. Similarly, Task 3 required a high average time from the two groups. In



(a) Number of Developers Completed Tasks in Set A



(b) Number of Developers Completed Tasks in Set B

Fig. 9 Number of developers completed tasks in both sets by the two groups

contrast to the other tasks which are bug-related, this task is specifically a feature request: developers must identify three files-to-edit, leading to a longer navigation duration.

Regarding completion factor, Figure 9a and b present the numbers of developers from each group who completed each evaluation task in the two Sets. They show that the experimental group outperformed the control group in terms of task completion rates in both Task Sets. Examining the completion rates in Task Set A reveals that, among the seven developers in the experimental group, the completion rate was 100% for the three first evolution tasks. Only one out of the seven developers in the control group completed Evolution Task 1 and four completed Evolution Tasks 2 and 3. In addition, only two developers out of five successfully performed Tasks 4, 5, and 6, respectively.

In Task Set B, files related to the completion of Evolution Tasks 1 and 2 were identified successfully by all the eight developers in the experimental group. Similarly, 40% (2 out of 5) of the developers located the files to solve Tasks 4 and 5. The complexity level of Task 1 and 6 is difficult as they both are core-related bugs. Predictably, none of the developers in the control group successfully completed Task 1. For Task 6, only one developer from the control group completed the task, compared to four out of five developers in the experimental group. Task 3 is a feature request that demands significant file navigation, thus it is not a surprise that only one developer could successfully locate its files, while six out of eight were successful in the experimental group.

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 in Task Set B. 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 CITR helped diminish the average time and navigation effort for the experimental group, there is still a disparity between the actual total time spent by each developer in the group. To illustrate Table 11 shows the demographics of these developers in the group and the time in minutes took them to complete tasks in Task Set A. Evidently, developer's efficiency and experience impact the amount of time and effort when dealing with an unfamiliar software system. Developers without CITR 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.

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

CITR 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. 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. For example,

one of the tasks reports the appearance of white lines between fields on the configuration tab. Some developers were searching for the names of the fields that are on the tab rather than the term "configuration tab" using the search functionality. The incorrect search keywords resulted in the return of unrelated classes and developers spending all of their time skimming through unrelated 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. One of the developers, for instance, who was working on the configuration tab task, began by expanding every package in the package explorer window, reading through the names of files, and randomly opening and closing files. When we questioned him about it during the after-experiment interview, he explained that he was searching for a class with the name "configuration tab" while looking through the files in the package explorer.

Table 11 Developers' Experiences Result in Different Task Completion Time

Experimental Group	
(a) Demographics of Experimental Group	
Education	
Bachelor	3
Masters	13
Doctorate	9
Programming Experience	
5 Years or More	15
3-5 Years	4
1-3 Years	5
0-1 year	0
Java Experience	
5 Years or More	3
3-5 Years	2
1-3 Years	12
0-1 year	8
Eclipse IDE Experience	
5 Years or More	5
3-5 Years	1
1-3 Years	7
0-1 year	12
Professional Experience	
5 Years or More	8
3-5 Years	3
1-3 Years	7
0-1 year	7

Table 11 continued

(b) Completion Time by the Experimental Group for Tasks in Task Set A

Tasks - Set A

	Task 1	**Task 2**	**Task 3**
P1	20	6	15
P2	6	4	5
P3	10	10	13
P4	7	5	11
P5	15	8	16
P6	15	8	9
P7	10	4	5
	Task 4	**Task 5**	**Task 6**
P1	10	10	0
P2	6	8	6
P3	15	5	6
P4	11	9	8
P5	7	4	4

The experimental group, on the contrary, followed one same navigation approach. All developers started performing the tasks by navigating to every recommended file, reading through the source code, and identifying related methods or functions to be edited to resolve the bug. Despite that we only asked the developers to point out the files that should be edited to complete the tasks, some developers went even one step further and specified the source code that needed to be changed or even made the change.

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. We also observed that some developers did not make an attempt to comprehend the underlying cause of the bug, which would ease the identification of the files to be changed to fix it. Instead, they simply relied on guessing whether a file was relevant by observing the number of keywords in the file that matched the user interface.

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. The random browsing and scrolling through files in the package explorer led to a few developers abandoning the tasks within 20 to 30 minutes.

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 CITR 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 we asked developers to locate file(s)-to-edit that were not part of the set of recommendations, in the case of Evolution Task 3 and 5, we observed that developers followed two different strategies. In the first strategy, they followed the relevant methods cross-reference to locate these files. The second strategy relied on the Eclipse built-in search tool to search for relevant keywords within the current open window, followed by locating relevant methods and classes. From these observations, we found that the experimental group could apply a more structured navigation, guide their attention and effort to understanding relevant system elements, avoid investigating irrelevant files, and efficiently determine more related files.

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

Studies suggest that companies should collect and store their developers' daily interactions (Bao et al. 2017). The findings of our observational experiment demonstrated that using developers' interactions with the system can enhance navigation by resulting in a more structured behavior, as well as boost developers' productivity by reducing navigation effort and time. Consequently, we encourage software companies to incorporate interaction collection through their daily operations in order to increase productivity, advance software development, and, ultimately, satisfy client demands.

User-experience and feedback We collected developers' answers to the interview/questionnaire questions, compared, and summarised them. Questionnaire results are reported in Table 12.

When asked about the difficulties in finding an entry point or knowing how to start debugging, 68% of the developers stated that it is very difficult while 32% had moderate to easy time locating an entry point.

In **Q2**, we asked the developers to rate the difficulty of completing the tasks using CITR recommendations. Most developers (20 out of 25) strongly agreed that completing the tasks using CITR recommendations was not difficult at all, while the remaining five seemed to have difficulty. These answers support that a few developers could not successfully complete the tasks.

Table 12 Post-experiment questionnaire answers

	1	2	3	4	5
Q1	Not at all 1 - 4%	3 - 12%	4 - 16%	10 - 40%	Very Difficult 7 - 28%
Q2	Not at all 6 - 24%	9 - 36%	5 - 20%	5 - 20%	Very Difficult 0 - 0%
Q3	Not at all 0 - 0%	0 - 0%	1 - 4%	6 - 24%	Absolutely 18 - 72%
Q4	Not Related 0 - 0%	0 - 0%	3 - 20%	6 - 24%	Very Related 16 - 64%
Q5	No Time Saved 0 - 0%	0 - 0%	1 - 4%	8 - 32%	Saved Time 16 - 64%
Q6	Yes 24 - 96%	No 1 - 4%			

All developers strongly agreed with **Q3** that CITR 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 CITR recommended files to the given tasks in **Q4**: 88% stated the recommendations were very relevant. In **Q5**, all developers expressed a positive impression of how CITR helped them spend less time navigating through system elements because CITR provided them a few entry points to start with.

> 96% of the developers confirmed that they could completely rely on CITR 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 CITR 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 viewed the set of recommendations as entry points to the system, which helped them avoid aimlessly searching through the package explorer and saved them valuable time.

Two of the developers from the experimental group who did not complete one of their tasks still found completing the tasks and navigating through the files challenging even with having CITR recommendations. They reported that CITR limited their search space and directed their navigation, however being a newcomer to the system made it daunting to skim through the files and identify the ones to edit. 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 experience, during the interview they confirmed that the experience is more of educational experience rather than practical experience.

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

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

8.3 *RQ3*: How does CITR compare to MI (Mining Programmer Interaction Histories) Lee et al. (2014) 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* 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 truth data. The values from the results of the two methods of MI are then analyzed. Based on the results, we measure the effectiveness of CITR by comparing the statistical values of CITR to MI.

Generally, we observed that simulated file(s)-to-edit recommendations from MI vary using the two recommended context formation methods and various v and e values. Table 13 presents the recommendation results of applying *MI-EA* to interaction traces of Change Task 1 with different *v-e* sliding windows along with CITR recommendations and the ground truth of the same task. Unsurprisingly, as the value of the sliding window decreases, MI makes recommendations at more edit events and hence recommends a higher number of files-to-edit. We also observed that the the set of recommended files differs for every set of *v-e* values in each run. One potential reason for this difference is that MI uses the files in the context to recommend files-to-edit by mining association rules through the set of interaction traces. As the context gets updated, MI creates new rules, which generate different recommendations based on the current context.

Furthermore, we observed that when $v = 4$ and $e = 3$, *MI-EA* recommendations do not intersect with CITR recommendations nor the ground truth. This mismatch is possibly due to the presence of these three files {`Pderesources.properties, PDEUIMessages.java, VersionDialog.java`} in the context at the time of recommendations, and MI does not recommend files that are included in the context. On the contrary, when *v-e* are set to their minimal values, $v = 1$ and $e = 0$, MI recommends all the three correct files. However, it recommends a number of irrelevant files that have a negative impact on the quality of the recommendations. This pattern of recommendation was evident across most recommendations from the six change tasks. All MI-EA recommendation results can be found online[8].

Table 14 compares the recommendation results of *MI-EA* with *MI-VOA* methods under the same set of sliding windows. *MI-VOA* yielded a significantly larger amount of recommended files than *MI-EA*, all with the same *v-e* values. The higher number of recommendations can be attributed to the fact that *MI-VOA* generates recommendations at view and edit events, while *MI-EA* only provides recommendations at edit events.

Given that *MI-EA* provided less better quality recommendations over *MI-VOA* in the case of our dataset, we further investigate the accuracy statistically by presenting precision, recall, and F-measures values of the generated recommendations from *MI-EA* method for the six change tasks, and comparing these values with those derived from CITR.

We computed precision, recall, and F-measure values using the set of results from *MI-EA* and the set of ground truths. Figure 10 shows the resulting precision and recall curves of the recommendation results from *MI-EA* under various (*v-e*) sliding windows. The data depicted in the graphs indicates that, when the values of the (*v-e*) sliding window are at their maximum, *e.g.,* between 6 and 4, the values of precision and recall are either at 0 or below 0.50. When the (*v-e*) values are high, the majority of the relevant view and edit events occur within the context and MI does not recommend events that are already presented in the context. In contrast, the results show consistently higher recall values (average of 0.72) for Tasks 1, 3, 4, 5, and 6 when the context contains no edit events ($e = 0$). This suggests that MI-EA can recommend the majority, if not all, relevant files with view-only events in the context. Nevertheless, most of these tasks exhibited a notably low precision, with average value of 0.26 for Tasks 1, 2, 3, and 4, which shows that the proportion of relevant files among all predictions is minimal, as evidenced in Table 14.

Table 13 Simulation results of *MI-EA*, CITR recommendations along with ground truth data from change task 1

MI-EA (v=4, e=3)	MI-EA (v=4, e=2)	MI-EA (v=2, e=1)	MI-EA (v=1, e=0)	CITR - GT
PluginVersionPart.java	FeatureSection.java	FeatureSection.java	FeatureSection.java	Pderesources.properties
	FeatureSelectionDialog.java	FeatureSelectionDialog.java	FeatureSelectionDialog.java	PDEUIMessages.java
	IHelpContextIds.java	IHelpContextIds.java	IHelpContextIds.java	VersionDialog.java
	IPreferenceConstants.java	IPreferenceConstants.java	IPreferenceConstants.java	
	PDELabelProvider.java	PDELabelProvider.java	PDELabelProvider.java	
	Pderesources.properties	Pderesources.properties	Pderesources.properties	
	PluginVersionPart.java	PDEUIMessages.java	PDEUIMessages.java	
	Utilities.java	PluginVersionPart.java	PluginVersionPart.java	
		Utilities.java	Utilities.java	
			VersionDialog.java	

Table 14 Results of MI-EA against MI-VOA from change task 1

MI-EA (*v*=**4**, *e*=**3**)	***MI-VOA*** (*v*=**4**, *e*=**3**)
`PluginVersionPart.java`	`FeatureSection.java`
	`FeatureSelectionDialog.java`
	`IHelpContextIds.java`
	`IPreferenceConstants.java`
	`PDELabelProvider.java`
	`PDEPlugin.java`
	`PluginSection.java`
	`PluginSelectionDialog.java`
	`PluginVersionPart.java`
	`Utilities.java`
MI-EA (*v*=**1**, *e*=**0**)	***MI-VOA*** (*v*=**1**, *e*=**0**)
`FeatureSection.java`	`AbstractCreateFeatureOperation.java`
`FeatureSelectionDialog.java`	`BuildSiteAction.java`
`IHelpContextIds.java`	`CreateFeaturePatchOperation.java`
`IPreferenceConstants.java`	`FeatureSection.java`
`PDELabelProvider.java`	`FeatureSelectionDialog.java`
`Pderesources.properties`	`GenerateFeatureBuildFileAction.java`
`PDEUIMessages.java`	`IHelpContextIds.java`
`PluginVersionPart.java`	`IPluginContentWizard.java`
`Utilities.java`	`IPreferenceConstants.java`
`VersionDialog.java`	`JUnitWorkbenchLaunchShortcut.java`
	`LaunchAction.java`
	`MainTab.java`
	`NewFeatureProjectWizard.java`
	`OrganizeManifest.java`
	`PDELabelProvider.java`
	`PDEPlugin.java`
	`pderesources.properties`
	`PDEUIMessages.java`
	`PluginSection.java`
	`PluginSelectionDialog.java`
	`PluginVersionPart.java`
	`Utilities.java`
	`VersionDialog.java`

To evaluate CITR recommendations accuracy and relevancy, we used MI results as a comparison baseline. To facilitate the comparison, we take the average precision and recall values of all the (*v*-*e*) sets for each change task. Figure 11 shows that the recommendation accuracy of CITR is consistently higher than that of MI across all change tasks, except for Change Task 2 where the accuracy is almost identical. For example, CITR recommends files-to-edit from Change Task 1 with precision–recall ratio of 1.00, while MI only yields ratio of 0.17–0.52. These findings indicate that CITR was capable to recommending all relevant files

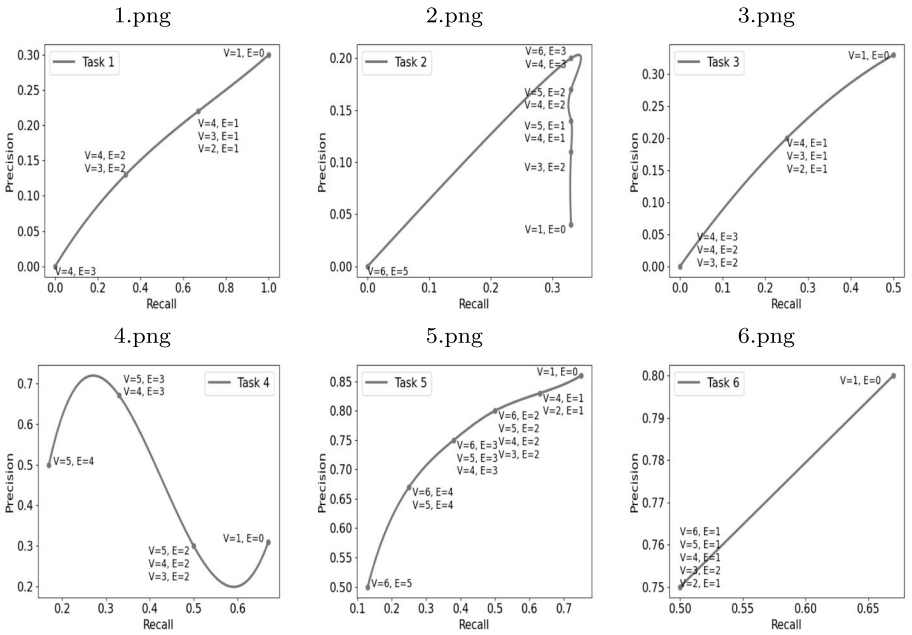


Fig. 10 Precision and recall curves of MI-EA recommendations for all change tasks

without any false positives. Similarly, CITR showed equivalent performance for Change Task 3 and 4 with precision–recall ratios of 1.00–0.50 and 0.80-0.67, respectively. In contrast, MI only achieved precision–recall ratios of 0.13–0.18 and 0.34–0.50, respectively. Consequently, CITR significantly outperforms MI in terms of F-measure values. As shown in Fig. 12, our approach shows F-measures values of 1.00, 0.66, and 0.73 for Change Tasks 1, 3, and 4. Whereas MI had average F-measure values of 0.26, 0.15, and 0.38.

As shown in the results, CITR and MI yielded comparable results for Change Task 2, with precision–recall ratios of 0.17–0.33 and 0.24–0.31, respectively. Section 8.1 discussed the reasons behind the lower precision–recall ratios, which we attributed to the presence of irrelevant files in the ground truth. Moreover, the results from Change Tasks 5 and 6

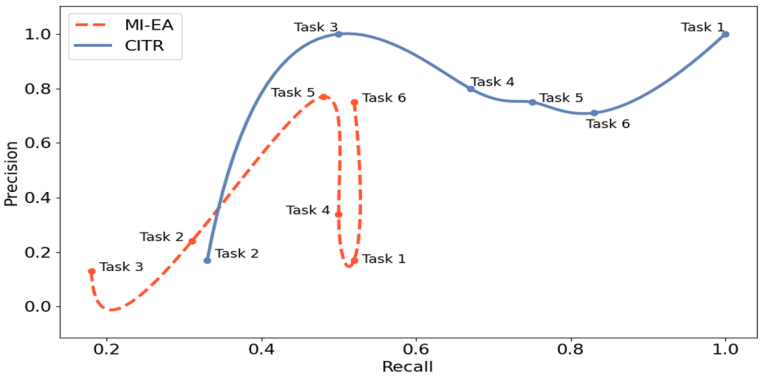


Fig. 11 MI-EA and CITR precision and recall curves

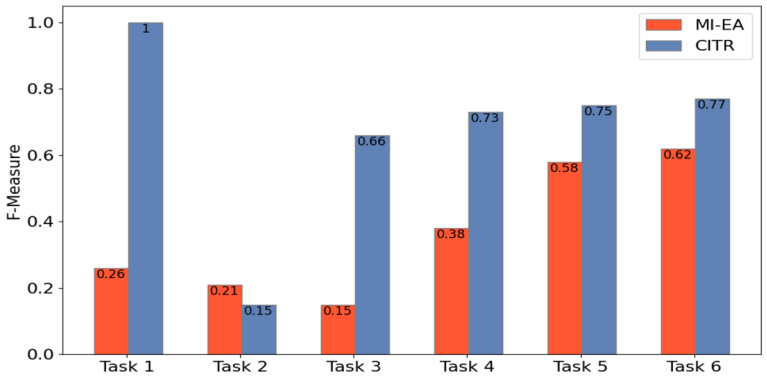


Fig. 12 CITR and MI F-measure values

revealed comparable precision but differing recall. MI demonstrated an average precision–recall of 0.77–0.48 and 0.75–0.52, while CITR exhibited higher recall ratios of 0.75–0.75 and 0.71–0.83, respectively. To gain insight into the similar precision values, we investigated the interaction traces generated by both tasks. Developers who completed both tasks did not produce a significant number of events and did not interact with a large number of files, possibly because of the complexity level of these tasks may have been within the developers' grasp, leading to successful completion with minimal navigation. When the size of data is low, *i.e.,* less interaction traces, there is a greater likelihood of relevant files being present in the recommendations, resulting in high precision and recall. Although both approaches demonstrated strong precision, CITR produced more relevant recommendations, with greater recall values.

Through our tools analysis in Section 7.3, we noticed that 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, CITR does not build recommendations based on different contexts. 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 explains why CITR 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. CITR guides the navigation of newcomers with no prior knowledge of the current system. Hence, it helps newcomers understand and complete the tasks using the recommended files, and not relying on random navigation.

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

9 Threats to validity

Change tasks To avoid the authors' bias of judgement, we hired an external experienced evaluator to classify the difficulty of the candidate change tasks and determine the required completion time of each change task. We chose moderate and difficult complexity tasks that require a maximum of 45 minutes to complete. We avoided selecting complex tasks because they require more time from participants and may result in interruptions or participants dropping out of 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 six different change tasks from a large open source system in the design of the ITs collection (in Section 6.2), completed by participants with various educational and industrial backgrounds, and using a common IDE and language, Eclipse and Java.

Mylyn noise This threat is related to the tool used to collect participants' events, Mylyn Eclipse plug-in. 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.

Generalisability External threats pertain to the possibility to generalise our results. In the study, we generated six recommendations (consensus task interaction trace) from six input change tasks by applying the consensus algorithm to an input of four to eight participants' interaction traces for each task. We had a challenging time recruiting participants to perform the change tasks and thus collect their ITs due to the COVID-19 pandemic. Although there is no recommended number of input developers' ITs from each task in order to generate a recommendation, we intend to expand the study to generate recommendations from a larger number of input ITs to evaluate whether the number of input ITs could potentially affect the outcome of the recommendations. Additionally, due to the small number of participants, ITs were collected from participants performing the same task rather than similar tasks on various software instances. That is to help eliminate the threat of generating heterogeneous ITs. Having a high number of participants in the future study should allow us to consider incorporating similar change tasks performed on different software instances.

Remote experiment Due to the COVID-19 pandemic, we had to change the observational controlled experiment (in Section 7.2) 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[8]. To increase the reliability of our results, we employed multiple measures:

precision, recall, and F-measure for quantitative evaluation; an observational controlled 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 controlled experiment (in Section 7.2) on observation and interpretation of video recordings of developers performing some evaluation change tasks. We could have been biassed 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.

10 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 interaction trace recommender, CITR, that is based on task-related developers' interaction traces collected from resolved change tasks. CITR builds consensus recommendations by applying the consensus algorithm to the set of developers' interaction traces. The approach can recommend relevant file(s)-to-edit to help developers, particularly newcomers, to complete change tasks that are similar to the input tasks with minimal effort and time.

We evaluated our approach using a series of three evaluations: quantitative, qualitative, and comparison. In the quantitative evaluation, we measured the accuracy of the recommendations against ground truth. Measures showed that CITR can recommend accurate and relevant file(s)-to-edit with average precision of 74%, recall of 68%, and F-measure of 68%.

In the qualitative evaluation, we carried out an observational controlled experiment to measure the extent to which recommendations could increase developers' productivity. Results demonstrated that CITR 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. 2014). Results showed that CITR can achieve higher recommendation accuracy and relevancy than that of MI with average F-measure value of 68% and 37% respectively.

We concluded that CITR can guide developers' navigation path towards resolving tasks and thus increase their productivity by recommending relevant file(s)-to-edit, and that the consensus algorithm is an efficient tool to compute such recommendations.

In the future, we plan (1) to broaden the study by carrying it out on an industry-tailored software system, on multiple client instances of the software, while involving real developers

in collecting their interactions with the system and inviting them to evaluate the tool; (2) to perform the qualitative evaluation experiment on two sets of real-industry developers, newcomers and experienced; (3) to involve a large number of developers completing a large number of similar change tasks in order to investigate the results of the consensus algorithm on a larger set of task-related interaction traces; (4) to investigate the possibility of optimizing the quality of Mylyn collected events to reduce noise and atomising the pre-processing step; (5) to enhance developers' navigation experience by developing an IDE plug-in 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; (6) to search and develop a ranking algorithm that can rank the recommended files according to their relevance to the task at hand; and, (7) to apply our approach at the method level to investigate whether the approach could recommend method(s)-to-edit.

Data Availability The datasets generated during and analysed for this study are available in the replication package: https://www.ptidej.net/downloads/replications/emse22a/.

Declarations

Funding and–or Conflicts of Interests/Competing Interests. The authors declare that they have no known competing interests or personal relationships that could have (appeared to) influenced the work reported in this article.

References

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

Ali A, Meilă M (2012) Experiments with kemeny ranking: What works when? Math Soc Sci 64(1):28–40. https://doi.org/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–https://doi.org/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. https://doi.org/10.1007/978-3-642-45135-5_10

Bao L, Xing Z, Xia X, Lo D, Li S (2017) Who will leave the company?: a large-scale industry study of developer turnover by mining monthly work report. In: 2017 IEEE/ACM 14th international conference on mining software repositories (MSR), IEEE, pp 170–181

Biegel B, Baltes S, Scarpellini I, Diehl S (2015) Code basket: Making developers' mental model visible and explorable. In: 2015 IEEE/ACM 2nd international workshop on context for software development, IEEE, pp 20–24

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):1202–1212. https://doi.org/10.14778/2809974.2809982

Chattopadhyay S, Nelson N, Gonzalez YR, Leon AA, Pandita R, Sarma A (2019) Latent patterns in activities: A field study of how developers manage context. In: 2019 IEEE/ACM 41st international conference on software engineering (ICSE), IEEE, pp 373–383

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

CRM (2022) 17 crm statistics: Growth, revenue, adoption rates & more facts. https://crm.org/crmland/crm-statistics

Cubranic D, Murphy G (2003) Hipikat: recommending pertinent software development artifacts. In: 25th International conference on software engineering, 2003. Proceedings., pp 408–41. https://doi.org/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 241–24. https://doi.org/10.1109/VLHCC.2005.32

Erlebacher A (1977) Design and analysis of experiments contrasting the within-and between-subjects manipulation of the independent variable. Psychol Bull 84(2):21. https://doi.org/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, pp 47–58,https://doi.org/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, pp 7–18. https://doi.org/10.1145/2635868.2635905

Hammouda I, Lundell B, Mikkonen T, Scacchi W (2012) Open Source Systems: Long-Term Sustainability. Springer

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. https://doi.org/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. https://doi.org/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, pp 492–50. https://doi.org/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 536–53. https://doi.org/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. https://doi.org/10.1016/j.jss.2013.03.103

Lee S, Kang S, Kim S, Staats M (2014) The impact of view histories on edit recommendations. IEEE Trans Softw Eng 41(3):314–330. https://doi.org/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 70–7. https://doi.org/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 276–28. https://doi.org/10.1109/QSIC.2014.11

Oracle (2022) 60 critical erp statistics: 2022 market trends, data and analysis. https://www.netsuite.com/portal/resource/articles/erp/erp-statistics.shtml

Parnin C, Rugaber S (2009) Resumption strategies for interrupted programming tasks. In: 2009 IEEE 17th international conference on program comprehension, pp 80–8. https://doi.org/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, pp 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 '1. https://doi.org/10.1145/2957792.2957810

Robbes R, Lanza M (2010) Improving code completion with program history. Autom Softw Eng 17(2):181–212. https://doi.org/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. https://doi.org/10.1109/MSR.2010.5463278

Robillard M, Coelho W, Murphy G (2004) How effective developers investigate source code: an exploratory study. IEEE Trans Softw Eng 30(12):889–90. https://doi.org/10.1109/TSE.2004.101
Robillard MP (2008) Topology analysis of software dependencies. ACM Trans Softw Eng Methodol (TOSEM) 17(4):1–36
Robillard MP, Dagenais B (2010) Recommending change clusters to support software investigation: an empirical study. J Softw Maint Evol Res Pract 22(3):143–164. 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,https://doi.org/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 251–26.https://doi.org/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, pp 23–3. https://doi.org/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 325–33. https://doi.org/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 152–16. https://doi.org/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. https://doi.org/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. Empir Softw Eng 23(2):645–692. https://doi.org/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 157–16. https://doi.org/10.1109/ICSM.2009.5306335
Teitelman W, Masinter L (1981) The interlisp programming environment. Computer 14(4):25–3. https://doi.org/10.1109/C-M.1981.220410
Wan Z, Murphy GC, Xia X (2020) Predicting code context models for software development tasks. In: 2020 35th IEEE/ACM international conference on automated software engineering (ASE), IEEE, pp 809–820
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 213–22https://doi.org/10.1109/ICSM.2011.6080788
Ying A, Murphy G, Ng R, Chu-Carroll M (2004) Predicting source code changes by mining change history. IEEE Trans Softw Eng 30(9):574–58. https://doi.org/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 31–4. https://doi.org/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 563–57. https://doi.org/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 135–14. https://doi.org/10.1109/ICPC.2007.18

Authors and Affiliations

Layan Etaiwi[1] · Pascal Sager[2,3] · Yann-Gaël Guéhéneuc[4] · Sylvie Hamel[5]

✉ Layan Etaiwi
mashael.etaiwi@polymtl.ca

Pascal Sager
pascaljosef.sager@uzh.ch ; sage@zhaw.ch

Yann-Gaël Guéhéneuc
yann-gael.gueheneuc@concordia.ca

Sylvie Hamel
hamelsyl@iro.umontreal.ca

1 Polytechnique Montréal, Montréal, Canada

2 University of Zurich, Zurich, Switzerland

3 Zurich University of Applied Sciences, Winterthur, Switzerland

4 Concordia University, Montréal, Canada

5 Université de Montréal, Montréal, Canada