POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Consensus-Based Recommendation Technique for Software Engineering Applications

LAYAN ETAIWI
Département de Génie informatique et génie logiciel

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Génie informatique

Avril 2023

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Cette thèse intitulée :

Consensus-Based Recommendation Technique for Software Engineering Applications

présentée par **Layan ETAIWI**
en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de :

Daniel ALOISE, président
Sébastien LE DIGABEL, représentant du directeur des É.S.
Foutse KHOMH, codirecteur de recherche
Yann-Gaël GUÉHÉNEUC, codirecteur de recherche
Sylvie HAMEL, codirectrice de recherche
Jinghui CHEN, membre
Gabriele BAVOTA, membre externe

DEDICATION

To my father's soul, the late Faisal Etaiwi,
and my beloved mother, Safa Almasri,
both of whom supported me unconditionally
and taught me to be independent and determined.
Dad, you could not witness my success
because death and cancer defeated you...

A quote from Albert Schweitzer:
At times our own light goes out and is
rekindled by a spark from another person.
Each of us has cause to think with deep
gratitude of those who have lighted
the flame within us.
My parents have been that spark for me
when my light blew out.

ACKNOWLEDGEMENTS

It has been a truly life-changing experience for me to pursue this PhD, and I could not have done it without the support I received from so many people.

First and foremost, a big debt of gratitude is owed to my supervisor, Dr. Yann-Gaël Guéhéneuc, whose support and encouragement have been invaluable throughout my Ph.D. journey. I would like to thank him for believing in my potential, his patience, and always willing to assist in any way he could. Further thanks are owed to my co-supervisor, Dr. Sylvie Hamel, whose insight and extensive knowledge into the subject of bioinformatics steered me through this research. Her enthusiasm for the projects was really influential in shaping my research and professional development. I am grateful to both of them for offering the perfect mix of freedom, guidance, and feedback. I would like to say a special thank you to Dr. Foutse Khomh for accepting my supervision at Polytechnique Montréal

I wish to sincerely thank the members of my examination committee: Dr. Daniel Aloise, Dr. Sébastien Le Digabel, Dr. Jinghui Chen, and Dr. Gabriele Banota, who took patience to read, evaluate, and provide valuable comments on my dissertation.

My utmost gratitude and appreciation is extended to the Swiss Confederation for awarding me the Swiss Government Excellence Scholarship that is funded by the Federal Commission for Scholarships for Foreign Students (FCS). I am grateful to them for hosting me as a research scholar at Zurich University (UZH) and Zurich University of Applied Sciences (ZHAW), as well as for providing me with opportunities for learning, networking, and professional and research development.

My appreciation also goes out to Pascal Sager and Gabriel Ullmann for their energy, collaboration, and help throughout my research projects.

If not for the numerous developers who participated in the experiments and shared their experiences and insights through surveys and interviews, the content of this dissertation would have been limited to questionable speculation.

Heartfelt thanks and gratitude go out to my parents, siblings and cousin, Hazem, for their love, encouragement, and unwavering support they have shown me throughout this challenging period.

Finally, a special thank you to my friends Aleksandra and Olga, who have always believed in me, given me the strength to keep going through hard and good times, and put up with my stress and whining over the course of the past few years of study.

RÉSUMÉ

L'ingénierie logicielle est un domaine intensif en connaissances. Les activités quotidiennes des développeurs produisent une grande quantité de données, telles que du code source, des historiques de modifications, des traces d'interactions, *etc.* Cela crée une richesse de données logicielles, mais rend difficile l'analyse et l'extraction de données bénéfiques pour répondre aux besoins des développeurs dans le but d'accomplir des tâches spécifiques. Afin de faciliter l'analyse des données logicielles et l'identification d'informations pertinentes pour une tâche donnée, les chercheurs en génie logiciel développent des systèmes de recommandation, ou RSSE (Recommender Systems for Software Engineering), qui soutiennent le développement et la maintenance logicielle, ainsi que l'amélioration de la qualité des processus de prise de décision. Un système de recommandation pour le génie logiciel est une application qui fournit des éléments jugés précieux pour une tâche en génie logiciel donnée dans un contexte spécifique.

Depuis le début des années 1990, différentes techniques de recommandation ont été proposées. Plus précisément, le premier système de recommandation a été introduit en 1992. Goldberg *et al.* ont créé Tapestry [1], un système de recommandation par courrier électronique qui recommande une liste de diffusion aux utilisateurs en fonction de leurs centres d'intérêt. Dans le domaine de l'ingénierie logicielle, une large gamme de RSSEs ont été développés sur la base de différentes techniques de recommandation pour soutenir les activités en génie logiciel. Bien que ces techniques puissent être efficaces, des études ont montré qu'elles peuvent présenter certaines limites. En effet, certaines de ces techniques ne sont pas en mesure de générer des recommandations sans une connaissance suffisante et des métadonnées sur les éléments. De plus, d'autres techniques, telles que les règles d'association, nécessitent une contribution de l'utilisateur avant de faire des recommandations. Bien que des techniques telles que l'extraction de données puissent être efficaces, elles sont coûteuses en termes de calcul, car elles nécessitent des ensembles de données énormes pour fournir des recommandations hautement précises. De plus, à notre connaissance, aucun de ces systèmes de recommandation pour l'ingénierie logicielle n'a été appliqué, testé et évalué sur une variété de types de données et d'applications logicielles pour démontrer leur généralisabilité. Il est donc important de fournir une technique de recommandation capable de produire des recommandations sans nécessiter de vastes ensembles de données, et ayant été prouvée comme suffisamment généralisable pour traiter différents types de données pour différentes applications logicielles.

Dans cette thèse, nous utilisons la technique de classement consensus qui trouve un classement

consensus en agrégeant un ensemble de classements différents pour générer un classement qui est le plus proche de tous les classements d'entrée. Par conséquent, nous proposons une technique de recommandation basée sur le consensus qui construit des recommandations en appliquant un algorithme de consensus. La technique de consensus recommande des éléments sous la forme d'un classement consensus et peut produire des recommandations pertinentes sans avoir besoin d'un grand volume de données. Contrairement aux techniques de recommandation existantes, la technique proposée devrait démontrer sa capacité à générer des recommandations en utilisant différents types de données d'entrée pour résoudre différents problèmes liés au génie logiciel. Nous pensons que la technique de recommandation basée sur le consensus proposée dans cette thèse permettra aux développeurs de trouver des informations pertinentes, d'accomplir leurs tâches et d'améliorer leur productivité.

Pour valider l'efficacité et l'applicabilité de la technique, nous étudions et évaluons la recommandation basée sur le consensus dans trois domaines du génie logiciel. Dans le premier domaine d'étude, nous visons à guider les développeurs d'applications mobiles dans la planification de la prochaine version de leurs applications. Pour ce faire, nous extrayons, prétraitons, catégorisons et regroupons les avis des utilisateurs de quatre applications mobiles. Nous priorisons et recommandons ensuite l'ensemble regroupé d'avis d'utilisateurs en appliquant notre RSSE proposé. Dans le deuxième domaine d'étude, nous examinons la technique de consensus dans le contexte de la navigation des développeurs de logiciels. L'objectif de cette étude est de guider les développeurs dans l'exécution de tâches de maintenance et de développement sur des systèmes logiciels personnalisés. Avec cette technique, nous mettons en place un système de recommandation qui propose aux développeurs une trace d'interaction de tâche consensue composée de fichier(s) à modifier lié(s) à la tâche. Dans le dernier domaine, l'accent est mis sur la prise en charge de la conception d'architecture d'un moteur de jeu. Nous utilisons la recommandation basée sur le consensus pour étudier son applicabilité aux données d'architecture du moteur de jeu, et son succès à fournir un classement consensus des sous-systèmes fondamentaux de l'architecture de moteur qui peuvent aider les développeurs tout au long du processus de conception de l'architecture. L'objectif final de toutes ces études est de nous aider à évaluer l'applicabilité de la technique de recommandation basée sur le consensus proposée à plusieurs types d'ensembles de données et à la prise en charge de diverses tâches de génie logiciel.

Nos recherches futures prennent trois directions distinctes. Tout d'abord, nous prévoyons de mener une étude de revue de littérature qui fournit une connaissance complète sur les étapes de construction d'un système de recommandation pour l'ingénierie logicielle et qui peut être considérée comme une bible pour les développeurs novices lorsqu'ils développent des SR. Ensuite, nous proposons une technique de regroupement des avis d'utilisateurs basée

sur l'apprentissage en profondeur. Cette technique fera partie d'une approche tout-en-un qui peut automatiquement nettoyer et prétraiter les avis, catégoriser, regrouper et recommander un classement prioritaire des avis d'utilisateurs. Enfin, nous avons l'intention d'étendre nos études de recherche de manière générale en impliquant l'industrie et de vrais développeurs, en utilisant de vraies données de l'industrie, en augmentant la taille de l'ensemble de données, en automatisant le prétraitement des données et en mettant à disposition un système de recommandation basé sur un consensus complet pour tout type d'applications logicielles.

ABSTRACT

Software engineering is a knowledge-intensive domain. Daily developers' engineering activities produce a large volume of data, such as source-code, change history, interaction traces, *etc.* This creates software data richness, but it makes it difficult to analyse and extract beneficial data to support developers' needs to accomplish specific tasks. As a means of facilitating the analysis of software data and the identification of relevant information for a given task, software engineering researchers develop recommendation systems for software engineering RSSEs that support software development and maintenance, as well as improve the quality of decision-making processes. A Recommendation system for software engineering (RSSE) is a software application that recommends items that are deemed valuable for a given software engineering task within a specific context.

Various recommendation techniques have been proposed since the early 1990s. Specifically, the first recommendation system was introduced in 1992. Goldberg *et al.* built Tapestry [1], a mailing recommendation system that recommends a mailing list to users based on their interests. In the software engineering domain, a wide range of recommendation systems for software engineering (RSSEs) have been developed based on various recommendation techniques to support software engineering activities. While these techniques can be effective, studies demonstrated that they can present some limitations. Some of these techniques are unable to generate recommendations without sufficient knowledge and metadata about the items. Furthermore, others, such as association rules, require user input before making recommendations. Although techniques such as data mining can be efficient, they are computationally expensive due to the fact that they require enormous datasets in order to provide highly accurate recommendations. In addition, to the best of our knowledge, none of these software engineering recommendation systems were applied, tested and evaluated across a variety of data types and applications to demonstrate their generalisability. Thus, it is important to provide a recommendation technique that can generate recommendations without requiring large datasets and can run on different types of data for different software applications, demonstrating generalisability.

In this thesis, we make use of the consensus ranking technique which finds a consensus ranking by aggregating a set of different rankings to generate one ranking that is the closest to all of the input rankings. Hence, we propose a consensus-based recommendation technique that builds recommendations by applying a consensus algorithm. The consensus technique recommends items in the form of a consensus ranking and can produce relevant recommendations

without the need for a large volume of data. Unlike existing recommendation techniques, the proposed technique is expected to demonstrate its capacity to generate recommendations using different input data types to address different software engineering-related issues. We believe that the proposed consensus-based recommendation technique in this study will support developers finding relevant information, carry out their tasks, and enhance their productivity.

To validate the effectiveness and applicability of the technique, we investigate and assess the consensus-based recommendation in three software engineering areas. In the first area of the study, we aim to guide mobile app developers through the planning of their apps' next release. To accomplish this, we extract, preprocess, categorise and cluster user reviews of four mobile applications. We then prioritise and recommend the clustered set of user reviews by applying our proposed technique. We also examine the consensus-based technique in the context of software developer navigation. The purpose in this study is to guide developers through performing maintenance and development tasks on instances of customised software systems. Using the technique, we implement a recommendation system that recommends to developers a consensus task interaction trace consisting of task-related file(s)-to-edit. In the last area, the focus is placed on supporting game engine architecture design. We employ the consensus-based recommendation to investigate its applicability to game engine architecture data, and its success at providing a consensus ranking of fundamental engine architecture subsystems that can assist developers through the architecture design process. The end objective of these studies should be to help us assess the applicability of the proposed consensus-based recommendation technique to several types of dataset and to supporting various software engineering tasks.

Our future research takes three distinct directions. First, we plan to conduct a literature review study that provides a comprehensive knowledge about the steps of building a recommendation system for software engineering and that can be regarded as a Bible for novice developers when developing RSs. Next, we propose a user review clustering technique that is based on deep learning. This technique will be part of an all-in-one approach that can automatically clean and preprocess reviews, categorise, cluster, and recommend a prioritised ranking of user reviews. Lastly, we intend to expand our research studies in general by involving the industry and real developers, using industry data, increasing the size of datasets, automating data preprocessing and making available a complete consensus-based recommendation system for any type of software applications.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

RSSEs	Recommendation Systems for Software Engineering
RSs	Recommendation Systems
LP	Linear Programming
CF	Collaborative Filtering
CBF	Content-Based Filtering
KBR	Knowledge-Based Recommendation
RMSE	Root Mean Squared Error
MAE	Mean Absolute Error
RP	Review Prioritiser
GR	Gold Ranking
TSITs	Task-related Set of Interaction Traces
IT	Interaction Trace
CITR	Consensus Task Interaction Trace Recommender
MI	Mining programmer Interaction histories
COSA	COnsensus Software Architecture

CHAPTER 1 INTRODUCTION

1.1 Research Context

Software development requires software developers to engage in a wide range of tasks, including designing, implementing, testing, debugging, and many others. These tasks create a large amount of data with which developers interact, such as source-code, design decisions, commit logs, interaction traces, etc. The immense growth in software engineering data creates a challenge for software developers to find specific information that meet their needs, solve engineering related issues, and support ongoing development tasks. For example, if developers need to discover information that helps them implement features or decide what new features they should implement next. Using information retrieval techniques help developers in navigating the vast data spaces. However, there is still a pressing need for recommendation systems that reduce data overload, deliver the most pertinent data and ease the performance of activities.

Recommendation system for software engineering (RSSE) is defined as a software application that provides information items estimated to be valuable for a software engineering task in a given context [3]. Recommendation systems play a significant role in software development by reducing information overload, recommending the most relevant information, and improving decision making process and quality. A typical recommendation system consists of three main components [4]: (1) a data collection method, (2) a recommendation technique, and (3) an interface to present recommendations. Three basic recommendation approaches have emerged; collaborative filtering, content-based filtering and knowledge-based recommendation [5]. Recommendation systems in software engineering can produce a variety of recommendations for different purposes. For example, Rose [6] recommends source-code elements to be changed based on mining program's version history, DPR [7] suggests design patterns based on Goal-Question-Metric (GQM) to help designers with software design problems, A LA [8] helps software developers find documents and metadata that related to their tasks, and AR-Miner [9] prioritises mobile apps reviews based on a ranking model to help developers plan their apps' next release.

Research in the area of recommendation systems development began to emerge in the early 1990's. Particularly, in 1992, Goldberg *et al.* built Tapestry [1], the first collaborative filtering based mailing recommendation system. Later in 1994, Resnick *et al.* [10] introduced the GroupLens system, a collaborative filtering based system, it uses readers' ratings on some articles to then recommend them to other readers with similar interests. In 1995, a video

recommender that relies exclusively on ratings information was presented by Hill *et al.* [11] to recommend videos to watch. Similarly, the Ringo system [12] provided personalised recommendations from any type of dataset based on social information filtering technique. Since then, this topic has been explored extensively in the context of a wide variety of modes, such as Amazon's item recommendation in the commercial field, and cancer drug prioritisation [13] in bioinformatics. Particularly in the software engineering domain, recommendation systems have attracted a great deal of research attention. Pakdeetrakulwong *et al.* [4] reviewed twenty six recommendation systems for diverse software development life cycle phases and identified benefits and usefulness of these systems. Similarly, Mohebzada *et al.* [14] performed a systematic mapping review that included twenty three studies of recommendation systems for software engineering RSSEs for requirements engineering. These studies and numerous others demonstrate how the field of research on recommendation systems for software engineering RSSEs has grown significantly and is still growing.

Designing and developing RSs is a multi-step process that involves defining a problem that needs to be solved and task to be achieved, determining the input data, preprocessing the data and building the recommender. The type of input data and source depend on the context of the task. Recommendation techniques rely on noise-free datasets. However, in the real-world, collected data is likely to be incomplete, inconsistent and redundant and hence must go through steps of preprocessing. Data preprocessing can include steps like cleaning, transformation, and reduction. In terms of building the core function of the recommender, there is a set of recommendation techniques that can be implemented. The choice of a technique that takes the input data and transforms them into a set of recommendations depends on the concrete recommendation problem.

1.2 Research Statement

The extensive research studies on the three basic recommendation approaches have led to the development and implementation of various algorithms and techniques under each approach. Several recommendation systems for software engineering activities [7, 15–17] have been developed and evaluated under the collaborative filtering approach. It is divided into two categories, memory-based approach and model-based approach [18]. The idea of memory-based approach works by building a dataset of users and their preferences for items and recommends to given users items that other users from the dataset with similar interest prefer. Similarity measures like correlation-based and cosine-based are used to compute the similarity between items/users. A model-based approach builds recommendations using machine learning and data mining algorithms such as Singular Value Decomposition (SVD), Matrix

Completion Technique, Association Rule, or Clustering. Content-based recommendation systems [8, 19–21] base their recommendations on feature extraction from items' content [18]. The approach uses Vector Space techniques such as Term Frequency Inverse Document Frequency (TF/IDF) or Probabilistic Techniques to compute the feature similarity between items, and recommends items that are similar to the ones a user preferred in the past. Unlike the two aforementioned approaches, knowledge-based recommendation systems [22–24] recommends items based on knowledge about the users/ items and their relationships [5].

Despite the success of these techniques, research studies of a wide scope have identified some potential limitations. **Sufficient Knowledge** These techniques require sufficient information about the items/users for the algorithms to generate relevant recommendations. For example, if a new time is added to the dataset, the system will be unable to recommend it unless it is accompanied by significant metadata, such as users ratings. Thus, not having adequate information is one of the major problems that reduces the quality of recommendations. Content-based techniques, for instance, need to have an in-depth knowledge about the features of the items in the dataset before recommendations can be made. Accordingly, the effectiveness of content-based techniques depends on the availability of item knowledge [25]. **Irrelevant Interaction** Techniques, such as association rules, require users' input to the recommender before making recommendations, and may suggest unrelated items if the users interact with the "wrong" item. **Scalability** The majority of recommendation systems, on the other hand, rely on machine learning techniques. Machine learning requires large datasets to train and create models. Maintaining and preprocessing datasets can become challenging, and computation can become costly as a result [5]. **Generalisability** To the best of our knowledge, all available software engineering recommendation systems have been developed, studied, and evaluated on one specific problem, in one specific application, dealing with a single data type. For example, Holmes *et al.* [15] proposed a recommendation system that can provide recommendations about the relevancy and cost of source-code elements that a developer intends to reuse. The authors, however, never evaluated the generalisability of their approach by applying it to a different data type, or employing it to resolve other domain issues, such as recommending design patterns.

To address these limitations, we propose a consensus-based recommendation technique, a collaborative filtering based recommendation, that generates recommendations by applying a consensus algorithm. The proposed technique recommends items in the form of a consensus ranking and is able to provide recommendations to resolve different software engineering-related issues, using different types of datasets. We hypothesise that the proposed technique will assist software developers in finding relevant information, resolving engineering issues, and completing ongoing development and maintenance tasks.

We formulate our thesis statement as follows:

We propose a software engineering recommendation technique based on the consensus algorithm that applies to various data types and resolves various software engineering-related issues in a variety of applications.

1.3 Research Methodology

A recommendation technique must be scalable and is able to generate satisfactory recommendations without the need for a large dataset, constant knowledge about the items, and instant input from the user. We propose a consensus algorithm-based recommendation technique that can handle different types of input data and generate recommendations that can solve any software engineering application problem. The core function of the technique involves taking as input a set of items, measuring the distance between two items using a predefined distance measure, and producing a consensus ranking consisting of a set of recommended items. We briefly describe employing the proposed technique to address software engineering issues in three different applications; mobile applications, software systems, and game engine architecture. Figure 1.1 presents an overview of our research methodology.

Recommending Mobile App Reviews: Studies [9, 26, 27] showed that information in users' mobile app. reviews help developers maintain and develop their apps. As a result, it is crucial for app developers to take user reviews into account when updating their apps. However, the volume of user reviews received for some apps is large and surpasses developers' ability to analyse, process them and extract useful information manually. We solve these challenges by proposing a RSSE that preprocesses, categorises, and clusters user reviews, followed by implementing the consensus algorithm to recommend a consensus ranking of prioritised reviews that developers should consider when planning their apps.' next release.

Recommending Interaction Traces for Software Navigation: Forking a software system into multiple instances is a good practice to accommodate different clients' needs. Customising and maintaining each client's instance is time consuming and challenging for developers, especially newcomers. Some of the change tasks that developers perform on these instances can be the same for each client or very similar. While completing a change task, each developer generates a task interaction trace consisting of source-code elements interacted with for completing the task. To validate the applicability of the proposed technique, we apply the consensus algorithm to a set of collected developers' interaction traces from previously completed tasks to recommend a consensus task interaction trace consisting of file(s)-to-edit

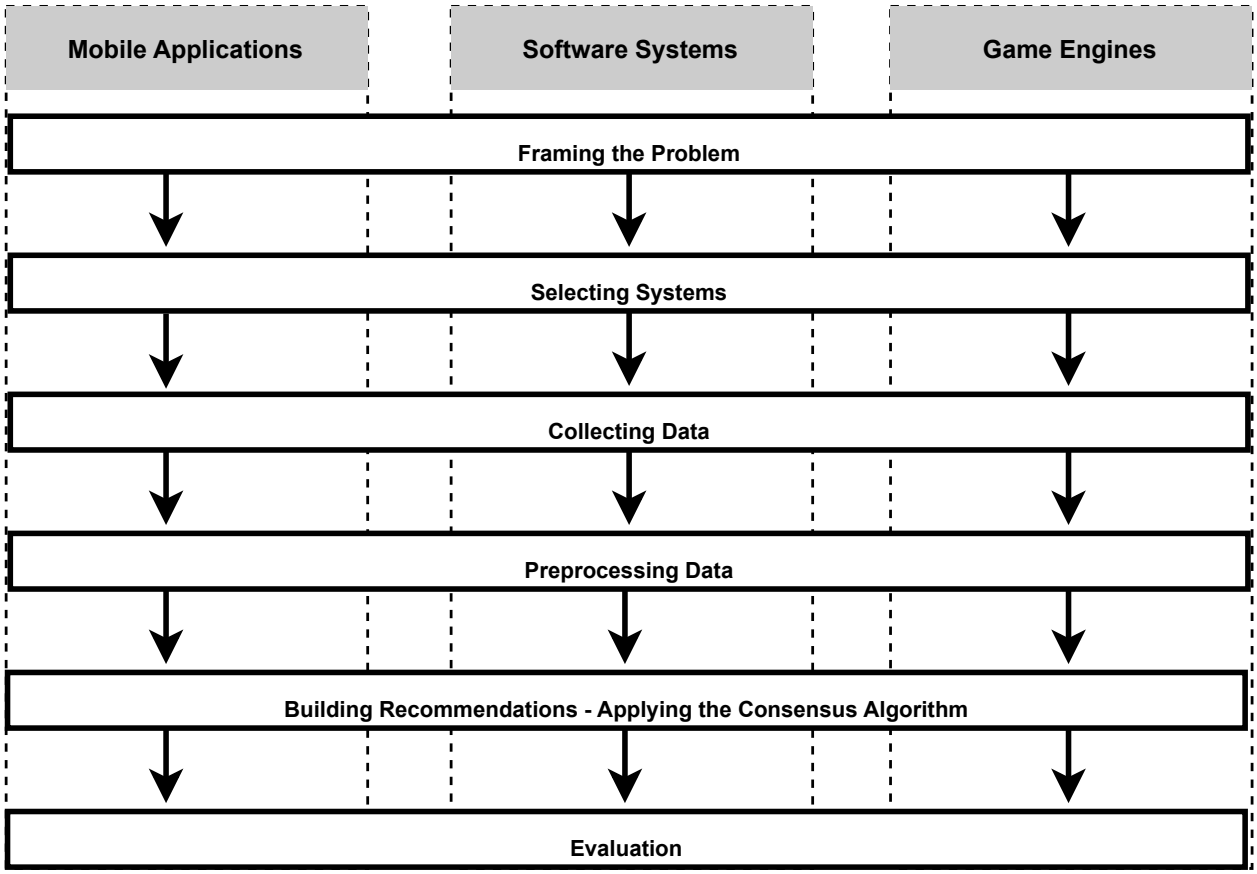


Figure 1.1 Overview of Research Methodology

that can guide developers' navigation towards completing similar tasks on other software instances.

Recommending Game Engine Architecture Subsystems: Growing users' expectations from video games made the process of developing video games much more complex. Due to this complexity, developers have come to rely on frameworks (game engines) to help them with the development process by providing generic, reliable and reusable software subsystems such as a rendering engine, physics engine, audio system, *etc.* Each game engine has a unique architecture, and there is currently no standardised architectural model that can be used when designing engine architecture. We investigate the effectiveness of the proposed recommendation technique in addressing the problem of identifying architecture subsystems when building game engines. We aim to guide developers towards designing a game engine architecture by recommending a model that presents a consensus fundamental engine architecture subsystems ranked by their degree of coupling. The model is based on the application

of the consensus algorithm to a collection of recovered existing game engine architectures.

1.4 Research Contributions

Based on the proposed research methodologies, we make the following contributions.

1.4.1 Chapter 3: Fundamentals in a Nutshell

We divide this chapter into two sections. In the first Section, we discuss the emergence of the consensus rankings from optimal permutations, define key concepts and formulas and explain in-depth how the algorithm produces a consensus ranking. We also go over calculating a Kemeny score and distance measures used to determine the distance between items. Lastly, we discuss methods of handling incomplete rankings and compare some of the consensus algorithms. In the second Section, we provide an overview of the fundamental steps of building a recommendation system for software engineering. This overview can be used as a starting point for recommendation builders with limited knowledge when building RSs. We start by discussing the importance of clearly defining the problems and tasks that the recommendation system is intended to solve, and the possibility of generating valuable recommendations. Furthermore, we present possible input data types and sources, and outline basic data preprocessing techniques and methods. Lastly, we consider some evaluation measures for evaluating the effectiveness of the recommendation system. We constructed this Section based on observations made from the research studies presented in Chapters 4, 5 and 6 as well as abstract review of a few RSSEs studies [28–30]

1.4.2 Chapter 4: Prioritising Mobile App Reviews for Mobile Application Evolution

In this chapter, we build Review Prioritiser (RP); a consensus-based prioritisation technique that prioritises and recommends mobile app. user reviews to assist app developers in deciding what improvements to implement in the next releases of their apps. We implement three validation methods to evaluate the success of the technique and the meaningfulness of the prioritised ranking of reviews on four Android apps. First, we compare the results with manually-prioritised rankings of user reviews by the apps' developers, and then compute the kendall rank correlation coefficient to statistically measure the correlation between them. In the second method, we ask developers to qualitatively evaluate the relevance of the recommendations. In the last method, we send a questionnaire to app developers asking for their thoughts on how likely it is that our recommendations will help them plan the next release.

We find that the consensus rankings and the manually-prioritised rankings have a strong correlation. Using the consensus algorithm to prioritise and recommend user reviews can produce useful rankings, and successfully help developers planning their apps' new release. As a result, our findings are consistent with our research statement, which states that the proposed software engineering recommendation technique based on the consensus algorithm works on mobile app user reviews and can resolve mobile application-related issues. This chapter is based on a published conference paper [31].

1.4.3 Chapter 5: Recommending Task Interaction Trace to Guide Developers' Software Navigation

We propose Consensus Task Interaction Trace Recommender (CITR); a task-based recommendation technique that recommends files-to-edit to guide developers towards completing similar change tasks on other clients' instances. We demonstrate the effectiveness of the technique by conducting a series of evaluations. To assess the results accuracy, we compare them to predefined ground truth data and compute measures such as precision and recall. Additionally, to determine the qualitative success of the technique in supporting developers, we conduct an observational comparative experiment with 30 developers performing change tasks with and without the recommendations. Lastly, we compare CITR against MI (Mining programmer Interaction histories), an existing file-level recommendation approach. We conclude that CITR can generate accurate recommendations, boost developers productivity by allowing them to complete tasks in less than half the time and effort required, and outperform a comparable recommendation technique, MI. The results confirm that our RSSE is applicable to developers' interaction traces data type and has demonstrated success for software navigation. This chapter is based on a submitted journal paper [32].

1.4.4 Chapter 6: Finding Common Game Engine Architecture subsystems and their Coupling Degree

To support the process of creating and maintaining game engines, we propose COnsensus Software Architecture (COSA), an approach based on applying the consensus algorithm to a set of game engine architectures. Our approach generates a model that suggests the most commonly used subsystems in game engine architectures, ranked by their degree of coupling. Based on studying and extracting subsystems of 10 game engine architectures, We show that all recovered subsystems are essential and should be considered when designing a game engine architecture. In addition, we discover that core systems, low-level rendering, third-party SDKs and world editor are the most coupled subsystems. We conclude that the software

engineering recommendation technique we propose is general enough to process game engine subsystem data type and make recommendations in the domain of game engine architecture. This chapter is based on a to be submitted conference paper [33].

1.5 Thesis Organisation

The rest of this thesis is organised as follows. In Chapter 2, we discusses related work in the areas of recommendation systems and consensus algorithms usage. In Chapter 3, we provide an overview of the consensus algorithms and discuss their implementation techniques. In addition, we outline basics steps for building recommendation systems. In Chapter 4, we proposes an approach for prioritising mobile app user reviews. In Chapter 5, we study using the consensus algorithm to recommend file(s)-to-edit that can help developers perform similar change tasks on software systems. In Chapter 6, we presents our study for creating a model for fundamental game engine architecture subsystems. Finally, in Chapter 7, we conclude with some future works while highlighting the success of the consensus algorithm in processing various data types and resolving various software engineering-related issues in a variety of applications.

CHAPTER 2 RELATED WORK

2.1 Recommendation Systems

A story from history: As the internet usage grew rapidly in the early 1990s, recommendation systems were introduced to assist users in filtering out useful information. In 1992, Goldberg *et al.* built Tapestry [1], a collaborative filtering based mailing recommendation system. The system collects users' reactions to documents they read, and filters mails belonging to these documents into lists of interests. Later, the GroupLens system [10] was built in 1994 to help readers find articles of interest from a huge pool of available articles. The system captures readers' ratings on news articles they read, and recommends articles to readers who are like-minded and have similar interests. Similarly, the Ringo system [12] generates personalised music recommendations by applying four different social information filtering algorithms. Ratings assigned by other people with similar tastes are used to recommend music to a user. Based on standard formulas for computing statistical correlations, the system determines which users have similar tastes.

Research on Recommendation Systems: The pressing need for efficient recommendation systems has driven researchers to conduct in-depth studies that explore different RSs characteristics, techniques, evaluation methods and compare RSs approaches. Such studies promise to provide a comprehensive guide to follow when developing a recommendation system. Ricci *et al.* and Robillard *et al.* provided handbooks [30, 34] that are dedicated entirely to recommendations systems. The books study in detail approaches and techniques for building recommender systems, ways of interacting with recommender systems, systems design, implementation and evaluation. To further support research in the development of recommendation systems, [35] conducted a survey of recommendation systems in software engineering in order to identify limitations and discuss possible improvements.

Non-Software Engineering Domains: Recommendation systems have been the subject of numerous research studies and commercial development in a variety of fields since their discovery. In agriculture, for instance, RSs make a significant contribution to the efficient management and use of resources. In [36], the authors used the Apriori model and hybrid filtering approach to build a web-based recommendation system. The system analyses customer purchasing behaviour for vegetables using Apriori, and it recommends to farmers which vegetables to cultivate for the next season based on historical purchases and best-selling vegetables. In the field of e-Commerce, RSs play a major role. Several of the largest commerce websites, such as Amazon and eBay, use recommendation systems to help their

customers find products to purchase. The Wasabi Personal Shopper (WPS) [37], is a domain independent database browsing tool designed for on-line access to electronic products. The system applies a knowledge-based similarity retrieval approach to suggest new items to users. E-learning is another area that has attracted a lot of recommendation system research attention. A personalised e-learning material recommender system (PLRS) was proposed in the work of Lu [38]. The framework utilises a multi-attribute evaluation method and a fuzzy matching method to recommend learning materials to students that meet their needs.

Software Engineering Recommendation Systems: Over the past several years, recommendation systems for software engineering (RSSEs) have become an active area of research. They have proven to help software engineers with data overload and software development activities. To recommend to developers artefacts that are relevant to change tasks, [39] proposed Hipikat. The tool forms a group memory using source-code versions, bugs, electronic communication, and web documents. It then recommends relevant artefacts based on inferring links between the archived artefacts in the group memory. DebugAdvisor [19] on the other hand is a recommendation system that helps improve productivity of debugging by automating the search for similar issues from the past. The recommender uses a fat query interface to store and allow developers to search through all software repositories, and link-analysis algorithms to compute similarity and retrieve relevant debug issues. Another example of a recommender for software engineering activities is DPR [7]. DPR (Design Pattern Reocmmender), an automated system for recommending design patterns for developers to use for a particular design problem. The recommender produces recommendations based on a simple Goal-Question-Metric (GQM) and weight functions.

2.2 Consensus Ranking Applications

In the early twentieth century, researchers began to investigate the problem of aggregating multiple rankings into one consensus ranking, and it has been studied in numerous applications. We present here several applications that employ rank aggregation.

Elections The problem of designing voting rules for an assembly was a long-standing problem in the theory of elections during the second half of the eighteenth century. To deal with the problem, [40] proposed the Kemeny-Young method that is able to provide a consensus order of a list of candidates and combine the qualities of two previously proposed methods; Borda and Condorcet. Similarly, [41] investigated two similar schemes (the Dodgson and Kemeny schemes) to find a winner in an election. For both schemes, the study showed that the problem of finding a winner in an election is NP-hard.

Bioinformatics An important application of the consensus ranking in bioinformatics involves the classification of disease-related proteins/genes and the construction of genetic maps. In [42], the authors study gene expression patterns (calculating the probability of occurrences of each gene in given tissues). Several techniques have been developed to measure gene expression, such as microarrays, SAGE, BodyMap and MPSS. However, not every technique was able to completely cover all the expression levels of human genes. They presented a ranking system, mRNAs, that aggregates the results from all existing techniques into one consensus ranking. The consensus ranking has also been studied to address issues in biological data queries. [43] addressed the issue of scientists not receiving ranked sets of answers to their biological queries by proposing a consensus algorithm based on the concept of the median. Moreover, [44] investigated the use of rank aggregation to find genes related to prostate cancer. Many microarray studies provided different ranked lists of genes associated with particular diseases. To solve the significant variations, the authors proposed a rank-aggregation approach for combining results from several microarray studies.

Social Sciences [45] tested and evaluated several rank aggregation methods. The authors conducted an empirical study where participants are asked to rank the occurrence of events in populations, geography, or history. A consensus ranking was then generated from applying the rank aggregation methods. [46] is a similar study in the area of social sciences. The study performed a systematic comparison of Kemeny rank algorithms to generate rankings from a batch of social choice datasets.

Other Applications In environmental sciences, traffic noise was studied in [47] to identify road stretch priority variables that contribute to the noise and their degree of importance. Identified variables were then ranked by degree of relevance by high-level experts. Several rank aggregation methods were applied to prioritise road stretch variables and generate a single decision. Likewise, [48] presented a preference aggregation algorithm; it aggregates users' preferences in order to suggest the best items for the group as a whole. The algorithm is based on a novel local-search algorithm for aggregating users' preferences into a single consensus ordering. The idea of formulating a consensus ranking has been also investigated in the context of the Web. [49] studied several heuristic rank aggregation algorithms to obtain a consensus ranking of the result from various search engines. In spite of the fact that rank aggregation methods are not used in the practice of sports, researchers applied them to sport data for decision-making purposes. [50] uses the Kemeny ranking method for ranking couples of dancers in competitions. The paper also suggests modifying the Kemeny method to adapt ranking with possible ties.

CHAPTER 3 FUNDAMENTALS IN A NUTSHELL

3.1 Background on the Consensus Rankings

Finding a consensus ranking is defined as aggregating a set of N different rankings of n items into one ranking that orders the n items closest to all of the N rankings within a specified distance [46]

Example 3.1 *A group of three friends plan to have dinner and each individual ranks five cuisines, [Arabic (1), French (2), Indian (3), Korean (4), Vietnamese (5)], options based on their own personal preferences. They suggest using the concept of rank aggregation to find an optimal order:*

$$\left.\begin{aligned} R_1 &= [3,1,5,4,2]] \\ R_2 &= [5,1,2,3,4] \\ R_3 &= [1,4,2,3,5] \end{aligned}\right\} R^* = [1,3,5,4,2]$$

3.1.1 Definitions and Measure on Permutations

In its original formulation, the rank aggregation problem was introduced first for a set of permutations.

Definition 3.1.1 *A permutation π is a bijection of [n] items onto itself [51]. It represents a strict total order of the items of [n].*

A permutation π of {1,2, . . . , n} is denoted as $\pi = \pi_1\pi_2...\pi_n$, where π_j denotes the item at position j and π_i^{-1} denotes the position of item i.

Example 3.1.1 *The permutation $\pi = [1,3,5,4,2]$ is of size $n = 5$. Position 5 represents item "2" and denoted as $\pi_5 = 2$. In permutation π, we say item "4" precedes item "2", and symbolised as $4 \prec_\pi 2$ (or $4 \prec 2$).*

Kendall-τ Distance, Kemeny Score, and Optimal Permutation

As a means of finding a consensus across a set S of permutations, a classical Kendall-τ distance is used to measure the distance between two permutations, and a Kemeny score is calculated to define an optimal permutation [51].

Definition 3.1.2 *The Kendall-τ distance takes two permutations (π, σ) from a set S, and counts the number of pairs of items whose order differs between the two permutations. Arithmetically, Kendall-τ distance D between two permutations π and $\sigma \in S_n$ is defined as:*

$$D(\pi, \sigma) = |\{(i, j) : i < j \wedge$$
$$(\pi[i] < \pi[j] \wedge \sigma[i] > \sigma[j] \ \vee \ \pi[i] > \pi[j] \ \wedge \ \sigma[i] < \sigma[j])\}|$$

Example 3.1.2 *The Kendall-τ distance between R_1 and R_2 from Example 3.1. The underlined pairs represent the difference in order between the two permutations, which equates to a distance of* 6*.*

$R_1 = [3,\ 1,\ 5,\ 4,\ 2]$
$R_2 = [5,\ 1,\ 2,\ 3,\ 4]$

$\underline{(1,2)}$	$\underline{(2,3)}$	(3,4)	(4,5)
$\underline{(1,3)}$	$\underline{(2,4)}$	$\underline{(3,5)}$	
(1,4)	(2,5)		
$\underline{(1,5)}$			

The Kemeny score is used to evaluate how far a permutation is from a set of permutations. It measures the sum of the Kendall-τ distances between a permutation π and every permutation in the set S. Finally, the optimal Kemeny score is used to define one or more optimal permutation(s) that with a minimal distance [51].

Definition 3.1.3 *The Kemeny score K between a permutation π and a set of permutations $\mathcal{P} \subseteq S_n$, all on the same set of items, is denoted as follows:*

$$K(\pi, \mathcal{P}) = \sum_{\sigma \in \mathcal{P}} D(\pi, \sigma).$$

Definition 3.1.4 *An optimal permutation π^* of a set of permutations $\mathcal{P} \subseteq S_n$ that minimises the kemeny score is given as follows:*

$$\forall \pi \in S_n : K(\pi^*, \mathcal{P}) \leq K(\pi, \mathcal{P})$$

Example 3.1.3 *From the permutations and optimal permutation in Example 3.1, where $R_1 = [3, 1, 5, 4, 2]$, $R_2 = [5, 1, 2, 3, 4]$, $R_3 = [1, 4, 2, 3, 5]$ and $R^* = [1, 3, 5, 4, 2]$. The Kemeny score of R^* is the result of the pairwise disagreements between R^* and the set of permutations: $(1, 3)$ in $R_1, (1, 5), (2, 3), (2, 4), (3, 5)$ in R_2, and $(2, 3), (2, 5), (3, 4), (4, 5)$ in R_3. Thus $K(\pi^*, \mathcal{P})$ is $1 + 4 + 4 = 9$.*

3.1.2 Definitions and Measure on Rankings

In real life applications, rankings can be incomplete. This happens when not all the n items 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. Rankings can also be not strictly-ordered, when some items are ranked at the same position, *i.e.,* items are tied (with equal rank). For example, in elections, a voter could have chosen to put more than one candidate in the first position. To deal with incomplete rankings and rankings with ties, a generalisation of rank aggregation problem has been introduced.

Definition 3.1.5 *Let U be a universe of items. A Ranking R is an ordered set of items $[n]$. It is presumed that the ranking is complete if $n = U$, otherwise it is considered incomplete. A ranking can also be a bucket order on $[n]$, in which case it is referred to as ranking with ties. A ranking with ties is defined as $R = [\mathcal{B}_1, ...\mathcal{B}_k]$, where $\mathcal{B}_i$ is a bucket at position i. We denote $R[x] = i$ if $x \in \mathcal{B}_i$ [51].*

Example 3.1.4 *Let $U = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, and $R = [[5, 6, 8], [9], [1, 3], [2]]$. Since item 7 is missing from the ranking, it is considered incomplete. At position 1 is bucket $\mathcal{B}_1$ which contains items $5, 6$ and 8. The position of item 2 is $R[2] = 4$. In the ranking, item 1 is equal to item 3, i.e., $1 \equiv 3$, since both items are positioned in the same bucket.*

Generalised Kendall-τ Distance, Generalised Kemeny Score, and Consensus Ranking

To measure the distance between two incomplete rankings with ties, a generalised version of the Kendall-τ distance has been introduced [43, 52].

Definition 3.1.6 *The generalised Kendall-τ distance, G, between two rankings R and C, is:*

$$\begin{aligned} G(R, C) = \#\{(i, j) : i < j \wedge \\ ((R[i] < R[j] \wedge C[i] > C[j]) \vee (R[i] > R[j] \ \wedge \ C[i] < C[j]) \vee \textbf{\textit{(1)}} \\ (R[i] \neq R[j] \wedge C[i] = C[j]) \vee (R[i] = R[j] \ \wedge \ C[i] \neq C[j]))\} \ \textbf{\textit{(2)}} \end{aligned}$$

Which is the sum of (1) the number of times item i and j disagree in their order in the two rankings, or (2) the number of times the two item are tied (in one bucket) in one ranking, and not tied in the other one.

Example 3.1.5 *The generalised Kendall-τ distance between R and C. The underlined pairs represent the difference in order between the two rankings, which equates to a distance of 7.*

$$
\begin{array}{c}
R = [[3,\ 1],\ [5],\ [4],\ [2] \\
C = [[5,\ 1,\ 2],\ [3],\ [4]] \\
\underline{(1,2)} \quad \underline{(2,3)} \quad (3,4) \quad (4,5) \\
\underline{(1,3)} \quad \underline{(2,4)} \quad \underline{(3,5)} \\
\underline{(1,4)} \quad \underline{(2,5)} \\
\underline{(1,5)}
\end{array}
$$

In the same manner as the Kemeny score, given a ranking with ties R, the generalised Kemeny score measures the sum of the generalised Kendall-τ distances between the ranking and every ranking in the set S. Finally, generating a consensus ranking from a set of rankings requires finding the ranking that has the smallest generalised Kemeny score [51].

Definition 3.1.7 *Let S_n be the set of all possible rankings with ties over $[n]$ items. Given any subset of rankings $\mathcal{R} \subseteq S_n$, and a ranking R, the generalised Kemeny score K is defined as:*

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

Definition 3.1.8 *A consensus ranking R^* of a set of rankings with ties $\mathcal{R} \subseteq S_n$, under the generalised Kemeny score is mathematically defined as:*

$$K(R^*, \mathcal{R}) \leq K(R, \mathcal{R}),$$

Example 3.1.6 *Consider the set of rankings with ties $\mathcal{R} = \{R_1, R_2, R_3\}$, where $R_1 = [[3,1],[5],[4],[2]], R_2 = [[5,1,2],[3],[4]], R_3 = [[1,4],[2,3],[5]]$. The consensus ranking $R^* = [[1],[3,2],[5],[4]]$. The generalised Kemeny score of R^* is the result of order reversing $(1,3)$, $(2,4),(2,5)$ in R_1, $(1,5),(3,3),(2,5),(3,5)$ in R_2, $(3,2),(2,4),(3,4),(5,4)$ in R_3, tying $(3,2)$ in R_1, $(3,2)$ in R_2, and untying $(3,1)$ in R_1, $(5,1,2)$ in R_2, $(1,4)$ in R_3. Thus $K(R^*, \mathcal{R}) = 16$.*

3.1.3 Incomplete Rankings

We discussed in the last section the generalisation of Kendall-τ distance and Kemeny score to handle rankings with ties. Nevertheless, real-life databases may contain missing data, *i.e.,* incomplete data. In rank aggregation problem, there are two proposed methods to deal with incomplete rankings by converting them into rankings over the same items; projection and unification [53]

Projection The objective of the projection method is to remove from all rankings any items that do not appear in at least one ranking. The major drawback of this method is that it could result in the removal of items that may be pertinent to the problem we are trying to resolve.

Example 3.1.7 *Consider the set of rankings* $\mathcal{R} = \{R_1, R_2, R_3\}$ *with*

$$R_1 = [[1, 3], [4]]$$
$$R_2 = [[4], [1], [3, 5]]$$
$$R_3 = [[2], [1, 4], [3]]$$

Undoubtedly, the set of three rankings are incomplete since item 2 *is not present in* R_1 *and* R_2 *and item* 5 *is missing from* R_1 *and* R_3*. Applying the projection method results in removing these items.*

$$R_1 = [[1, 3], [4]]$$
$$R_2 = [[4], [1], [3]]$$
$$R_3 = [[1, 4], [3]]$$

Unification The unification method adds a unification bucket at the end of each incomplete ranking. The buckets consist of items that are present in other rankings but not in the current ranking. There are two considerations to be taken into account when applying the unification method. First, it must be noted that the missing items from a ranking are less important than the present items in that ranking. Second, even though missing items from a ranking are grouped together in one unification bucket, they do not possess an equal rank.

Example 3.1.8 *Consider the same set of rankings* $\mathcal{R} = \{R_1, R_2, R_3\}$ *from Example 3.1.7. Applying the unification adds a bucket with missing items at the end of each of the rankings.*

$$R_1 = [[1, 3], [4], [2, 5]]$$
$$R_2 = [[4], [1], [3, 5], [2]]$$
$$R_3 = [[2], [1, 4], [3], [5]]$$

In the remainder of this thesis, the unification method is applied to incomplete rankings before applying any consensus algorithm.

3.1.4 Consensus Algorithms

Many algorithms exist to address the problem of finding a consensus ranking. Brancotte *et al.* [52] studied 14 different consensus algorithms using the generalised Kendall-τ distance and classified them into score-based algorithms and positional-based algorithms. The foremost searches for a consensus by focusing on the disagreement between the order of the items, while the positional-based algorithms focus on the position of the items in each ranking.

[52] extensively compared and studied ranking algorithms with experiments on real, synthetic, and differently-sized datasets from different fields. The outcomes of the experiments showed that the BioConsert algorithm [43] and ExactAlgorithm [52] outperform the other algorithms providing highest quality results on both real and synthetic datasets. KwikSort [54] comes second after BioConcert, especially when the dataset is extremely large ($n > 30,000$). If execution time is a concern and the rankings do not contain many ties, then BordaCount [55] and MEDRank [56] are usable. In our research, the number of items and rankings are < 100, therefore execution time is not a concern. Hence, we intend to generate recommendations using BioConcert, ExactAlgorithm, and KwikSort algorithms.

The ExcatAlgorithm [52], non-heuristic approach, is based on a linear programming (LP) approach that finds an exact consensus solution (*i.e.,* ranking) by maximising or minimising an objective function such that all predefined constraints are satisfied. However, the intrinsic complexity of linear programming limits the ability to compute optimal solutions to large datasets. While BioConcert and KwikSort algorithms are heuristic score-based algorithms, they differ in the way they construct a 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 generalised Kemeny score is stabilised. The two operations are (1) changeBucket: moves an item from one bucket and adds it into an existing bucket and (2) addBucket: moves an item from a bucket to put it in a new bucket [43]. KwikSort, on the other hand, employs a divide-and-conquer approach. It randomly assigns one of the items as a pivot and then recursively places the rest of the items in two buckets after and before the pivot until a consensus ranking is reached with a minimised generalised Kemeny score.

3.2 Fundamentals for Building Recommendation Systems for Software Engineering Applications

Collected input data is likely imperfect, inconsistent and cannot be directly used to generate recommendations. Further, the process of evaluating and choosing a perfect technique is

one of the hardest steps in building recommendation systems. A successful recommendation system should be based on a consistent and well-defined methodology for preprocessing the input data, building and evaluating the system. In this section, we discuss fundamental phases for building recommendation systems for software engineering. This overview can be used as a starting point for recommendation builders with limited knowledge when building RSs. The Section is based on observations made from our research studies, Chapters 4-5-6, and a review of the few major works that have been done for construction RSSEs. Figure 3.1 illustrates an overview of the phases involved in building a recommendation system. We break the process down into a series of phases: Problem Identification (Section 3.2.1), Data Collection (Section 3.2.2), Data Preprocessing (Section 3.2.3), Recommendations Generation (Section 3.2.4), and Recommendation System Evaluation (Section 3.2.5).

3.2.1 Problem Identification

A recommendation system is designed to assist software developers in solving particular engineering problems by recommending tools, source-code, documents, navigation patterns, among others. For more fine-grained and precise recommendations, it is necessary for recommender builders to first identify the problem that the recommender intends to resolve and whether the recommender will be able to provide valuable recommendations to a developer facing a certain challenge. In particular, the recommender's objective and the task that the recommender aims to accomplish should be clearly outlined [28]. A thorough understanding of the objective and task is crucial to the production of valuable recommendations.

By clearly defining the problem, objective, and task, recommender builders can accurately determine what type of input data and tools are required and the target developers. The type of input data can impact the analysis and preprocessing of the collected data as well as the development effort needed for that particular recommender. For example, source-code takes less effort to parse and analyse than developers' interaction traces. Additionally, knowing the target developers can help determine the kind and level of recommendations that a recommender may provide. A novice developer, for instance, requires more detailed recommendations than an expert developer [28]. Examples of software engineering problems that have been solved with the help of recommending systems include helping software developers complete a particular task by recommending relevant source-code, plan software development by recommending code to reuse, design the software through suggesting design patterns, and many others.

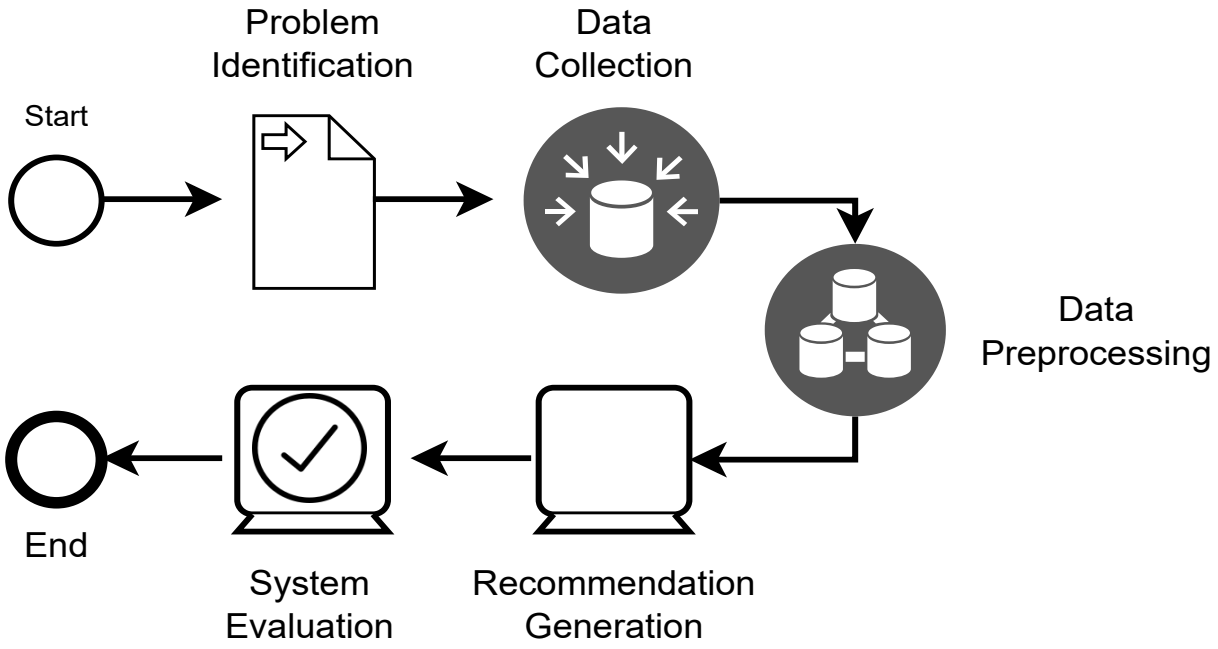


Figure 3.1 Overview of the Fundamental Phases of Building RSs.

3.2.2 Data Collection

Recommendation systems are data processing systems that actively collect different types of data to be able to generate recommendations. Until sufficient constructed data is available, a recommender cannot effectively provide accurate recommendations. Input data is primarily about the items to recommend. Software engineering researchers have used various data types and collection tools for building recommendation systems. Some of the commonly used input data are [28]: • **Developers' Interaction History Data** Studies employed developers' interaction history data to recommend relevant artefacts, reusable pieces of code, predict defects, or warn against code violation. Tools, such as Mylyn, Switch, and OCompletion, can be used to track, collect and extract developers' interactions with the system. • **Source-Code** Source-code related data is another input type that has been used actively for building RSSEs. This type of data can be generated by a source-code static analysis of the software under study. • **Bug Reports** Data of this type can reveal a great deal of information concerning the history of software development issues. Recommender builders can extract bug data from issue tracking systems, which are platforms for software users, developers and testers to track and report bugs. • **User Reviews** Online user reviews behave like word-of-mouth, which can significantly impact a software system's success or failure.

Review scrapping is the process for extracting online user reviews from web sources, such as Google Play, using web scraping tools. • **Commits** Expertise recommendation systems rely on commit logs from version control systems, such as SVN, CVS, and Git, to recommend developers with expertise in a specific part of the software system. • **Explicit Data** The explicit data collection strategy involves explicitly asking developers to provide data about the issue being investigated through tracking developers' interactions, feedback collection, survives, or observational experiments.

3.2.3 Data Preprocessing

Accuracy, completeness, consistency and currentness are four dimensions that define data quality [57]. However, collected input data is far from reflecting these dimensions and often noisy, unreliable, redundant, inconsistent, and incomplete. Low-quality data does not directly apply to recommendation systems, as they affect the performance and accuracy of the generated recommendations. Therefore, preprocessing of the collected data is mandatory.

Data preprocessing is a multi-step process that transforms raw data into a form that is sufficiently comprehensible and of high quality [30]. From the entire recommendation system building process, preprocessing data takes the most effort and time ($> 50\%$ of total effort) [58]. For instance, preprocessing can include aggregating commits, parsing source-code, tokenizing text, and abstracting software. The product of a successful data preprocessing is a reliable dataset that can be fed directly to the recommendation technique.

In the remainder of this section, we describe the most common data preprocessing techniques that are used to prepare and clean datasets. This phase includes: Data Cleaning, Data Integration, Data Transformation, and Data Reduction [29, 59]. The diagram (3.2) below depicts the basic techniques involved in data preprocessing.

- **Data Cleaning** Data cleaning or data cleansing is the process of fulfilling missing data, removing duplicates, correcting or filtering out incorrect data, and deducting inconsistencies by applying cleaning techniques.
- **Data Integration** In some studies, input data can be coming from different sources. In such a situation, data integration must be applied in order to combine the data into a single unified representation. It is important that this process is carefully performed in order to avoid redundancies and inconsistencies, which may result in inaccurate data.
- **Data Transformation** It is about transforming raw data into consistent format. It consists of two operations, data normalisation and data generalisation. Data normal-

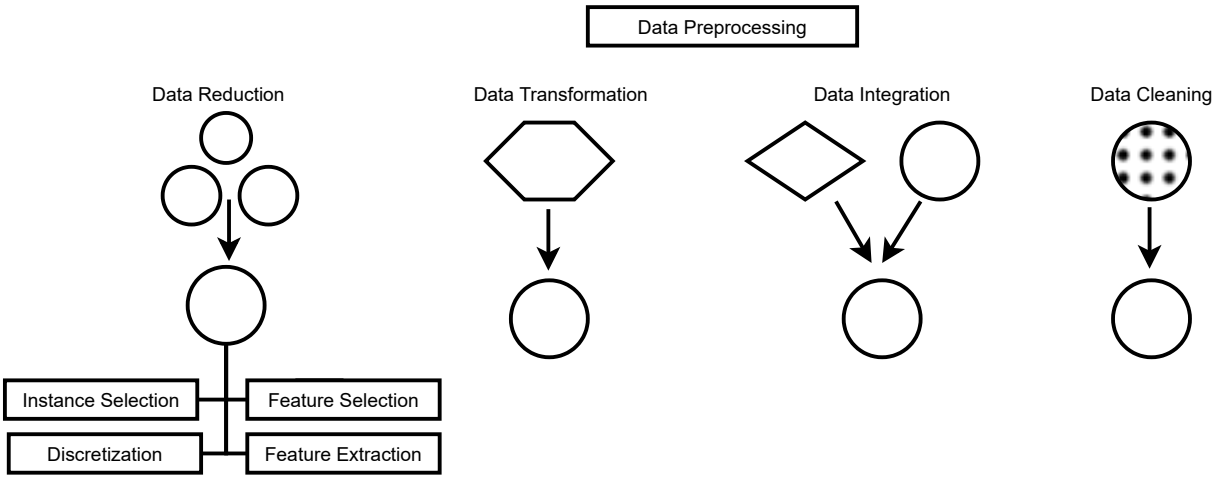


Figure 3.2 Overview of Data Preprocessing Techniques

isation is a technical process of transforming data to be on the same scale without changing the content of the data. This process reduces processing time and complexity. For example, it can be applied if some values in the dataset are measured in inches while others are measured in centimetres. In contrast, data generalisation involves converting low-level data features into high-level features. It helps provide a clear and general picture of the input data. For example, low-level features such as postal code can be generalised to high-level features such as city or state.

- **Data Reduction** The size of the dataset can be very enormous, making the processing of the data very expensive. Such cases make the data reduction process necessary. Data reduction consists of a set of techniques that can be used to eliminate redundant and noisy features or instances, or simplify features using discretization techniques. It archives a reduced presentation of the data while maintaining the integrity of the original data. In this process, four techniques can be used to resolve basic data issues: data dimensionality (Feature Selection (FS)), data redundancy (Instance Selection (IS)), features simplification (Discretization), and missing data (Feature Extraction).

  - **Feature Selection (FS)** Feature selection achieves reducing the dimensionality of data by selecting a subset of data features and removing as much irrelevant data as possible. The main objective is to identify a subset of data features that can generally represent the main issue and be used as input to train an algorithm. As feature selection reduces the dimensionality of data, it enables algorithms to run faster, reduce costs, and produce highly accurate results.

- **Instance Selection (IS)** This technique involves selecting a subset of data that can represent the entire dataset while producing the same results as if the entire dataset was used. While IS is a separate process from FS, both processes may be applied in conjunction with one another. As a complementary process of FS, IS promotes data reduction in order to scale down the data. Instance selections have the advantage of allowing algorithms to run on large datasets while focusing only on relevant parts of the dataset.
- **Discretization** In general, data is presented in a variety of formats, such as discrete, numerical data, continuous data, categorical data, *etc.* Numerical data, whether it is discrete or continuous, is assumed to be ordinal, *i.e.,* the values have an order. In categorical data, however, there is no order amongst them. Depending on the algorithm being used, some algorithms can only run on nominal values. As a result, a discretization technique is required to convert the continuous data into a nominal format that the algorithm can understand. The discretization process converts quantitative data into qualitative data by splitting the numerical values into a number of non-overlapped intervals. The values become discrete by associating each numerical value to a specific interval. The outcome of the discretization process is a set of nominal data. In practice, the discretization process simplifies the reading of data and makes algorithms more efficient at generating accurate results.
- **Feature Extraction** Feature Extraction extends feature selection by allowing the modification of the selected subside of features. It generates new artificial features from the subset of features. The goal of this process is to further reduce the dimensionality of the dataset by generating artificial features that are smaller in size than the original subset.

3.2.4 Recommendations Generation

Once the problem is defined, tasks are created, input data is determined, collected and cleaned, the recommender builder must select one or more recommendation techniques to implement the core function of the recommendation system. The recommendation technique(s) take the input data and transform them into recommendations. Based on the selected recommendation technique and how it operates, the set of generated recommendations may contain one or multiple recommendations.

Over a long time, extensive research has been conducted to study and develop recommendation algorithms and techniques. Various techniques have emerged to choose from to re-

solve problems in different domains. These techniques are constructed under three basic recommendation approaches; collaborative filtering (CF), content-based filtering (CBF) and knowledge-based recommendation (KBR) [5]. No one technique is designed to solve a particular problem.

All three approaches have strengths and weaknesses, and each is derived from different sources of recommendation knowledge. There is no set rule for determining which approach or technique to apply. The technique employed is determined by the type of input data and the nature of the problem being studied.

The idea of collaborative filtering approach originated from an information filtering technique that uses the opinion of a group of people to recommend items to individuals. The implementation of the approach works by building a user-item matrix of items that users like. It then recommends to users items that other users with similar interests and preferences liked in the past. Similarity measures are used to calculate the interest similarity between users. This approach is the most widely used for building recommendation systems. In a content-based filtering approach, the emphasis is more on the analysis of item features in order to generate recommendations. The system extracts features from the content of the items in the dataset, uses similarity metrics to measure the similarity between features and recommends items that are with similar features. Therefore, it is essential to provide an in-depth description and content about the items in order to extract features. For example, if a user likes a mobile application that belongs to the food category, then the system will recommend another app from the same category. Unlike the other two approaches, the knowledge-based approach is based on collecting and extracting explicit knowledge about both the user, the item and their relationships. The approach tries to construct a functional knowledge from this relationship that provides information about how a particular item can benefit a particular user's needs. Under this approach, two methods are used to produce recommendations: content-based and constraint-based. The case-based method bases recommendations on knowledge of specific cases, where items are presented as cases and recommended in a way that solves the user's specific problem. The constraint-based method, in contrast, specifies a set of constraints that relate user's needs with items.

3.2.5 Recommendation System Evaluation

Providing accurate and valuable recommendations is crucial to achieve the objective of building the recommendation systems. Therefore, RSs must undergo some evaluation processes. The quality of the recommendation technique can be assessed through different evaluation methods. There is no golden evaluation method that should be used for a perfect evaluation.

In this section, we review the two most common evaluation strategies that help evaluate the effectiveness of the recommendation systems and compare them: Evaluation Experiments and Evaluation Metrics. Depending on the characteristics of the recommendation system, an evaluator may employ both strategies or only one.

Evaluation Experiments There are three way to conduct evaluation experiments:

- **Offline Experiments** This type of experiment is considered the easiest to perform as it involves no real interactions with users. It consists of using existing datasets. The dataset is split into "training" subset and "test" subset. The former is used as input to evaluate the recommendation technique, while the latter serves as ground truth data for measure computation. The experiment is performed by simulating interactions of users using the recommendation system and predicting that these interactions will be the same or similar to the interactions of real users using the recommendation system when it is deployed. This simulation should help estimate the performance of the technique [34].

- **User Study** This type of experiment can be time-consuming as it involves interacting with a small set of real users, however it can provide a good insight into the impact of recommendation systems on users behaviour. User study experiment is best suited for evaluating recommendation techniques that depend on real user input, such as association rules. It consists of defining a set of evaluation tasks for users to complete while using the recommendation system in a controlled environment; experiment in which external variables are controlled or their influence on experimental results is eliminated. While users conduct the experiment, quantitative and qualitative data about the system performance can be collected, such as recording user behaviour, number of completed tasks, total time needed for each task, or percentage of accurate results. In most cases, the user study is followed by a survey question to collect qualitative data regarding the user's experience with the system. User study experiment can be Between Subject design or Within Subject design. In the Between Subject, users are divided into groups, and each group performs and experiences a different set of tasks than the other group. On the contrary, in the Within Subject design, all users perform and experience all tasks in the set [34].

- **Field Study** Field studies are designed to experiment with real scenarios and conducted in collaboration with industrial companies that provide access to a natural work environments of the users. The recommendation system is used by the users while performing their day-to-day work tasks. Users are observed to assess the impact

of the system on their behaviour as well as the performance of the system. Qualitative interviews can be conducted after the experiment to obtain constructive feedback about the experiment in general and the recommendation system in particular [28].

Evaluation Metrics Regardless of the experiments' ability to provide an informative assessment of the performance of the recommendation technique and its impact on user behaviour, it is necessary to assess the accuracy of the technique in the form of numbers that can be compared to other techniques. There are several accepted evaluation metrics that can be used for this purpose; however, the metric chosen depends on the specific recommendation under investigation. Metrics can be grouped into two categories: Error Metrics and Accuracy Metrics.

- **Error Metrics** These types of metrics tell us how accurate the recommendations are and what is the amount of error (*i.e.,* how far the recommendations are from the actual values). Two metrics are introduced, Root Mean Squared Error (RMSE), and Mean Absolute Error (MAE). The difference between the two metrics is that the error values are equally weighted in MAE, while large error values are highly penalised in RMSE. RMSE is therefore a more appropriate metric if large errors are undesirable. In both metrics, lower error values indicate higher accuracy [30].

  Root Mean Squared Error (RMSE) RMSE is calculated by finding the difference between the actual value and the recommended value, which is called the *residual*. Residual can be positive or negative. If $\hat{y}$ is the recommended value and y is the actual value, and N is the number of values, RMSE is calculated by squaring the residuals, averaging the squares, and taking the square root as follows:

$$RMSE = \sqrt{\frac{\sum(\hat{y} - y)^2}{N}}$$

  Mean Absolute Error (MAE) MAE is calculated by finding the average absolute deviation between the recommended value and the actual value:

$$MAE = \frac{\sum \mid \hat{y} - y \mid}{N}$$

- **Accuracy Metrics** Accuracy provides a measure of how close the produced recommendations are to a set of expected predefined recommendations that is referred to as ground truth or gold standard. The accuracy of a recommendation indicates how well

it delivers the defined objective of the recommendation system. The most commonly used accuracy metrics are Precision, Recall, and F-measure [60].

Precision P represents the proportion of correctly recommended items. The greater the precision, the more accurate the recommended items made by the recommendation system.

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

Recall R represents the proportion of recommended items that are actually met. The greater the recall, the more items that are actually recommended by the recommendation system.

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

F-measure (F) is computed to help simplify Precision and Recall into one metric. It represents the accuracy of the recommendation. The greater the F-measure, the more accurate the results of the recommendation system.

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

Where TP (true positives) the recommended items that are relevant, FP (false positives) the recommended items that are not relevant, and FN (false negatives) the relevant items that are not recommended. An optimal value is 1, which means the recommendation technique is capable of generating accurate recommendations. These metrics make comparing between recommendation techniques and across the same dataset very straightforward.

CHAPTER 4 PRIORITISING MOBILE APP USER REVIEWS FOR MOBILE APPLICATION EVOLUTION

4.1 Introduction

In this chapter, we test and validate the consensus algorithm to prioritise and recommend mobile apps user reviews. The mobile apps industry grew tremendously in the past few years. Apple App Store and Google Play, the two largest marketplaces for mobile apps, launched in 2008 and host more than 5 millions apps. As of September 2019, Google Play and Apple App Store reached a total of 5.5 million apps [61].

In addition to the download service, these marketplaces allow users to rate and review the apps using a five-point Likert scale and unstructured text. These ratings and reviews are important. Beside guiding prospective users in the choice of apps to download, they provide a valuable source of information for app maintenance and development. Harman *et al.* [62] showed that there is a high correlation between user ratings and reviews and numbers of downloads. Other studies [9,26,27] showed that up to one third of the information in the user reviews could help developers maintain and develop their apps. Therefore, app developers must consider user reviews when updating their apps.

However, processing and analysing these reviews present three challenges. First, the volume of user reviews received for some apps is extremely large and surpasses developers' ability to read them manually. For example, while the WordPress app receives about 100 reviews every month [63], the Facebook app gets more than $4,000$ reviews per day [64]. Second, reviews contain unstructured text that is difficult to parse and analyse [65]. Third, the portion of high-quality reviews is relatively small, only about one third is useful for app improvement [9]. To solve these challenges, many approaches exist to process, analyse, classify, and cluster user reviews [27, 62, 65–69]. For example, AR-Miner [9] tags reviews as informative and non-informative, extracts, groups, and ranks topics from reviews. CLAP [70] categorises user reviews into categories and clusters related reviews. URR [71] classifies reviews based on topics and sub-topics taxonomy and links reviews to source-code. These approaches could help developers identify important reviews, however their prioritisation techniques are limited to either labelling them as high/normal, or ranking them based on a fixed formula. They are not flexible enough to consider every possible review attribute (*i.e.,* attributes to rank reviews) into the prioritisation process.

We claim that developers would benefit from a prioritisation technique that (1) takes into

account multiple review attributes and (2) finds a consensus among all reviews and their attributes to help developers plan the next releases of their apps. To validate our claim, we propose Review Prioritiser (RP), an approach that uses the consensus algorithm to recommend a prioritised list of user reviews that could help app developers in deciding what improvements to implement in the next releases of their apps.

Figure 4.1 summarises the steps involved in developing Review Prioritiser. First, we obtain the publicly-available user reviews and used in a previous work [70]. Second, we preprocess, categorise, and cluster the reviews using CLAP [70]. Third, we analyse reviews to define a set of review attributes that are commonly used by developers to rank reviews (cardinality, oldest date, average rating, category). Fourth, we rank the clusters of reviews based on these four attributes. For example, the ranking by the oldest date is an ordered list of clusters in which the first cluster contains the review with the oldest date among all reviews. Fifth, we apply the consensus algorithm to the rankings to generate one consensus ranking of prioritised clusters of reviews for developers to address in the next release.

We validate Review Prioritiser by prioritising user reviews of four Android apps released on Google Play. First, in quantitative evaluation, we compare the consensus rankings generated by RP with rankings manually-prioritised by four app developers by computing the Kendall rank correlation coefficient. Second, we perform a qualitative evaluation by inviting app developers to evaluate the consensus rankings of RP. Finally, we conduct a questionnaire user study to gain an understanding of whether app developers would consider our approach to plan their next release. Results show that our approach can produce quantitatively and qualitatively useful rankings and, thus, help app developers prioritise users reviews when working on a new release.

The rest of this chapter is as follows. Section 4.2 presents related works. Section 4.3 describes a motivating example. Section 4.4 presents Review Priortiser. Section 4.5 presents evaluation methods and results. Section 4.6 discusses the results. Finally, Section 4.7 discusses threats to validity and Section 4.8 concludes the chapter.

4.2 Related Work

Many studies pertain to user feedback to extract features for different purposes. Harman *et al.* [62] extracted features from user feedback via data mining and reported a strong correlation between user rating and number of app downloads. Machine learning approaches were used [27, 65] to process user reviews and extract features to classify these reviews into maintenance and evolution categories. Iacob *et al.* [66] analysed hundreds of reviews to define

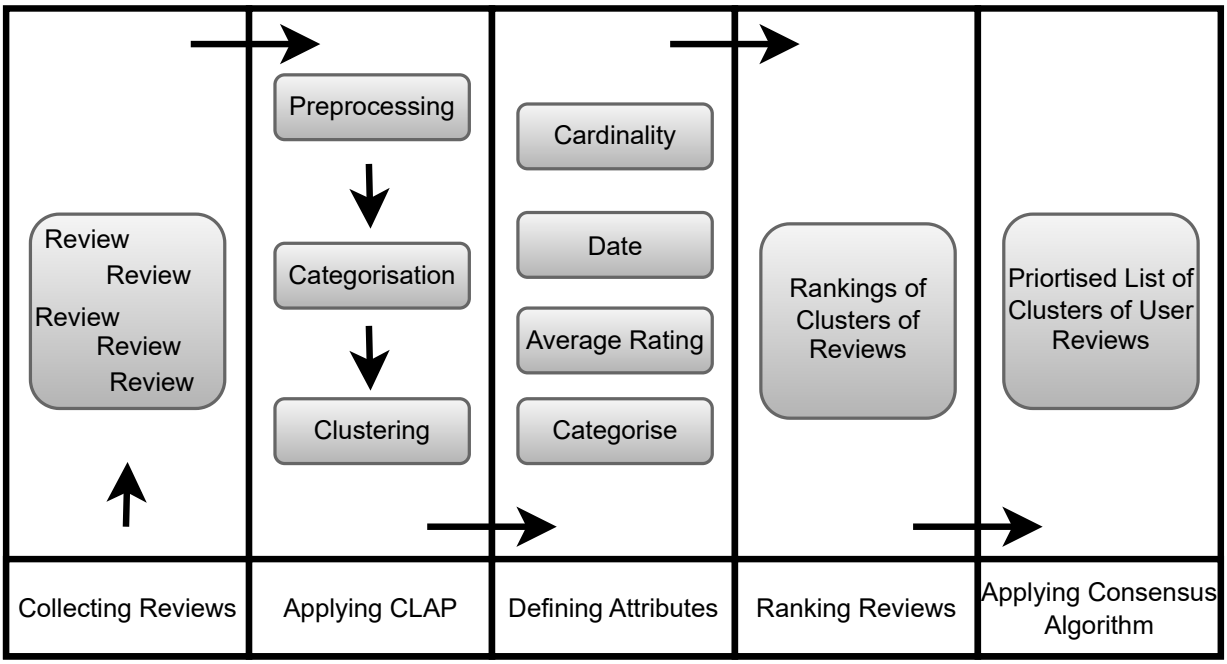


Figure 4.1 Overview of Review Prioritiser Approach

a set of the most common topics discussed in reviews. In addition to extracting features for classifying reviews, Ciurumelea *et al.* [71] linked classified reviews with source-code to help developers find code to modify.

Studying user feedback for information retrieval has gained much attention, *e.g.,* automated information retrieval systems to query specific features or keywords in reviews [69]. Pagano and Maalej [26] performed an exploratory study and concluded that (1) numbers of reviews decrease in time, (2) most reviews focus on three main topics, and (3) positive reviews lead to high download. Fu *et al.* [67] identified reasons for users to like/dislike apps at three levels of granularity. Gebauer *et al.* [72] used structural equations to identify factors impacting user reviews.

Our approach is different from the previous ones. We focus on studying user reviews to prioritise them in a consensus way to help developers plan their apps' future releases.

Laurent *et al.* [73] proposed a semi-automated technique to prioritise software requirements using a probabilistic traceability model. Avesani *et al.* [74] introduced a case-based ranking framework for software requirements prioritisation, which uses a pairwise comparison technique to identify/explain preferred requirements. Beg *et al.* [75] used B-trees to prioritise software requirements. These approaches do not apply to mobile app reviews because they (1) do not work well on a large number of reviews, (2) assume that clients participate in

the prioritisation process, and (3) do not perform preprocessing, categorising, and clustering reviews activities.

Keertipati *et al.* [76] defined three user review prioritisation approaches on four review attributes (frequency, rating, negative emotions, and deontics). Another approach proposed by Gao *et al.* [77] is PAID to find issues in reviews at phrase level rather than sentence level. These approaches do not take into account all possible review attributes in the prioritisation process, and do not perform any review preprocessing.

AR-Miner by Chen *et al.* [9] was the first automated approach to preprocess, classify, and rank user reviews. CLAP by Scalabrino *et al.* [70] performs better than AR-Miner when categorising and clustering reviews. In addition, CLAP prioritises the clusters of reviews by labelling them as high/normal. Our approach differs from these two. It aggregates the results of many rankings and provides developers with one consensus ranking of ordered reviews.

4.3 Motivation

To motivate our work, we use this running example: Matthew is an app developer for WordPress [63]: an Android app on Google Play, with an average rating of 4.5, over 135,435 submitted reviews, and over 10 million downloads. Matthew must regularly update his app with new features and bug fixes to maintain its high rating and user satisfaction. He needs to know what is the general opinion of users about his app, expected features, and reported issues. Therefore, he frequently reads the many user-submitted reviews on Google Play. However, he finds himself spending too much time analysing, and extracting information from the reviews due to the unstructured nature of these reviews. Table 4.1 shows some reviews for his app on Google Play on different dates. Clearly, the first and second reviews are somewhat useful and can help Matthew improve his app; the other two contain no useful development information.

ID	Date	Rating	Review
0	4/4/2014	5	I need the justify post feature
2	4/5/2014	4	Stats broken in last update
5	4/6/2014	2	It does not work
6	4/8/2014	3	It does not go through my self hosted blog

Table 4.1 A Few User Reviews from WordPress App. v2.7.1

To reduce effort, Matthew looks for an automated mobile app review analysis tool that can do the job for him. He chooses and applies CLAP on reviews of his app, which preprocesses, categorises, and clusters reviews for him as illustrated in Table 4.2.

ID	Review	Category	Cluster
0	I need the justify post feature	Feature	C1
1	More options, like text coloring	Feature	C1
2	Stats broken in last update	Bug	C10
3	Facebook shared posts show other photos	Bug	C11
4	Cannot log in	Bug	C12
7	slow and buggy	Perf.	C17
8	Unacceptably poor UI	Usability	C18
9	Wish it is easier to format text and images	Usability	C18

Table 4.2 CLAP Categories and Clusters for Some Reviews

Matthew successfully produced preprocessed, categorised, and clustered reviews but now he needs an approach to prioritise the clusters of reviews for an effective release planning. When it comes to ranking reviews, there are different attributes to consider. For example, Matthew could rank the clusters based on the average rating, oldest review's date, or categories as illustrated in Table 4.3.

Attribute	Ranking
Average Rating: order by average rating of each cluster ascendingly	[[C18],[C11,C17],[C10],[12],[C1]]
Oldest Date: order by the oldest submission date of the reviews in each cluster	[[C18,C1],[C10],[C12],[C11],[C17]]
Category: order by cluster categories: first bug, then performance, usability, and finally feature	[[C10,C11,C12],[C17],[C18],[C1]]

Table 4.3 Rankings of the Clusters Presented in Table 4.2

Applying different attributes, prioritises the clusters differently, making it problematic for Matthew to identify the most pressing reviews as none of the rankings might optimally provide the best order. Thus, we propose to apply a consensus-based recommendation technique to the set of these rankings to aggregate them into one consensus ranking that prioritises the review in a consensus fashion and will help Matthew identify the most important reviews to address in his next releases.

4.4 Review Priortiser

Review Prioritiser (RP) is an consensus-based recommendation approach that takes as input a set of rankings of clustered user reviews, uses a consensus algorithm to recommend a consensus ranking consisting of a prioritised list of user reviews. Review Prioritiser should support mobile app developers with app maintenance and evolution activities. We now

discuss (1) app selection and data collection, (2) data preprocessing, (3) review attributes definition and review ranking, (4) consensus ranking of user reviews generation.

4.4.1 App Selection and Data Collection

User Reviews usually come with the following information: (1) title, (2) free unstructured text, (3) star rating (between 1 and 5), (4) reviewer's username, (5) date of the review, and (6) version of the reviewed app. The approach can be applied to any size of dataset, reviews from any year, and from any mobile app store (such as Apple App Store , Google Play, or BlackBerry World).

In this study, we use the publicly-available dataset of user reviews by Scalabrino *et al.* [70], which contains 725 user reviews from Google Play for 14 Android apps. The reviews roughly cover the period from October 2010 to May 2014 and cover different app categories. The diversity of the categories ensures the variety of the reviews that are submitted by different users with different interests, which helps achieve a higher level of generalizability when evaluating our approach in the following section. Table 4.4 provides information about each app's name, category, and number of collected reviews. We decided to eliminate the data of Harvest Moon BTN app from the dataset because the app does not exist anymore.

App Name	Category	Number of Reviews
BOINC	Eduaction	30
Lightning Web Browser	Communication	27
~~Harvest moon BTN~~	~~Game~~	~~34~~
Timeriffic	Tools	25
iFixit: Repair Manual	Lifestyle	5
DuckDuckGo	Tools	17
eBay	Shopping	260
Barcode Scanner	Shopping	21
Ringdroid	Music	23
2048	Puzzle	11
Viber	Communication	108
Dolphin Emulator	Arcade	107
LinePhone	Communication	28
WordPress	Productivity	29

Table 4.4 List of Android Apps Contained in the Dataset

4.4.2 Data Preprocessing

User reviews contain useful information for app developers to maintain and evolve their apps. Yet, they come with challenges: (1) some apps receive high volume of reviews daily, (2) a

single review can report more than one issue, (3) reviews include unstructured text [65], and (4) low quality, non-informative reviews are numerous [9]. As a result, preprocessing the reviews before prioritisation becomes mandatory.

We performed a comparison study of five automated review analysis tools: (1) ARdoc (App Reviews Development Oriented Classier) [64], (2) URR (User Request Referencer) [71], (3) SUR-Miner (Software User Review Miner) [62], (4) AR-Miner (App Review Mining) [9], and (5) CLAP (Crowd Listener for releAse Planning) [70]. To help decide which tool we can employ to preprocess the dataset, we defined a set of questions: (1) is the tool automated? (2) does the tool apply NLP techniques (*e.g.,* tokenizing and stemming)?, (3) are non-informative reviews filtered out?, (4) does the tool categorise reviews? (5) does the tool cluster related reviews? and, (6) how accurate are the results of the tool?

We applied each of the tools to the extracted dataset. Two of the authors compared the results of the tools and concluded that CLAP is the only tool that answers positively all the questions. CLAP preprocesses, categorises the reviews into eight categories (bug, feature request, performance, security, energy, usability, and other), and clusters related ones together with high accuracy. Table 4.5 presents a sample of the results of applying CLAP to the WordPress app reviews. Full results are available in our replication package [78].

Review	**Category**	**Cluster**
I need the justify post feature	Feature	C1
Stats broken in last update	Bug	C10
It is slow and buggy.	Performance	C17
Wish it is easier to format text and images.	Usability	C18
Some error is coming and it does not install.	Bug	C15

Table 4.5 Results of Applying CLAP to WordPress Reviews

4.4.3 Attributes Definition and Reviews Ranking

In this step, we rank the clusters of reviews that are produced by CLAP according to a set of defined review attributes independently. We performed an iterative reviews analysis to identify attributes that app developers could use for ranking reviews when deciding what change requests to address in the next release. We identified four attributes: cardinality, oldest date, average rating, and category.

- **Cardinality**: clusters of reviews are ranked according to the number of reviews in each cluster decreasingly. Clusters with higher numbers of reviews indicate that the same issue has been reported by many users.

- **Oldest Date**: clusters of reviews are ordered chronologically (from oldest to most recent) based on the oldest date of review included in the cluster.
- **Average Rating**: we compute the average rating of all reviews in a cluster and rank clusters with lower average rating first.
- **Category**: we sent a survey to some app developers and Ph.D. students with app development experience to understand how they would rank categories. 17 developers answered as follows: bug, security, performance, energy, usability, and feature. We rank the clusters according to their ordering of the categories.

We order the clusters of reviews from CLAP based on the four chosen attributes to generate a set of rankings. Table 4.6 shows the results of ranking WordPress clusters of reviews and Table 4.7 summarises the reviews in each cluster. Ties in rankings indicate that tied clusters have the same value, hence, they have an equal rank.

Attribute	**Ranked Clusters**
Cardinality	[[9],[8,12],[1,4,14,18],[2,5,6,7,10,11,17,3,13,15,16]]
Date	[[2],[9,1,4,14,18,5],[6,10],[12],[8,15],[11,3,13,16],[7,17]]
Average Rating,	[[16],[12],[9],[18,15,3,13],[14,11,17],[4],[5,6,10,8,7],[2,1]]
Category	[[16,12,9,15,13,14,11,10,8],[17],[18],[3,4,5,6,7,2,1]]

Table 4.6 Ranked Clusters of WordPress reviews

Cluster	**Summary**	**Cluster**	**Summary**
C1	Text justify feature	C10	Stats is broken
C2	Ability to add featured image	C11	Posts tab not showing posts
C3	Ability to modify multiple posts	C12	App log in issue
C4	Ability to format text	C13	Update are removing websites
C5	Ability to edit profile	C14	html tags appear when editing post
C6	Unable to upload media to posts	C15	Posts getting published
C7	Black background theme option	C16	Installing app issue
C8	Website log in issue	C17	Slow and buggy
C9	Image upload error	C18	Poor UI and navigation difficulty

Table 4.7 Summary of Reviews in Each Cluster of WordPress

4.4.4 Consensus Ranking Generation

In this final step, we apply a consensus algorithm to the four formed rankings of reviews from the last step to produce one consensus ranking that best agrees with all the input rankings.

This consensus ranking should assist developers in identifying issues to resolve and features to include in the next release of their apps.

As discussed in section 3.1.4, we apply the top three performing consensus algorithms (ExactAlgorithm, BioConcert, and Kwiksort) to generate the consensus ranking of user reviews. Table 4.8 presents the consensus rankings produced by each algorithm when applying them to the four rankings from Table 4.6 for WordPress app along with the generalised Kemeny score of the results from each algorithm. Comparing the results generated from applying each consensus algorithm for all apps reviews, we notice that they are quite similar to one another with similar generalised Kemeny score as well. This is partly due to the small number of clusters in each ranking. Consequently, we decided to adapt the results of the ExactAlgorithm in this study since it guarantees an exact solution. We would consider BioConsert or Kwiksort when the dataset of reviews is much larger. The consensus rankings of the remaining apps are available in the replication package [78].

Consensus Algorithm	**Consensus Ranking**
ExactAlgorithm	Distance = 240 [[9],[12],[14,18],[8],[1,4],[10,11,3,13,15,16],[17],[2,5,6,7]]
BioConcert	Distance = 240 [[9],[12],[14,18],[8],[1,4],[10,11,3,13,15,16],[17],[2,5,6,7]]
Kwiksort	Distance = 241 [[9],[12,8],[1,4,14,18],[16,10,11,3,13,15],[17],[5,2,6,7]]

Table 4.8 The Consensus Rankings for WordPress from Applying the Three Selected Consensus Algorithm

4.5 Evaluation & Results

In this section, we evaluate the efficiency of the consensus-based technique in prioritising the user reviews and the usefulness of the results of Review Prioritiser (RP) for app developers. Three forms of evaluation are performed, quantitative, qualitative and questionnaire user study. The evaluation should provide answers for the following study questions:

RQ1: (Performance) How effective is the consensus algorithm in prioritising user reviews? We evaluate the performance of the consensus algorithm in prioritising review clusters (1) quantitatively by computing the correlation between the consensus ranking and a gold ranking (GR) *i.e.,* a set of review clusters manually-ranked by real app developers and (2) qualitatively by asking app developers to evaluate the results of Review Prioritiser.

RQ2: (Interest) Do mobile app developers prioritise user reviews and would they consider our approach when planning new releases? We address this question

by conducting an online questionnaire user studies of apps developers to understand how developers perform prioritisation activity and how interested they are in our approach.

4.5.1 Quantitative Evaluation

To quantitatively evaluate the efficiency of the consensus algorithm in prioritising user reviews, we compare every consensus ranking from the results with a gold ranking (GR), *i.e.,* a manually defined ranking by real app developers, by computing the generalised Kendall-τ distance and the Kendall rank correlation coefficient.

To create a gold ranking for every app in the dataset, we communicated with developers of the 13 apps We manually extracted the developers' email addresses from the app's Google Play Web pages. We sent each developer a description of our research, a link to a Web site presenting the clusters of reviews of their apps, and a guide on how to rank the clusters as they see fit. We did not provide them with the consensus ranking to avoid any bias. Developers of only four apps responded to our invitation (BOINC, DuckDuckGo, Viber, and WordPress), and provided us with gold rankings. We show the GR of WordPress app, while the gold rankings of the other three apps are available in the replication package [78].

$$\begin{aligned} GR_{WP} \;=\; & [[12],[16],[13],[8],[14],[15],[11],[10],\\ & [9],[18],[19],[6],[4],[1],[2],[3],[5],[7]]. \end{aligned}$$

We answer RQ1 by computing the similarity between every consensus ranking and its relevant gold ranking using two similarity measures: the generalised Kendall-τ distance (see Definition 3.1.6), and the Kendall rank correlation coefficient [79].

Generalised Kendall-τ Distance First, we compute the worst possible distance between any consensus ranking and the gold ranking by computing the generalised Kendall-τ distance between the GR and its reversed ranking. Then, we compute the actual distance between every consensus ranking and its relevant gold ranking using the same measure. Finally, we compare those two distances.

Table 4.9 reports the worst possible distances and the generalised Kendall-τ distances between the consensus rankings and the gold rankings for each app. The results suggest that the distances between the consensus rankings and the gold rankings are fairly small and far from reaching the worst distance. Despite that WordPress reports a relatively large distance, it is still 61.4% away from the worst distance.

Kendall Rank Correlation Coefficient is a correlation that statistically measures the association between two rankings. We measure the Kendall rank correlation coefficient for

App	Worst Distance	Actual Distance	Correlation Coefficient
WordPress	153	59	0.411
Viber	28	8	0.557
DuckDuckGo	21	7	0.514
BOINC	55	14	0.585

Table 4.9 Worst vs Actual generalised Kendall-τ Distance and Correlation Coefficient between the Consensus Rankings and Gold Rankins.

tied ranks between the two rankings of each app to measure if there is a statistically significant association between the consensus ranking and gold ranking. The correlation is defined arithmetically as [79]:

$$\frac{C - D}{\sqrt{(\frac{1}{2}n(n-1) - U)(\frac{1}{2}n(n-1) - V)}}$$

where C = number of concordant pairs, D = number of discordant pairs, n = number of items, U = number of tied pairs in the first ranking, and V = number of tied pairs in the other ranking. The correlation coefficient is in the interval $[-1, +1]$ in which, according to Cohen *et al.* [80], values of 0.5 or greater indicate a large correlation, values between 0.5 and 0.3 a medium one, values between 0.3 and 0.1 a small one, and values below 0.1 indicate no correlation. Table 4.9 reports a positively strong coefficient for BOINC, DuckDuckGo, and Viber ($0.557, 0, 514, 0.585$, respectively). WordPress achieves a medium correlation coefficient (0.411).

> *The strong and positive correlation between the consensus rankings generated by our approach and the gold rankings formed by app developers demonstrates that the proposed consensus-based approach is effective in prioritising user reviews.*

4.5.2 Qualitative Evaluation

To answer RQ1 qualitatively, and evaluate the performance of the consensus algorithm at providing a prioritised list of user reviews in terms that is meaningful and can aid developers with app maintenance and evolution, we consulted the same developers of the four apps to assess the performance of the generated consensus rankings. We shared with each developer the consensus rankings of the clusters of reviews of their apps and asked them "if your app reviews were prioritised in such a consensus ranking, do you believe this ranking would help

you plan a successful release and would you plan your next release according to this ranking?" We provided developers with five options from which to choose (strongly agree, agree, neutral, disagree, and strongly disagree).

Except for WordPress, developers reported that they "strongly agree" that the consensus rankings of user reviews are meaningful and they would follow the same recommended order when planning a next release. The WordPress developer, on the other hand, responded "neither" and stated that he has his own approach of prioritising new update requests. However, he fully agreed to follow the consensus ranking with minor changes to adapt to his approach. Thus, the qualitative answer to RQ1 is that applying the consensus algorithm can generate meaningful user reviews prioritisation.

> *The four app developers agree that the proposed consensus approach generates meaningful consensus rankings that they would use to plan their next releases.*

4.5.3 Questionnaire User Study

The objective of this study is to gain a better understanding of how developers prioritise issues to consider for the next release and how interested they would be in our approach. To answer RQ2, we perform an online questionnaire user study to collect and analyse developers' answers and opinions.

We started by mining data from 300 apps from Google Play to collect developers' contact information. The mined apps varied between open source, free, and paid apps. Apps that do not provide an email address in the developer information section or provide a generic email address, *e.g.,* `support@...`, `help@...` are removed from the dataset. At the end of the filtering process, we obtained 248 email addresses.

We used Google Form to design and administer the questionnaire. We sent an invitation email to the 248 email addresses explaining our project, the purpose of the questionnaire, instructions to the developers, and a link to the Google Form. Developers were given two months to answer the questionnaire. We received a few bounce-back emails from very popular apps, *e.g.,* Facebook and Google. After two months, we received seven answered questionnaires.

The questionnaire consists of four main parts. In the first part, we gather developers' background information (*i.e.,* app name, role, years of experience, education, gender, and age). The second part investigates the importance of user reviews. In the next part, we ask developers if they categorise and cluster their app reviews, as well as what techniques they

employ. The fourth part focuses on any review prioritisation approaches as well as important attributes (*e.g.,* frequency, average rating, number of devices, severity, category, date). At the end of the questionnaire, we explain our approach in greater detail and ask developers explicitly if they are interested in using it to improve their apps. It should be noted that developers' responses regarding the most important review attributes were used in Section refAttributes1 to help us with the attributes definition process.

After analysing the responses, we discovered that developers have 2 to 8 years of mobile development experience, are between the ages of 20 and 39, and come from various geographical locations (Europe, Mexico, and USA). 85% (6 out of 7) confirmed that they strongly rely on user-submitted reviews for their app's next release. 57% (4/7) reported doing review categorization, clustering, and prioritisation before planning the next release. While 42% (3/7) reported that they only considered urgent and most frequent reviews based on a manual analysis. 57% (4/7) of the developers confirmed using the attributes: cardinality, oldest date, average rating, and category for ranking their apps' reviews. Lastly, 86% (6/7) of the developers showed a great interest in using our approach.

> *The questionnaire user study demonstrated that some developers prioritise user reviews solely on the basis of urgency. Furthermore, developers expressed strong interest in using the proposed method to prioritise user reviews for their apps.*

4.6 Discussion

The quantitative results reported by the two similarity measures are promising. When using the Kendall-τ distance to measure how close or far is the consensus rankings of user reviews from the gold rankings, there is no fixed threshold that the distance between the two rankings should reach. However, we could compute the worst possible distance between two rankings. The further and smaller the distance between the consensus ranking and GR from the worst possible distance, the closer and similar are the rankings. As demonstrated in our results, the distance between every consensus ranking and GR are far from reaching the worst distance, which indicates that the approach is able to produce reliable results close to the manual prioritisation. It is also worth noting that it is expected not to have a very small generalised Kendall-τ distance between the consensus rankings and GRs. That is due to the fact that we are measuring the distance between two different rankings, where the consensus ranking contains ties, while the GRs contain no ties. Besides, we assume that the greater the

statistical correlation coefficient between the rankings, the better the result of the consensus algorithm. Our results showed a very strong correlation coefficient between the consensus rankings and GRs for three of the apps.

To further explain the medium correlation achieved by WordPress, we compared the consensus ranking of the ExactAlgorithm in Table 4.8 and the GR. We can notice a medium variation in the order of the clusters. To further analyse this variation, we contacted the app developer who generated the GR asking him if he believes that his approach of prioritising the clusters of reviews is the best approach. He stated in his email that whenever there is a new update planning, there is a continuous discussion between the app developers on what should be included in the new update. Each developer has a different opinion and in some situations they never reach a consensual decision. As a result, the medium coefficient between the consensus ranking and GR does not doubt the performance of our approach but rather the personal judgement of the developer.

When addressing the qualitative evaluation, we tried in our study to contact more than a developer for each app to obtain a larger evaluation from different software experience perspectives. For the WordPress app, we reached out to many developers, however, we received responses from only two. One of the developers had less than a year of development experience, and he was not fully knowledgeable about the prioritisation process, therefore, his answer was eliminated. We received responses from only one developer for each BOINC and Viber. DuckDuckGo on the other hand, is an app owned by only one developer. The consensus agreement of all developers on the quality of the consensus rankings establishes good evidence that our approach is useful for prioritising user reviews, and could help developers achieve app maintenance and evolution tasks.

Responses from the questionnaire support the importance of considering user reviews during day to day app's improvement activities. More interestingly, most developers confirmed the necessity of analysing, categorising, clustering and prioritising for a more efficient release planning. In addition, they expressed how having an automated tool that can perform these tasks would reduce the amount of manual work. The high interest shown in our approach through the questionnaire responses, highlights the potential of the success of the approach in a real-world environment.

4.7 Threats To Validity

Threats to Construct Validity Building gold rankings to be compared with the consensus rankings requires a high level of accuracy and app development knowledge. To mitigate this

threat, we involved app developers in the creation of the gold rankings. These app developers certainly provided the most accurate gold rankings, however, there is a level of subjectivity in deciding what cluster to address first. More developers for each app are to be involved in building the GR in future work.

The authors are not mobile app developers and thus the defined ranking attributes might not cover all attributes used during review prioritisation. To alleviate this threat, we involved app developers in the selection of attributes through an online questionnaire. Although these attributes are inclusive, other attributes would provide other rankings that could increase the quality of our results. We will systematically study all possible attributes and their impact on our results in future work.

We favoured the use of ExactAlgorithm over BioConcert and Kwiksort algorithms since it is the only exact algorithm that provides the most optimal ranking on a smaller dataset. In future work, we plan to use BioConcert and Kwiksort on larger datasets since they were proven to provide high quality results in such cases [52].

Threats to Internal Validity: This could involve the tool selection to preprocess, categorise and cluster the reviews as there could be a risk of producing low accuracy results. To mitigate this threat, we compared the results of the most outperforming review analysis tools, and adapted CLAP as it was proven to provide the most accurate results. Despite that, CLAP could have its own threat in the employed machine learning and clustering techniques. Therefore, we plan to implement a complete review analysis tool that merges the use of machine learning along with deep learning for higher accuracy.

Threats to External Validity: We are confident that our approach can be generalised and applied to user reviews of any app store and different size of dataset. However, the approach could provide different results when the dataset is extremely large. In addition, we mitigate external validity of our work by providing a replication package [78] that provides all the data used in our approach and its study. From the original user reviews to the consensus rankings as well as the clustered reviews, attribute rankings, and gold rankings. Thus, others can confirm and reproduce our results.

4.8 Conclusion

Mobile apps have become an essential part of everyone's daily lives. Developers must tailor their mobile apps to the needs of their users. Hence, taking user feedback into account during app evolution and planning the next release is critical. However, prioritising user reviews to decide what to address in the next release is a complex task.

In this chapter, we presented a consensus-based approach (Review Prioritiser) to provide apps developers with a prioritised ranking of user reviews. We based the approach on the use of the consensus algorithm (ExactAlgorithm). Developing the approach involved multiple steps. We first preprocessed, categorised, and clustered together related reviews using the user review analysis tool, CLAP. Then, we ranked these clusters of reviews according to a predefined set of review attributes: cardinality, date, average rating, and category. Finally, we applied the consensus algorithm to the set of rankings to generate a consensus ranking of user reviews that can be used by developers when planning their next release.

We used three evaluation methods to evaluate our approach on four Android apps. In the quantitative evaluation, we employed the Kendall rank correlation coefficient to measure the similarity between the consensus rankings and gold rankings. Results showed that there is a strong correlation (average Kendall rank correlation coefficient of 0.516) between the consensus rankings and the manually-prioritised rankings. In the qualitative evaluation, we invited app developers to validate the meaningfulness of the consensus rankings generated by our RP. The developers agreed that the proposed consensus approach generates meaningful consensus rankings that would help plan their next releases. In the questionnaire user study, responses reported that most developers prioritise user reviews solely on the basis of urgency and developers agreed that they would use Review Prioritiser. We thus showed that the consensus-based recommendation technique can be used to recommend mobile app user reviews.

CHAPTER 5 RECOMMENDING TASK INTERACTION TRACE TO GUIDE DEVELOPERS' SOFTWARE NAVIGATION

5.1 Introduction

In this chapter, we apply the consensus-based recommendation technique to developer interaction traces to recommend files(s)-to-edit that could help developers in the development and maintenance of customised systems.

Software companies understand the importance of tailor-made systems that provide client-specific features. Accordingly, these companies develop customised systems that meet their particular clients' needs better than off-the-shelf systems, and are more reliable than completely original systems. Examples of customised systems are Web and mobile applications, e-commerce solutions, CRM (with a global market to reach $50 billion by 2025 [81]) and ERP (with a global market to reach $78.4 billion by 2026 [82]) systems.

To build customised systems, developers fork an original software system into instances, and customise each instance with the desired features and requirements. 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 [83]. Through forking, a software company can create independent instances from the original software system, customise the functionalities of each instance, and add new features in response to client requests.

As clients' requests grow in size, customised systems can expand in size and complexity. 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 [84]. 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 [85]. 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 [86].

Some of the change tasks that developers perform on customised software instances, whether

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

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 source-code file(s) [87]. Hence, we define a (exact and similar) change task as follows:

Definition 5.1.1 *A change task refers to either fixing bugs, improving performance, or implementing new features.*

Definition 5.1.2 *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.*

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 chapter, 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 a task interaction trace from completing each task. Eventually, for each particular change task, developers form a Task-related Set of Interaction Traces (TSITs).

Definition 5.1.3 *Developers' events are generated by developers' activities on source-code elements (i.e., opening, searching, editing, etc.) while completing a change task.*

Definition 5.1.4 *A developer's interaction trace (IT) is a set of events obtained by developers after the completion of a change task.*

Definition 5.1.5 *Task-related Set of Interaction Traces (TSITs) is a set of developers' interaction traces after the completion of similar change tasks.*

Existing works mined interaction traces to recommend files-to-edit based on association rules [86,88,89], or to create exploration strategies and investigate how developers understand programs [90–92]. While these works help developers complete their tasks, none considered studying the development and maintenance of instances of a software system, or providing task-specific recommendations. They only considered software in a isolation and provided recommendations for the entire software system. Moreover, some of previous works do not provide developers with accurate recommendations [93,94]. For example, Lee *et al.* [94] evaluated their approach against Team Tracks [95], Team Tracks recommended three methods, none of which were required for completing the change task. In addition, some of these approaches built their recommendations using association rules between elements frequently edited or viewed together, which lead to recommending unrelated elements if developers interact with the "wrong" elements. Further, most of the proposed approaches rely on data mining techniques which require large datasets for training and evaluation. Finally, previous 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.

To address these limitations, we propose Consensus Task Interaction Trace Recommender (CITR); a task-based recommendation approach that uses the consensus algorithm to recommend file(s)-to-edit based on an aggregated set of developers' interaction traces. We only consider selection and edit types of events for a broad context of input interactions and higher recommendation accuracy [86]. Our approach uses 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 the consensus algorithm to developers' interaction traces, our approach creates a consensus task interaction trace as a recommendation. Each recommendation is a set of relevant file(s)-to-edit and help developers, particularly newly-hired developers, perform new change tasks that are similar to the input task-related interaction traces.

To investigate the success of the recommendation approach, we conduct a series of evaluations. Quantitatively, we determine the accuracy of the recommendation results by defining ground truth data as a basis of comparison and using precision and recall metrics as performance measurements. Qualitatively, we evaluate to what extent gathered tasks interaction traces from previous developers can help developers navigate through software systems and analyse success rate. We also conduct an observational comparative experiment of 30 developers undertaking identical evaluation change tasks with and without CITR recommendations. Lastly, we compare CITR against MI [86], an existing file-level recommendation approach.

The rest of the chapter is structured as follows: Section 5.2, reviews related work. The next section, supports our work with a motivating example. Section 5.4, describes the approach, implements a study to collect developers' ITs and applies the consensus algorithm to generate recommendations. Section 5.5, applies three evaluation methods to investigate the success of the approach. Section 5.6 reports and discusses the results. Section 5.7 discusses limitations. Finally, Section 5.8 concludes the chapter and discusses plans for future works in this area of study.

5.2 Related Work

Previous research related to this area of 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 software engineering recommendation systems; building recommendation systems using interaction traces; and, research studying developers' navigation behaviour.

Use of Interaction Traces for Software Engineering Activities Researchers studied and analysed developers' ITs to ease software engineering daily activities. [91] 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, [96] analysed developers' interaction histories and characterised 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. [97] 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, [98] investigated interaction coupling in interaction histories and found that restructuring is more costly than other maintenance activities. Lastly, [99] 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.

Recommendation Systems for Software Engineering Activities To find system elements relevant to a task that developers are trying to perform, developers have used a variety of tools, ranging from grep to program databases [100]. Recent research studies used different sources of data to help build recommendation systems to improve program compre-

hension, navigation and eventually productivity. HeatMaps [101] is a recommendation tool that computes a Degree-of-Interest (DOI) value based on navigation history, change logs, and execution data. The tool presents artefacts in the IDE in colours ranging from red ("hot") to blue ("cold") according to the DOI value. Hipikat [39] on the other hand forms a group memory using source code versions, bugs, electronic communication, and web documents. It then recommends artefacts relevant to tasks based on inferring links between the archived artefacts in a group memory. [102] could retrieve source code relevant to tasks from clusters of change sets that contain common system elements based on defined filtering heuristics. Both [6] and [88] 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.

Interaction Traces Based Recommendation Systems Several works used developers' interaction traces to recommend or predict relevant system elements. Team Track [95] 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 [89] forms a relationship between system files as a developer browses them, and then recommends files that are relevant to the currently browsed ones. Meanwhile, [90] used interaction histories to build task contexts. Their approach recommends system elements according to the frequency and recency of elements in the context. NavClus [103] clusters sequences of navigation from interaction histories. It uses association rules to recommend system elements that are relevant to the developer's current navigation. [104] used change history data to propose a code completion tool that helps developers complete their change tasks. Switch! [92] was proposed to recommend system artefacts. It bases its recommendation on association between the context of the current task and interaction histories. Code context prediction was proposed by [105]. Based on learned abstract topological patterns from Mylyn interaction histories, the approach predicts code context as developers perform a development task. Similarly, [106] proposed a similar approach called Suade. The approach predicts elements based on the 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 recommendations 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 compare our approach to MI, CITR can recommend file(s)-to-edit that are more relevant to the given change tasks than what MI recommended.

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. [85] 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. [107] 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, [108] conducted a laboratory study with 25 developers. They identified 44 types of questions developers ask during change tasks. Similarly, to understand how developers perform feature location tasks, [109] 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, [110] 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, [111] observed and recorded 10 developers programming activities while performing development tasks to study how developers structure their development effort and whether context impacts the structure. They observed that developers organise 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.

5.3 Motivation

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 tasks 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.

5.4 Consensus Interaction Trace Recommend

5.4.1 Overview of the Approach

We present in this section a brief overview of the concept and steps of our approach, Consensus Interaction Trace Recommender (CITR). Figure 5.1 illustrates how the recommendation approach can be incorporated into the motivating use case (Section 5.3). 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 these tasks. In this study we consider both types of events; selection and edit. Including selection and edit 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. [86].

Collected events from performing each task are extracted and go through multiple preprocessing steps for noise removal. Preprocessing includes steps like eliminating Mylyn trigger events, JAR files and noise events. Once these events are preprocessed, 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 of 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 motivating use case in Figure 5.1. 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' software instances using Eclipse. The developers perform 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 a task-related set of interaction traces (TSITs) for change task T. This aggregation of developers' ITs into task-related sets 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

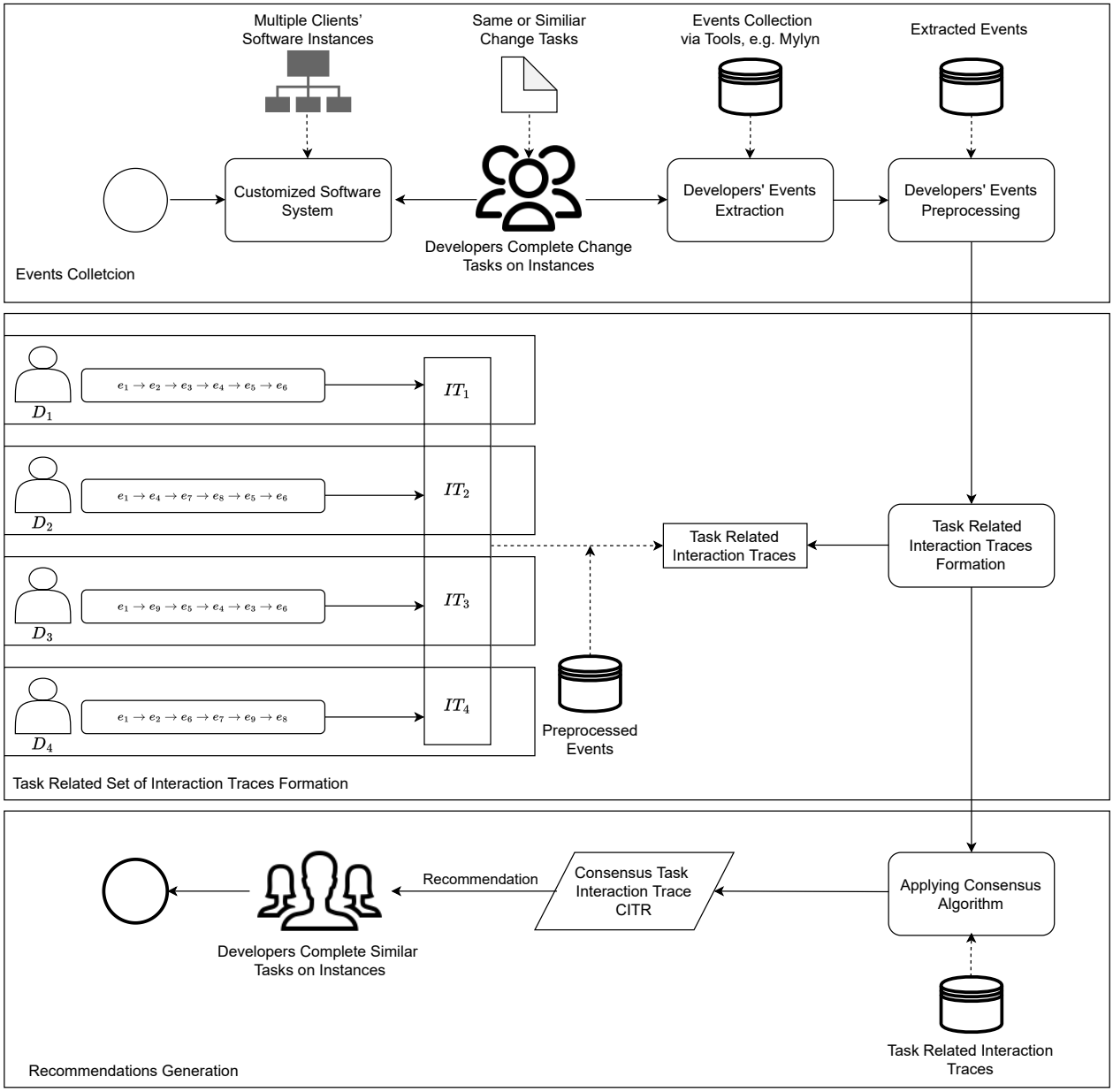


Figure 5.1 Overview of the Approach Concept

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.

5.4.2 Study Setup

We carry out in the next Section three evaluation methods to assess the accuracy of the results of the 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*:** To what degree does CITR recommend relevant files to given change tasks? We answer this question by building ground truth data and quantitatively comparing them with the results of CITR using precision and recall measures (in Section 5.5.1).
>
> ***RQ2*:** Given a change task, can CITR help guide developers' navigation paths to relevant file(s)-to-edit and increase their productivity? To evaluate productivity and navigation behaviour, we conduct an observational comparative experiment of 30 developers performing evaluation change tasks with and without the recommendations of the CITR, and compare their behaviours (in Section 5.5.2).
>
> ***RQ3*:** How does CITR compare to MI (Mining Programmer Interaction Histories) in recommending relevant file(s)-to-edit for specific change tasks? We answer this question by comparing the results of our approach with MI recommendation results under the same set of conditions (Section 5.5.3).

To generate the recommendations, carry out the evaluations and answer these RQs, we conduct an experiment of participants performing input change tasks to collect their events. Collected events from the experiment are then extracted, preprocessed and used as input data to generate recommendations.

Unlike previous studies, such as [6] and [86], that extracted and used archived developers' events as input 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 which is the same system that will be used for the evaluation (in Section 5.5). Then, we choose change tasks, recruit participants, and select tools for collecting the events. Next, we invite the participants to conduct the experiment by performing the change tasks and collect their events with the software elements. After collecting and extracting the events, we preprocess them. Finally, we form task-related set of interaction traces (TSITs) and apply the consensus algorithms on these TSITs to obtain recommendations.

1. Subject System

Among a population of Java systems, we chose an Eclipse-based plugin, PDE (Plug-in Development Environment), as the 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 many sub-projects and we chose to consider only the PDE-UI sub-projects. Participants will interact with PDE-UI files to complete change tasks.

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, and request 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.

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

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

To measure the complexity and time needed to complete each of 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. Tickets that required minimal navigation and comprehension effort and a few minutes to complete were marked as easy. The evaluator, on the other hand, rated moderately difficult tickets as those that required more cognitive effort to understand, navigation through several files, and 20 minutes to an hour to complete. Lastly, he classified any ticket that took him longer than an hour to complete or that he was unable to perform as highly difficult.

To the best of our knowledge, there is no theory that suggests the optimal length of the experiment task. However, in order 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 achieved this by randomly selecting two moderately difficult tickets as our change tasks that should not take longer than 45 minutes to complete, with an additional 15 minutes if necessary.

Authors of [112] 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 [112], adapt it as our third change task and use the collected events as part of our dataset. Table 5.1 presents detailed descriptions of the chosen change tasks.

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

Table 5.1 Change Tasks Used in the Study and their Descriptions

3. Participants

Considering the size and complexity of PDE-UI, we recruited only participants with experi-

ence with 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.

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

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

4. 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 [113], FeebBaG [114], or DFlow [115]. 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 [112].

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

[4]Online Form
[5]`https://www.eclipse.org/`
[6]`http://eclipse.org/mylyn/`

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 5.2.

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

Table 5.2 an Example of Mylyn Events

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 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 plugin installed.

Before they began performing their change tasks, we created, in the IDE, for each participant, a Mylyn task for each change task. As shown in Figure 5.2, on the left side, the PDE system is imported to the IDE and Mylyn tasks are created and ready to be activated under Task List on the right side.

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

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

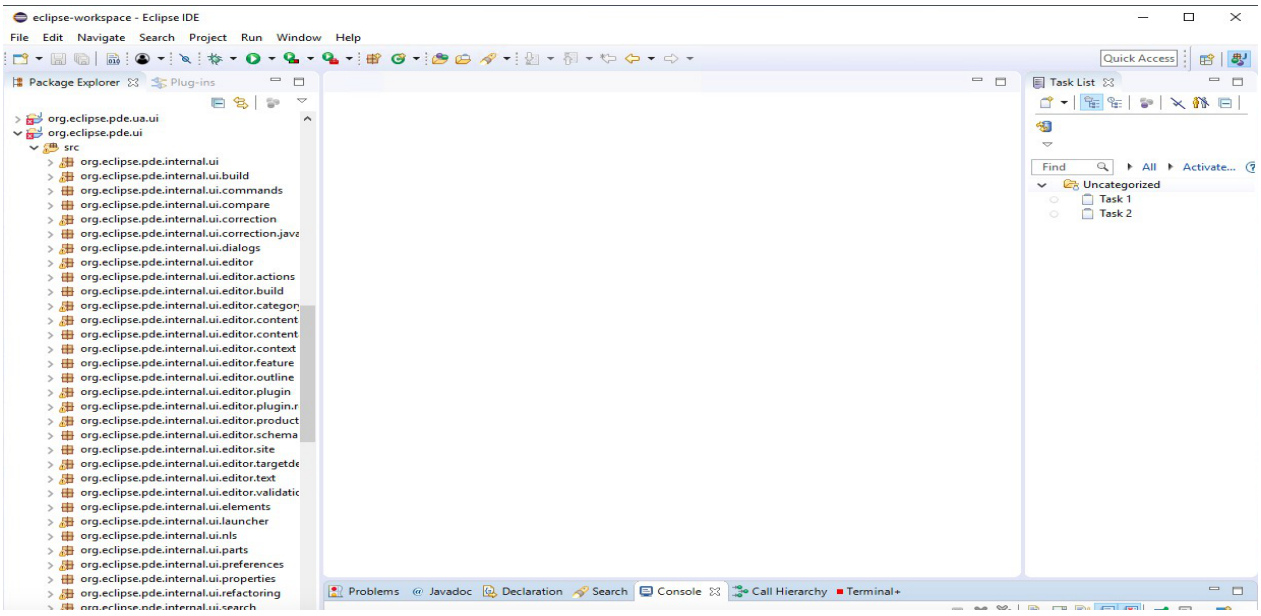


Figure 5.2 A Screen Capture of the IDE Before the Start of a Change Task

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 in the study repository [116]. To provide the participants with a hint of what they had to do and type of tasks, we created a demo change task. We gave the participants 20 minutes to replicate the bug described in the demo task, and they were not required to provide a solution for the task.

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

We then exported all Mylyn events from Eclipse IDE in XML format. Obtained events from the completion of Change Task 1 and 2 by seven participants, together with events related to Change Task 3 from [112] by four participants sum a total of 2,390 events. Figure 5.3 represents a sample of an extracted Mylyn event.

6. Events Preprocessing

We preprocessed each exported Mylyn event to extract participants' selection and edit activities and system elements on which the activities occur. This phase goes through multiple

```
<?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
```

Figure 5.3 Sample of Mylyn Event in XML Format

steps and starts by converting the extracted Mylyn XML files into CSV files.

As described in Step 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 [117]. Figure 5.4 shows the parts of the StructureHandle against a real StructureHandle taken from one of the participants' Mylyn events, while Table 5.3 identifies the parts of the StructureHandle.

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~c
onfigureShell~QShell;

Figure 5.4 Parts of Mylyn StructureHandle.

The paths to elements in the StructureHandle contain special characters that created noise and made it difficult to obtain the actual full paths. We implemented a tool that uses regular expressions to identify the parts of StructureHandle and either remove or replace these special characters with dots. The tool outputs a readable CompleteName for each StructureHandle

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

Table 5.3 Identification of the Parts of the StructureHandle in Figure 5.4.

that contains no special characters. Figure 5.5 compares a path to a system element as exported in the StructureHandle versus the path CompleteName after the removal of special characters.

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

Figure 5.5 StructureHandle vs CompleteName after Special Characters Removal.

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 [86] and [97], 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 preprocessing, 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 5.6 compares two screenshots taken from the interaction files of participant $P2$ and $P4$ for Change Task 1. Events 26 and 27 were performed by the two participants on exactly the same StructureHandle, hold the same type,

and accordingly are assigned the same ID number.

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

Figure 5.6 Illustration of Common Events between Participants Hold the Same ID Number.

Mylyn events relate to two levels: method-level events and file-level events. Method-level events occur in/on classes, fields, and methods. File-level events occur on Java files, such as opening or editing a file. Considering that our approach aims to recommend file(s)-to-edit, we therefore keep only file level events and remove those on method level. All collected and preprocessed participants' events that are used to generate CITR recommendations are available online [116].

7. Task-Related Interaction Traces Formation and Generating Recommendations
Each participants' 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 $(IT1, ITs, ...IT7)$. Each set of participants' ITs from performing a change task forms a task related set of interaction traces (TSITs). Given that the study experiment involved three change tasks, thus we were able to generate three TSITs.

Considering that the BioConcert and KwikSort algorithms provide best quality results (see Section 3.1.4 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 normalisation technique to complete the rankings (discussed in Section 3.1.3). 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. To illustrate, assume we have interaction traces of participants $P1$ and $P2$:

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

The application of the unification process produces the following ITs:

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

Table 5.4 shows consensus task interaction trace after applying both algorithms on a 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 minimises the disagreement between the set of input ITs and groups the less relevant files in a single bucket at the end of the consensus.

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

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

Table 5.4 CITR after Applying BioConcert and Kwiksort to TSITs of Task T.

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

Table 5.5 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 project repository [116].

5.5 Evaluation

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

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

Table 5.5 Translation of Recommended Consensus Interaction Trace from Applying BioConcert into Real Participants' Events.

5.5.1 *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 the last Step of Section 5.4.2 against ground truth data.

A ground truth data in this study 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 three change tasks (Section 5.4.2 - Step 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 extract these attached Mylyn events, we preprocess these events to remove any possible noise by following the same preprocessing steps that we adopted in Section 5.4.2 - Step 6. Following that, we extract a set of files from the set of preprocessed 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 5.6 presents the ground truth data created for Change Task 3. The remaining ground truth data for the other two tasks are available online in the project repository [116].

To measure the accuracy of the recommendation approach results and answer RQ1, we compute the quantitative Accuracy Metrics discussed in Section 3.2.5; Precision, Recall, and F-measure.

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

Table 5.6 Ground Truth Data Created for Change Task 3.

5.5.2 *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 User Study - Between-Subjects experiment (discussed in Section 3.2.5) with 30 developers performing a set of evaluation change tasks. The following details the experiment's procedures.

Setup. We use the same system, PDE-UI, as in (Section 5.4.2 - Step 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 change tasks that developers in this experiment perform to evaluate whether CITR results can help them complete the tasks and increase their productivity. Specifically, we choose evaluation change tasks that are similar in context to the change tasks that were used to generate the recommendations (Section 5.4.2 - Step 2).

To obtain evaluation change 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) attachments contain 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 ticket description from the search query results. We read through the descriptions to identify tickets that share the same context as the three change tasks. To illustrate, Change Task 1 is about issues related to the plug-in contents, Change Task 2 is related to issues on the tabs of the launch configuration window, while Change Task 3 is concerned with plug-in configuration errors. We therefore find tickets that are concerned with these three issues.

In the third phase, two authors randomly chose tickets from the second phase, performed them, assessed their complexity, and divided them into three categories: easy, moderate, and difficult. For equitable evaluation, we targeted moderate complexity 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 requested two Ph.D. students with prior professional experience but no background in the subject system to perform the tickets to confirm their complexity and if they could finish them in 30 minutes.

Finally, with the assistance of the Ph.D. students, we considered six evaluation change tasks that are similar in context to the change tasks. We describe them in Table 5.7.

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

Table 5.7 Evaluation Tasks Description.

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 5.4.2 - Step 3).

We sent out invitation emails to software engineering research groups from four universities (Concordia University, Polytechnique Montréal, Zürich University, and Zürich University of Applied Sciences). The email provided them with a registration form to gather relevant ed-

ucational 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 selected 34 developers (referred to as $D1$,..,$D34$), out of whom four abandoned the experiment. Of these 30 developers, five were senior undergraduate students, 15 were M.Sc. students, and ten were Ph.D. students. 53% 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 (six of them with more than 5 years, while the remaining 24 had between 1 to 5 years). All developers reported using different IDEs and being unfamiliar with the subject system. We provide detailed participant demographics in Table 5.8.

Education	
Bachelor	5
Masters	15
Doctorate	10
Programming Experience	
5 Years or More	17
3-5 Years	6
1-3 Years	7
0-1 year	0
Java Experience	
5 Years or More	2
3-5 Years	4
1-3 Years	10
0-1 year	14
IDE Experience	
5 Years or More	5
3-5 Years	5
1-3 Years	8
0-1 year	12
Professional Experience	
5 Years or More	6
3-5 Years	7
1-3 Years	8
0-1 year	9

Table 5.8 Demographics of Selected Developers

Procedure. We applied the Between-Subjects experiment design in which there is a control group and an experimental group, and each participant experiences only one level of a single independent variable. Our independent variable is the recommendations with two levels: with and without. We split the 30 developers into a control group and an experimental group (15

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

To ensure that the set of evaluation tasks are performed by developers with different experience levels, we split the six tasks into two sets (A and B with three evaluation change tasks in each set). Therefore, each control and experimental group was further divided into two sections, with each section performing a different set of evaluation tasks. Figure 5.7 illustrates this division. Furthermore, we chose to have the two groups of developers conduct the experiment on the same primary 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.

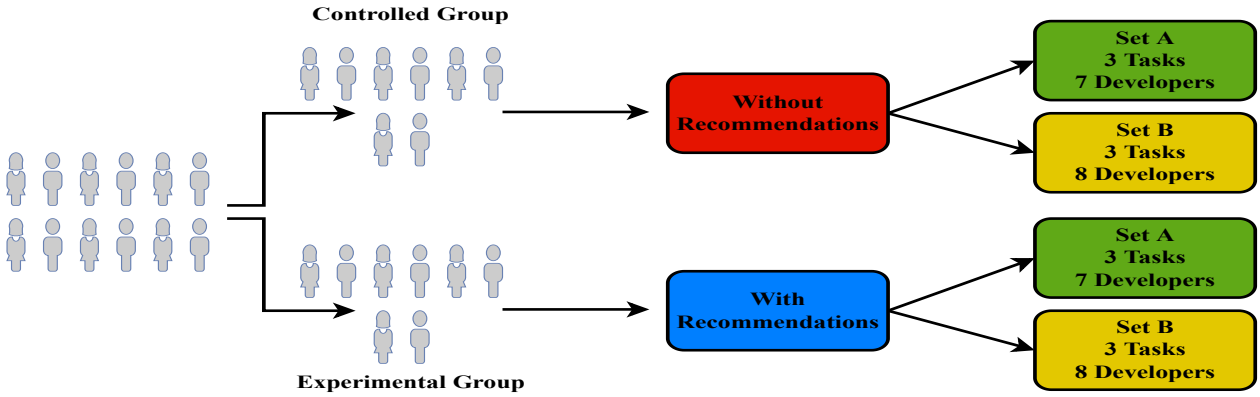


Figure 5.7 Divisions of the Developers into Groups and Sections.

The evaluation experiment took about four months to be completed due to COVID-19 restrictions. 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 task description documents recommended the consensus tasks interaction traces along with each task.

Once a developer was remotely connected, we gave each of them up to 10 minutes to explore the system, get familiar with Eclipse and the system structure. In addition, we provided them with a practice task to 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 first 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 second 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. However, we allowed developers in both groups to ask for five to ten extra minutes if needed, and told them not to be concerned if they did not accomplish a task and move to the following one.

When developers completed locating file(s)-to-edit for each evaluation task, we requested them to fill their answers in an answer sheet. We then stopped the video recording, and disconnected the remote access. Afterwards, we interviewed the developers and sent them a post-experiment questionnaire to gain their insight about the approach and the experiment in general, which will be discussed in the following section.

Measures. To study whether the 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 task.

In addition to measuring the ability to complete the evaluation tasks, we investigated the

effect of the recommendations on developers' behaviour and navigation paths. We carefully watched and analysed the video recordings of the 30 developers to study their navigation patterns and compared the patterns between the two groups. During the observation, we paused the videos whenever necessary to take notes and followed the mouse pointer's route on the screen to determine 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 behaviour and limit the number of consulted irrelevant system elements.

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

The questionnaire consisted of nine questions: from assessing the difficulty of dealing with unfamiliar systems, to difficulty locating related files using the help from the approach, to the relevance of the recommended files, and to whether or not the approach helped improve their performance completing the evaluation tasks. Six of the nine questions were rating scale questions, two were yes/no questions, while the remaining two were open-ended questions. The questionnaire is presented in Table 5.9.

5.5.3 *RQ3*: How does CITR compare to MI (Mining Programmer Interaction Histories) 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 [89], MI [86], Mining Change History [88], and NaCIN [118]. 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 then a query formed from the current developer's interaction with a given task at the time of recommendation [86].

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

Table 5.9 Post-Experiment Questionnaire.

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

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

both *AND* and *OR* operations, and generates recommendation at selection and edit events.

Procedure. We simulated file(s)-to-edit recommendations in MI using the interactions generated from the three change tasks in (Section 5.4.2 - Step 2). According to [86], applying different context creating methods, generates different recommendation results. Further, their results showed that methods with the AND operation yield higher recommendation accuracy. Therefore, we used methods with the AND operation: MI-EA, MI-VA, and MI-VOA.

Regarding the size of the sliding window, the authors of MI spread the *v-e* sized sliding window between 1 and 10. Considering that the size of our dataset was smaller than the dataset used in the MI evaluation experiment, applying MI to our dataset with a sliding window size greater than 4 did not generate any recommendations. Hence, we varied the *v-e* sliding window size from 1 to 4. In general, developers viewed 79 files for every one they edited [86]. Therefore, we set the number of *(v)* to 4 and the number of *(e)* to be less than 4. For each *v-e* value, we ran the simulation repeatedly over all the participants' interaction traces.

Measures. We evaluated MI recommendation results against 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 Section 5.5.1 as a baseline for the comparison.

5.6 Results & Discussions

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

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

Table 5.10 presents the precision, recall, and F-measure values that result from comparing the accuracy of the recommendations of CITR to the ground truth data from the three change tasks.

The precision and recall values of CITR recommendations generated from developers' interaction traces of Change Task 1 and 3 are encouraging. Results from both tasks generate a precision value of 1, meaning 100% of the time CITR produces recommendations that are

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

Table 5.10 Precision, Recall and F-measure Values of the Recommendations Accuracy.

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

To investigate the reasons behind the 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.java`), 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.

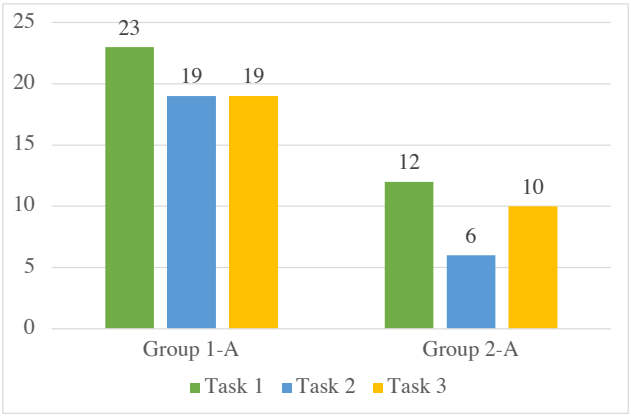
Considering that these two files are irrelevant to the change task, none of the participants made any kind of interactions on them while performing the task. Consequently, CITR did not recommend these files and instead recommended other files based on the navigation of all the participants who completed 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 statistically answered the first research question that considers the quality performance of CITR. Overall, CITR results achieved high average precision (72%), recall (61%), and F-measure (60%).

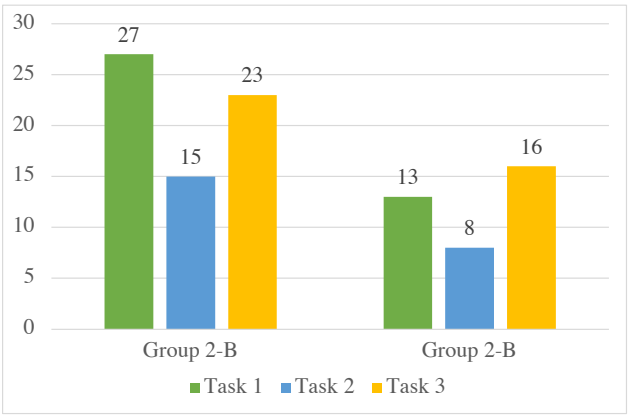
RQ1: CITR achieves high precision, recall, and F-measure and is able to recommend accurate and relevant file(s)-to-edit.

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

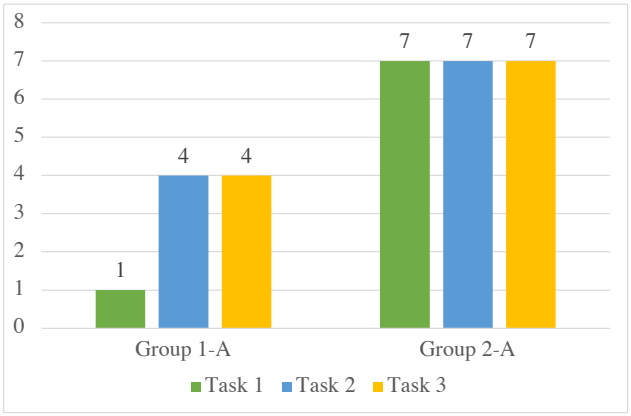
Developers Success Level Figures 5.8a and 5.8b compare the average time (in minutes) spent on each evaluation task from set A and B by the control group (1) (developers who completed the task without CITR recommendations) to the experimental group (2) (developers given CITR recommendations), while Figures 5.8c and 5.8d present the numbers of developers from each group who completed each evaluation task in the two sets.



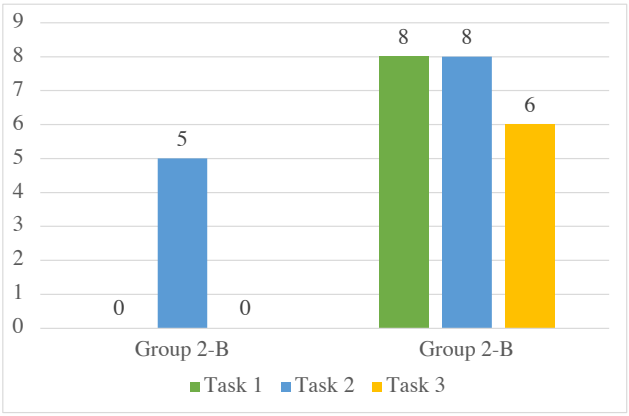
(a) Average Spent Time on Tasks in Set A



(b) Average Spent Time on Tasks in Set B



(c) Number of Completed Tasks in Set A



(d) Number of Completed Tasks in Set B

Figure 5.8 Average Spent Time and Number of Completed Tasks in Both Sets by the Two Groups.

Looking at the average completion time of each evaluation task, we observe that the control group spent on average about twice as much time as needed by the experimental group. In task set A, developers with recommended file(s)-to-edit needed 46%, 68%, and 47% less time to perform each task, respectively, in comparison to developers in the control group. Similarly, those developers in the experimental group could complete the tasks in set B in 52%, 47%, and 30% less time than those in the control group. Detailed tasks completion time by developers from both groups is available in the project repository [116].

Developers from both groups spent higher average time performing Evolution Task 1 in the two sets. Notwithstanding that the developers were given a practice task before the start of the experiment to familiarise themselves with the experiment, navigation through random classes was a significant component of Evolution Task 1.

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

Regarding completion factor, Figures 5.8c and 5.8d show that, in both task sets, results from the experimental group present a higher task completion rate than the first group. Examining completion rate of tasks in Set A reveals that among the seven developers in the experimental group, the completion rate was 100% for the three evolution tasks. Conservatively, only one out of the seven developers in the control group completed Evolution Task 1 and four completed Evolution Tasks 2 and 3. For Set B, the overall completion rate of the experimental group is higher in comparison to the control group. Particularly, files related to the completion of Evolution Tasks 1 and 2 were identified successfully by all the eight developers in the experimental group, while six of them identified the files related to Evolution Task 3.

None of the developers in the control group had any success with Evolution Tasks 1 and 3, while five of them completed Evolution Task 2. We expected this very low completion rate considering that developers had no prior system related knowledge. We noticed that only two developers from the experimental group could not identify the files related to Evolution Task 3. We hypothesise 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 for 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 experimental group. To illustrate Table 5.11 shows the demographics of developers in the group and the time in minutes took some of the developers to complete tasks in 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 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.

The experimental group, on the contrary, followed the same navigation approach. All developers started performing the tasks by navigating to every recommended file, reading through

Experimental Group	
Education	
Bachelor	3
Masters	7
Doctorate	5
Programming Experience	
5 Years or More	10
3-5 Years	2
1-3 Years	3
0-1 year	0
Java Experience	
5 Years or More	1
3-5 Years	2
1-3 Years	6
0-1 year	6
IDE Experience	
5 Years or More	4
3-5 Years	1
1-3 Years	5
0-1 year	5
Professional Experience	
5 Years or More	4
3-5 Years	3
1-3 Years	4
0-1 year	4

(a) Demographics of Experimental Group

Experimental Group			
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

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

Table 5.11 Developers' Experiences Result in Different Tasks Completion Time.

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

through irrelevant search results.

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

With the experimental group, our observation reveals that developers depended heavily on using the search dialog function to search for keywords related to fixing the bug at hand in the recommended files only to determine the needed file(s) to complete the tasks. Considering that the CITR recommends relevant files and the searched keywords could possibly appear in most of the files, developers used their own judgement and programming experience and spent limited effort comprehending methods that they believed to be relevant to the tasks. When developers were required to locate file(s)-to-edit that were not part of the set of recommendations, in the case of Evolution Task 3, they followed the relevant methods' cross-reference to locate these files. From these observations, we found that the experimental group could apply a more 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 [119]. The findings of our observational experiment demonstrated that using developers' interactions with the system can enhance navigation by resulting in a more structured behaviour, 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 5.12.

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

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

Table 5.12 Post-Experiment Questionnaire Answers.

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

All developers strongly agreed that 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: 80% stated the recommendations were very relevant. 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.

> *93% 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 treated the set of recommendations as entry points to the system which saved them from randomly hunting through the package explorer.

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

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

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

5.6.3 *RQ3*: How does CITR compare to MI (Mining Programmer Interaction Histories) in recommending relevant file(s)-to-edit for specific change tasks?

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

Based on the results, we measure the effectiveness of CITR by comparing the statistical values of CITR to MI.

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

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

Table 5.13 presents the recommendation results of applying *MI-EA* to interaction traces of Change Task 1 under the three sets of sliding window along with CITR recommendation of the same task. Unsurprisingly, as the value of the sliding window decreases, MI makes recommendations on more edit events and hence recommends a higher number of files-to-edit. We notice that when (v=4, e=2), *MI-EA* recommendations do not intersect with CITR recommendations nor ground truths, while the opposite is true in the case of (v=4, e=1) and (v=3, e=1). This observation suggests that the quality of the recommendations by MI depends greatly on the value of the sliding window and there is no a specific set of values that will guarantee high quality results as the number of events is always changing. CITR provides quality results without the need of defining a sliding window.

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

Table 5.13 Simulation Results of *MI-EA* and CITR Recommendations from Change Task 1.

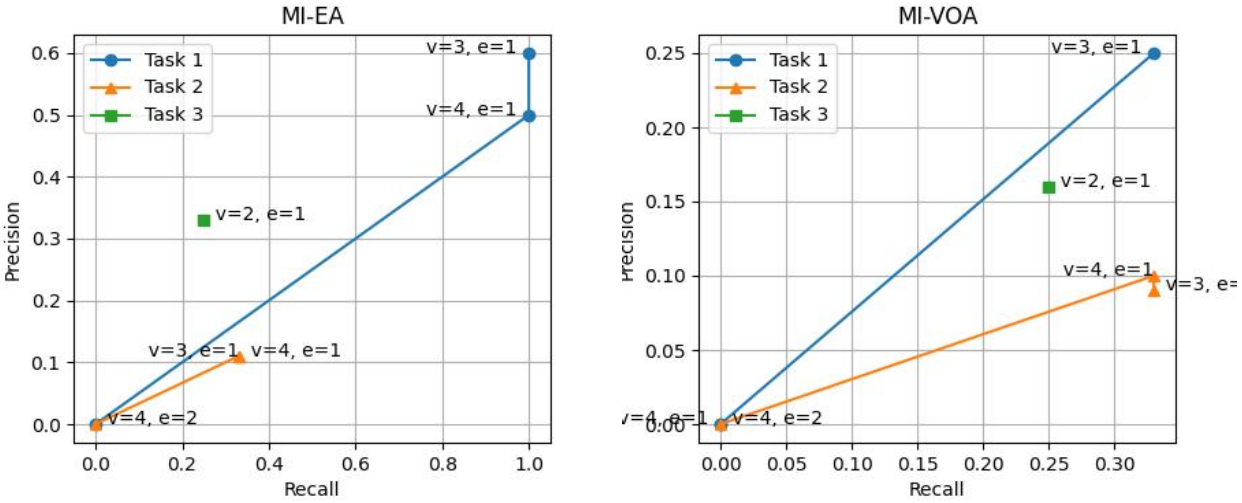
Table 5.14 compares the recommendation results of *MI-EA* with *MI-VOA* methods under the same set of sliding windows. *MI-VOA* with different sliding-window values produced nearly the same recommendations of files-to-edit. We also found that none of the recommended files intersect with CITR recommendations, except with (v=3, e=1). In addition to the previous observations, the use of the OR operation cannot provide high quality results with our dataset due to the small number of edit events, which we will assess further statistically. The rest of the results are available online [116].

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

Table 5.14 Results of MI-EA Against MI-VOA from Change Task 1.

Given that *MI-EA* provided better quality recommendations over *MI-VOA* in the case of our dataset, we further investigate the accuracy statistically by comparing precision, recall, and F-measures values of the results of both methods. We computed the measure values using the set of results from both methods and the set of ground truth. Figure 5.9 presents the resulting precision and recall curves of the recommendation results from *MI-EA* and *MI-VOA* under different (v-e) sliding windows. The figure shows recommendations from Change Task 1 using *MI-EA* consistently achieved higher precision and recall than using *MI-VOA* when (v=4, e=1) and (v=3, e=1). Analogously, *MI-EA* showed higher precision and exact recall values under the same sliding windows for Change Task 2. Comparing results of *MI-EA* to *MI-VOA* from Change Task 3, the former also showed higher precision and same recall values. In terms of F-measure, we took the average values from the three sets of (v-e) sliding windows for each change task. *MI-EA* performed an average F-measure of 0.47, 0.1, and 0.13 for each task respectively. Whereas *MI-VOA* performed lower F-measure from Change

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



(a) MI-EA Precision and Recall

(b) MI-VOA Precision and Recall

Figure 5.9 Precision and Recall of Recommendations from MI-EA and MI-VOA.

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

Figure 5.10 presents the precision and recall curves of the recommendation results of MI against CITR. As shown, examining the results of all the three change tasks, CITR recommends files-to-edit with precision values of 1, 0.17, and 1 and recall values of 1, 0.33, and 0.5, respectively. On average, MI yielded lower accuracy results with 0.36, 0.07, and 0.33 precision and 0.66, 0.22, and 0.25 recall, respectively. Consequently, CITR significantly outperforms MI in terms of F-measure values. As shown in Figure 5.11, our approach shows F-measures values of 1, 0.15, and 0.66 for each change task respectively. Whereas MI performed average

accuracy at 0.47, 0.1, and 0.13 of F-measure values.

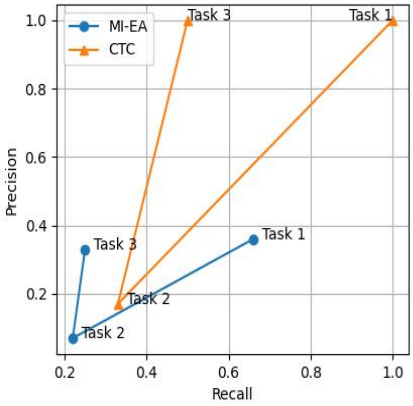


Figure 5.10 MI-EA and CITR Precision and Recall Curves

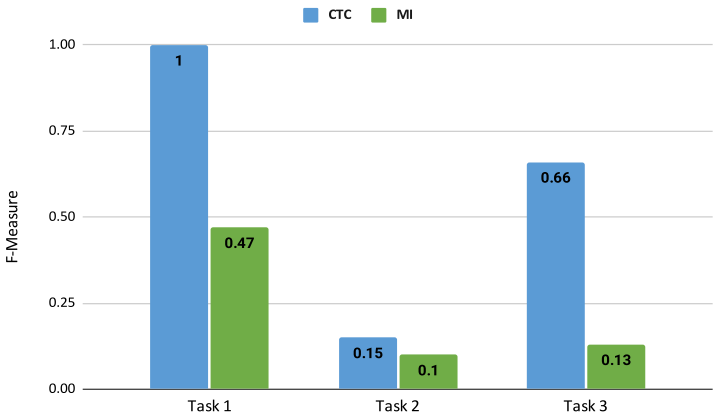


Figure 5.11 CITR and MI F-Measure Values

To better understand how CITR provided higher recommendation accuracy than MI, we analyse how the approaches work along with the recommendation results. To illustrate, in Table 5.13, CITR recommends `pderesources. properties`, `PDEUIMessages.java` and `VersionDialog.java` out of the interaction traces of Change Task 1, which precisely match the suggested files in the ground truth data. MI recommended three other files that are irrelevant to the context of this task. CITR can recommend more relevant file(s)-to-edit with less noise than MI. The technique used requires MI to specify the size of the context that will trigger the recommendation. As the *v*-*e* value changes and the sliding window moves, the context gets constantly updated, which affects what files to consider and eventually yields irrelevant files. With CITR, there is no need to specify a particular set of context to trigger

recommendation as the approach treats the whole set of interaction traces as one context and extracts the most relevant files based on the idea of the consensus.

Many recommendation tools [88, 89], 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 a small set of interactions that are associated together as one context. In essence, it combines and treats all developers' interaction histories as one task context, finds a set of consensus files among all the files, and recommends them to help completing other similar tasks. Ability to treat all interaction traces as one context for recommendations explain why 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 to get the tool to start making recommendations.

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

5.7 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 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 three different change tasks from a large open source system in the design of the ITs collection (in Section 5.4.2 - Step 2), completed by participants with various educational and industrial backgrounds, and using a common IDE and programming language, Eclipse and Java.

Mylyn Noise This threat is related to the tool used to collect participants' events, Mylyn Eclipse plugin. Mylyn introduces some noise, such as time-related noise, edit-related noise, duplicated events, or missing events. We implemented a preprocessing 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 three recommendations (consensus task interaction trace) from three input change tasks by applying the consensus algorithm to an input of four to seven 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 comparative experiment (in Section 5.5.2) from a laboratory experiment to a remote experiment. We could not control interruptions, which could impair developers' navigation behaviour and productivity. We could only ask developers to perform the experiment in a quiet environment, record their screens, and audio-call them using Zoom conferencing software.

Reliability We make all data used in this study available online in a public repository for replication purposes [116]. To increase the reliability of our results, we employed multiple measures: precision, recall, and F-measure for quantitative evaluation; an observational comparative experiment with video observation analysis, post-experiment interviews, and questionnaire for the qualitative evaluation; and, a comparison with an existing approach.

Measures Considering that our approach produces a set of consensus file(s)-to-edit with

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

Observation Bias We based the qualitative findings of developers' behaviour in the observational comparative experiment (in Section 5.5.2) 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.

5.8 Conclusion

In large, customised software systems, the successful completion of change tasks requires developers to understand elements that are scattered across the system. Customized systems increase in size and complexity as client demands increase. The growing complexity makes finding and understanding the subset of elements part of a change task challenging and require more time and effort, especially when the developers are newly-hired.

In this chapter, we proposed an approach, Consensus Task Interaction Trace Recommender (CITR), to recommend file(s)-to-edit. CITR builds recommendations by applying the consensus algorithm (BioConcert) to the set of developers' interaction traces. CITR can recommend relevant file(s)-to-edit that help developers, particularly newcomers, complete change tasks that are similar to the input tasks with minimal effort and time.

We evaluated our approach using a series of evaluations: quantitative, qualitative, and comparison. In the quantitative evaluation, we measured quantitatively the accuracy of the recommendations against ground truth data. In the qualitative evaluation, we carried out an observational comparative experiment to measure the extent to which recommendations could increase developers' productivity. Lastly, we compared our approach to a state-of-the-art approach, MI [86]. Results showed the followings:

- Quantitative results indicate that our approach can recommend relevant file(s)-to-edit with average precision of 72%, recall of 61%, and F-measure of 60%.
- A detailed qualitative analysis of the experiment supported by video recordings reveals

that developers with CITR recommendations can complete their tasks in/with less than half of the time and effort needed by the control group, and with higher completion rate of 95%.

- The experiment shows that developers with CITR recommendations have an increased performance at comprehending and navigating through system elements. In contrast, the control group spends a considerable amount of time following unstructured navigation relying on guessing and glancing.
- The comparison demonstrates that CITR returns higher recommendation accuracy than MI with average F-measure value of 60% and 20% respectively.

We concluded that CITR can guide developers' navigation path towards resolving tasks and increasing their productivity. We also proved that the consensus algorithm is an efficient recommendation technique to recommend relevant file(s)-to-edit.

CHAPTER 6 FINDING COMMON GAME ENGINE ARCHITECTURE SUBSYSTEMS AND THEIR COUPLING DEGREE

6.1 Introduction

In this chapter, we evaluate the effectiveness of using the consensus algorithm at recommending a consensus game engine architecture subsystems.

The history of video games began around six decades ago with computer scientists developing Spacewar in 1961 [120]. Since then, the video game industry grew and has become one of the most profitable markets. By the end of 2022, the global gaming industry will reach over 3 billion people and generate over $196.8 billion in revenue [121].

Video game development was originally done by a small development team writing code from scratch and rarely reusing core or subsystems. However, as the industry grew and its environment became highly competitive, users' expectations have grown, calling for advancements in game development techniques. These changes have created the need for a framework that can facilitate and shorten the development time by providing generic, reliable and reusable software subsystems such as a rendering engine, physics engine, audio system, *etc.,* so the developers could focus their effort on developing the game mechanics, which are the game's logic. Such frameworks are known as game engines. Examples of popular game engines are Unity engine[1], Unreal engine[2], and CryENGINE[3]. Most game engines are written in C++ [122].

The problem in game engines development is that developers do not always design and create an architectural model before coding. Hence, architectural decisions are only represented in the code and there are no available architecture designs that are readily available for analysis and comparison by game engine developers we designing an new engine. Such comparisons could be useful for identifying commonalities and suggesting ways to improve existing and future game engines [122].

In this chapter, we fulfill three objectives: determine architectural commonalities between game engines, create a model that presents a consensus of fundamental subsystems, and identify the degree of coupling of these subsystems. To achieve these objectives, we propose COnsensus Software Architecture (COSA), an approach based on applying the consensus

[1] www.unity.com
[2] www.unrealengine.com
[3] www.cryengine.com

algorithm to a set of game engine architectures. COSA can recommend a ranking that suggests the most commonly used subsystems in game engine architectures, ranked by their degree of coupling. The recommended ranking can be used by developers to decide which subsystems to include when designing a game engine architecture, as well as to support game engine reusability and maintenance by identifying the most coupled subsystems, allowing developers to focus their programming efforts on minimising coupling.

To investigate the meaningfulness of the generated recommendation, we evaluate the consensus ranking by comparing it to predefined ground truth data. Results show that all identified subsystems are fundamental and should be taken into account when designing an engine architecture. Moreover, results show that engine architectures contain similar subsystems. Lastly, they show that the most coupled subsystems are core systems, low-level rendering, third-party SDKs, and world editor.

The remainder of the chapter is organised as follows: next Section discusses related studies. Section 6.3, presents an overview of architecture in game engines. Section 6.4 explains the proposed approach. Section 6.5 discusses results. Section 6.6 presents possible threats to validity. Finally, last Section draws the conclusions of the chapter.

6.2 Related Work

Game Engine Architecture Gregory in his book [2] covers the huge field of game engine architecture succinctly. It discusses in depth the possible components and subsystems that make up a game engine and the architectural layers. Furthermore, it provides developers with the fundamentals of designing a game engine architecture. Rollings *et al.* [123], on the other hand, discuss the overall design of game engine architecture without delving into the individual subsystems. The book outlines characteristics that make a perfect architecture: Modularity, Reusability, Robustness, Trackability. In a similar sense, Marin al [124] propose a multi-agent system (MAS) for game engines that can prototype the engine architecture in a fast way. MAS archives that by specifying the subsystems that define the game engine. The proposed architecture increases engine performance and eases the process of comprehending the mechanisms of the game engine. Besides, SMASH (Stackless Microkernel Architecture for SHared environments) [125], another proposed methodology for designing game engine architecture. The proposed methodology decomposes the architecture in dynamic software modules interacting via a microkernel-like message bus. The design allows inserting, debugging and removing the modules right after defining the internal messaging protocol of the engine. The approach has been proven to increase the scalability and flexibility of game engines. Furthermore, [126] addresses the lack of research studies regarding game engine

architecture by engaging the academic community and game industry in answering survey questions and sharing their knowledge about the topic. They determined that constant technology changes affect the evolution of low-level components of an architecture, and identified best practices for game engine development.

Software Architecture Recovery The lack of assessed and validated architecture recovery techniques, led the authors of [127] to perform a comparative study of six architecture recovery techniques to determine their abilities at identifying architectures' subsystems and structure. The study results revealed that a significant amount of improvement can be made in all of the techniques examined. Additionally, [128] performed components and dependencies identification and structures extraction through the architecture recovery process using the Rigi system for the purpose of creating higher-level abstract representations of a subject system. Similarly, Wang *et al.* [129] proposed an improved hierarchical clustering algorithm called LIMBO Based Fuzzy Hierarchical Clustering (LBFHC) for conducting architecture recovery. The improved version of the algorithm includes features that can extract more information about the system components and assigns different weights for each feature to increase the accuracy of the results. The technique was evaluated on two open source legacy systems and results revealed that the technique is able to provide results with high accuracy and cohesion. A more challenging study was conducted in [130], which performed a case study to investigate Linux operating system. Although there is no architectural documentation that describes Linux structure at a high level, the study examined the system to recover and build a conceptual architecture. The investigation was based on examining existing documentations, grouping source-code files into subsystems, defining relations between files, and applying clustering techniques to form a system architecture. The investigation disclosed that there is a high level of dependencies between Linux files at all levels of abstractions. Moreover, [131] applies evolutionary coupling techniques for software architecture recovery. The study extracted inter-modules dependencies between files in three open source projects based on various levels of evolutionary coupling. Using MoJoFM metric to evaluate the accuracy of the results, they concluded that involving evolutionary coupling in the architecture recovery acn increase the results accuracy by up to 40%.

6.3 A Glimpse of Architecture in Game Engine

In this section, we provide background information on architecture, architecture recovery, and game engine architecture.

6.3.1 Software Architecture Recovery

Software architecture is the foundation of a project. Its significance is based on its ability to predict software quality and reduce maintenance costs for evolution. Architecture in software systems consists of the structure of components in a program or system, their interrelationships, and the principles and guides that control the design and evolution in time [132]. One of the areas in software architecture is architecture recovery, which is the extraction of high-level software architecture information from source code entities such as files and classes [133].

Researchers and developers use architecture recovery techniques to retrieve lost architectural knowledge that guided the development of a software system in the past [134]. They can then use this knowledge to plan the system's evolution and ease its program maintainability, understanding and knowledge transfer [135, 136].

In the process of architecture recovery, we group implementation-level entities (e.g., files, classes, or functions) into clusters, where each cluster represents a component [127]. Some authors refer to these components as subsystems [130], which is the naming we chose for this work.

Hierarchical, density-based and distribution-based clustering are popular approaches [133] applied in automatic or semiautomatic fashion. Entities can be clustered based on similarity regarding different attributes, such as number of references to variables, types or other entities [137], textual content [138], naming conventions or number of dependencies [139].

While architecture recovery is usually mentioned in the context of understanding legacy enterprise systems, researchers have also applied it to a range of popular open-source codebases such as the Linux kernel [130], the Chromium browser [140], the Bash Unix Shell and the CVS version control system [137]. In this work, we do architecture recovery on open-source game engines.

6.3.2 Game Engine Architecture

The game engine architecture is the implementation and organisational structure of subsystems. Game engine architectures differ from other software systems architectures because their subsystems are structured as a software stack of multi-layers of abstractions increasing layer by layer until the game mechanics are described [125]. The importance of such an architecture lies behind the necessity to manage the constantly changing requirements of games, recurrent releases, complexity of game engines and their libraries and APIs [141].

The architecture varies from engine to engine, and there is currently no widely used stan-

dardised game engine architecture. However, we show in Figure 6.1 a high-level architecture created by Gregory [2]. It lays out the runtime subsystems that make up a typical game engine into multiple layers of abstraction. The bottom layer interacts directly with the hardware and operating system. Starting in the resources subsystem and going up, all generic and re-usable game logic is represented, such as physics simulation, graphical rendering and audio playback. Finally, on top, the game-specific layer represents a specific game logic with limited reusability beyond the scope of the game being developed, such as in-game camera systems, artificial intelligence, weapon systems, *etc.*

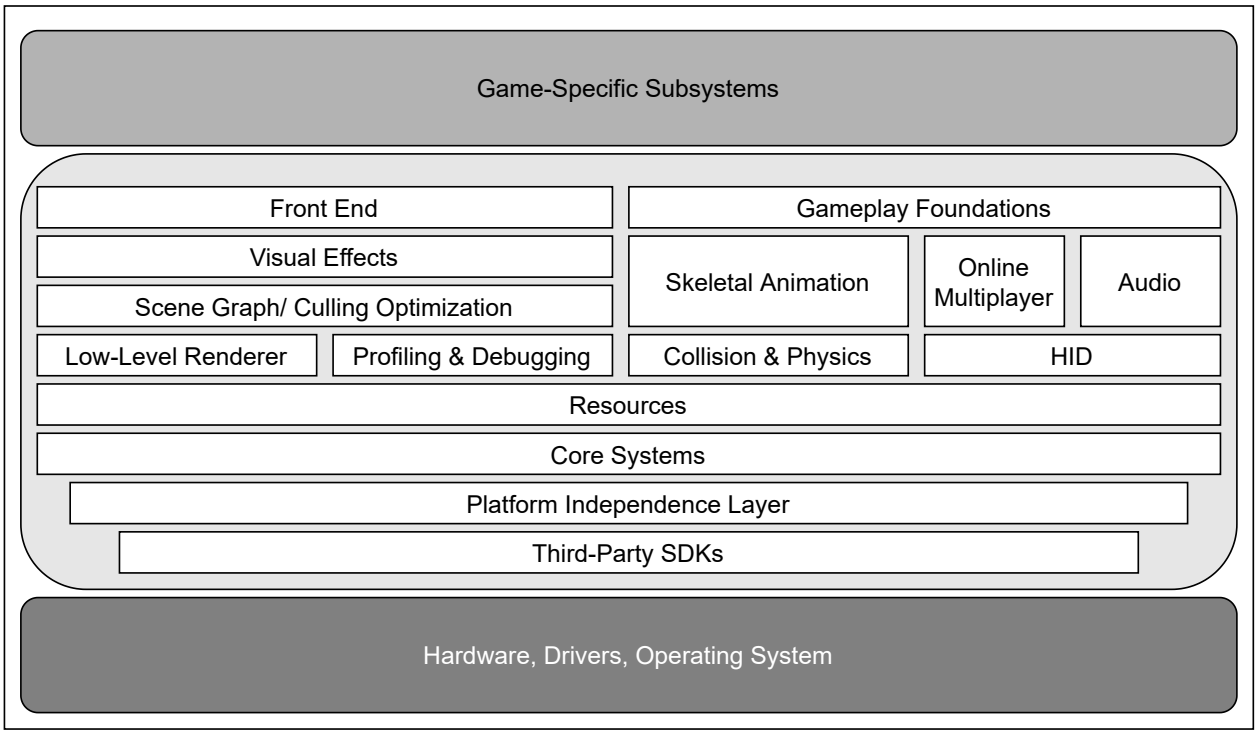


Figure 6.1 Summary of a High-Level Game Engine Architecture Adapted from Gregory [2].

There are few studies on game engine architecture. While there are books on this subject [2, 142, 143], often these publications tend to only briefly describe the high-level architecture before plunging straight down to the lowest level and describing how the individual components of the engine are implemented [144]. Furthermore, while such literature is an excellent source of information for writing an engine, it is of little help when the requirements differ from the solution described. Similarly, according to [124]: The current literature deals with the engine components, such as the behaviour specification, the scene render or the networking. Nevertheless, the game engine architecture connecting all these is a subject that has been barely covered. Seeking to cover this gap, we aim to provide more insight into the importance of high-level game engine subsystems through our study.

6.4 COnsensus Software Architecture

Figure 6.2 shows an overview of the approach, COSA (COnsensus Software Architecture). We break down the approach into several steps: game engine selection, subsystem identification and detection (architecture recovery), subsystem coupling calculation and ranking, and finally consensus algorithm application to obtain a model of a consensus of architecture subsystems ranked based on their degree of coupling.

The outcome of COSA should help us identify game engines' architectural commonalities and differences. Thus, it demonstrates which subsystems are present in all game engine architectures, which are only present in some architectures, how tightly the subsystems are coupled, and provides a consensus architecture of fundamental subsystems. We hence can help game engine developers in designing their engine architecture and focusing their efforts on developing loosely coupled subsystems.

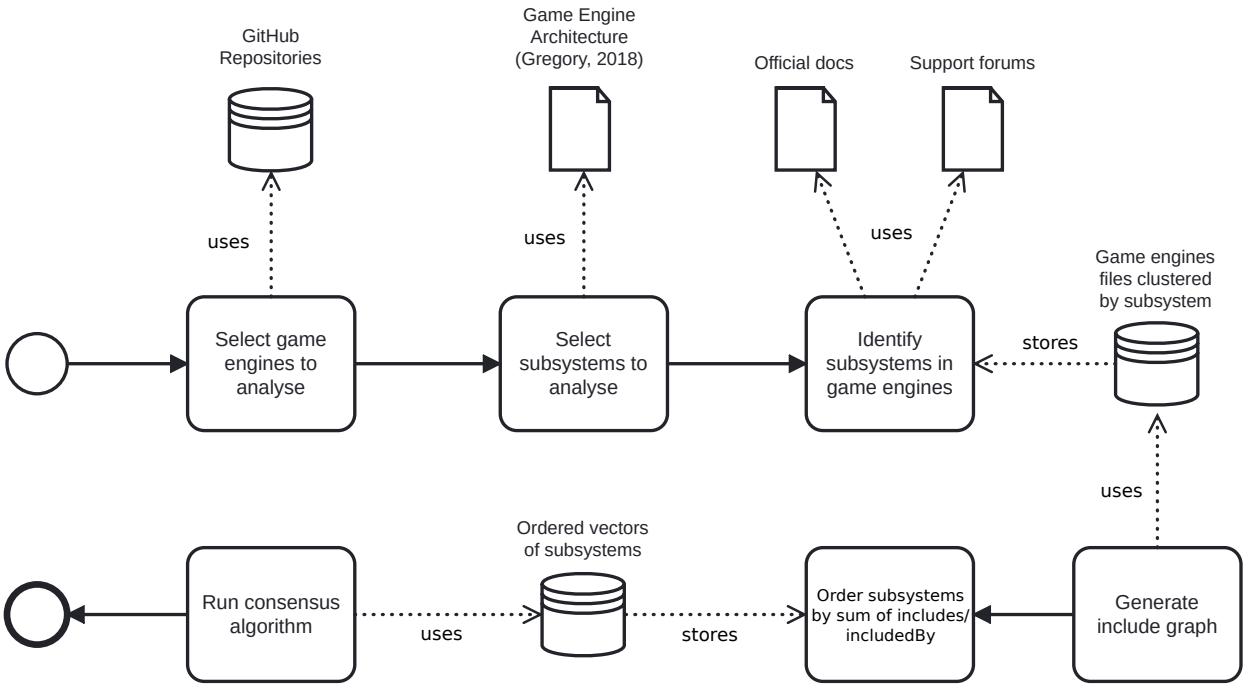


Figure 6.2 A Summary of the Steps Involved in COSA

6.4.1 System Selection

In the first step of our approach, we search and select systems that can be studied. To be able to find systems that serve the purpose of the study and help apply the approach successfully, we establish a set of adequate selection criteria. We define the following criteria for selecting game engines: • open source engines, • engines written in C++, • general-purpose engines, • engine repositories with the highest sum of forks, and stars and • unarchived repositories.

We decided to limit our search to engines with C++ as their primary programming language since C++ is the most widely used language for game engine development due to its performance and ability to reach low-level hardware [122]. In addition, we focused on general-purpose game engines as they target a broad range of game genres and therefore provide an overview of the features needed to make any game.

Considering that most open-source game engine repositories are stored and shared on GitHub, we chose it as our repository database. We used GitHub's search function to search and filter game engines that meet our predefined selection criteria. From the result of our search query, we selected the top 10 engines with the highest sum of forks and stars. Names of the selected engines are listed in Table 6.1.

Engine Name	**Forks + Stars**	**First Commit Year**
UnrealEngine	64100	2014
godot	59200	2013
cocos2d-x	23300	2010
o3de	6400	2021
Urho3D	4956	2011
gameplay	4900	2011
panda3d	4100	2000
olcPixelGameEngine	3963	2018
Piccolo	3892	2022
FlaxEngine	3613	2020

Table 6.1 Selected Game Engines along with the Sum of their GitHub Repositories Forks and Stars.

6.4.2 Subsystem Identification and Detection

We begin this step of the approach by defining ground truth data, which consists of a collection of basic subsystems that exist in any engine architecture. The ground truth data will be used later to validate the result of applying the consensus algorithm (Section 6.4.4). Then, we perform system architecture recovery by determining what subsystems compose the architecture of the selected system. We achieve that by analysing each system repository's

directories and files manually and clustering them according to the subsystems identified in the ground truth data.

In particular, we use the "Runtime Engine Architecture" proposed by Gregory [2] for defining the ground truth data. We chose Gregory's book since it is well-known among industry professionals and it aims to provide an in-depth discussion of the major subsystems that make up a standard game engine [2]. Besides reviewing game development and game engine foundational concepts, the author drills down into each game engine subsystem and discusses implementation details, performance issues and how the structure of these subsystems in the code impact the player and developer experiences.

Gregory [2] structures the engine architecture into 15 layered subsystems. While he divides each subsystem into a set of tools and smaller components, we choose to consider only the subsystems in the ground truth data. Although Gregory does not include the world editor (EDI) in the architecture, he emphasises the importance of including EDI when building a game engine, thus we add EDI to the ground truth data. Table 6.2 lists the 16 defined subsystems.

Abbrev.	**Name**
AUD	Audio
COR	Core Systems
DEB	Profiling and Debugging
EDI	World Editor
FES	Front End
GMP	Gameplay Foundations
HID	Human Interface Devices
LLR	Low-Level Renderer
OMP	Online Multiplayer
PHY	Collision and Physics
PLA	Platform Independence Layer
RES	Resources (Game Assets)
SDK	Third-party SDKs
SGC	Scene graph/culling optimizations
SKA	Skeletal Animation
VFX	Visual Effects

Table 6.2 Ground Truth Data of Subsystems

During the analysis process of directories and files contained in each engine repository for subsystem detection, we eliminated files that are not written in C++ or do not contain functionalities related to any subsystems. On the other hand, files containing subsystems-related functionalities are clustered to their corresponding subsystems. To determine whether a file contains functionalities related to a specific subsystem, we examined the followings:

- Directories, files, classes and methods naming. For instance, if a directory is named Audio, this indicates that the contents included therein are a part of the AUD subsystem.
- Source code comments that describe the semantics of a file, class or method.
- Official engine's documentation, wiki and/or support forums. Documentation can provide information on how an engine is structured, what subsystems are included, description of directories, *etc.*

When a file contains functionalities that are related to more than one subsystem, it is clustered into just one of those subsystems. We choose the most corresponding subsystem based on the engine documentation and authors' professional experiences.

We also identified several files that belong to subsystems that do not exist in the ground truth data. In these cases, we clustered the files with the most corresponding subsystems, even when a direct relationship could not always be found. If none of the subsystems specified in the ground truth data corresponds to the functions provided in the file, we exclude the file.

Lastly, we clustered together any 3rd-party libraries under the " 3rd-party SDKs" subsystem. Even while these libraries might contain some functionalities that serve a less generic subsystems, they are libraries that are not developed and maintained by the game engine developers. For example, the Piccolo engine contains no audio subsystem, but it includes *stb* libraries, which contain audio decoding and synthesising functionalities. These functionalities, however, are not used by Piccolo, even though they can be found on Piccolo's codebase. Therefore, the *stb* file in Piccolo was clustered into the SDK subsystem, and not the AUD subsystem. The clustering of engines's files into subsystems is available in the project repository [145].

6.4.3 Subsystem Coupling Degree Measurement

There are several metrics that can be used to measure the quality of a software system, and hence taken into consideration when developing the software, such as cohesion, inheritance, lines of codes, complexity, *etc.* However, in this work, besides finding the most fundamental subsystems to include in any system architecture, we opt for finding the degree of coupling of each subsystem. The underlying reason for prioritising coupling over other metrics is that highly coupled software systems are difficult to maintain, understand, test, or even reuse. Making a change to such a system requires more effort and time due to the increased dependency between its classes, especially when new releases are frequently expected, as they are in the video game industry.

Coupling was first introduced by Stevens *et al.* [146] and defined it as "the measure of the strength of association established by a connection from one module to another". In object-oriented programming, coupling is described as the dependency of one class on other classes. There are different measures that measure the degree of coupling between classes. Among these measures, *coupling Between Objects* (CBO) is the only measure that is a class-level measure, considers both import and export coupling, and determines the strength of coupling by the frequency of connections between classes [147]. CBO is a count of the number of classes that are coupled to a particular class [148]. Class A is coupled to class B if it references B and–or is being referenced by B.

To measure the degree of coupling of each subsystem in the recovered system architectures, we generate an include graph that shows the include relationships between files within each subsystem. We then calculate the degree of coupling by summing the number of include statements in each file and the number of times each file is included in other files (includedBy). Finally, we rank each architecture subsystem for each system based on their degree of coupling.

From all extracted subsystems, we generated an include graph for each subsystem in all engine architectures by using a script created by Francis Irving[4], which generates a *Graphviz* graph from the set of clustered files from the previous step. We then created a script whose pseudo-code is depicted in Listing 6.1. The idea is, for each engine, to select files related to each subsystem, compute coupling between objects (CBO) for the subsystem, add the subsystems to a hashmap with the name of the subsystem as the key and CBO as the value, and order the hashmap by value. The result is a set of game engine architecture of subsystems ordered by their degree of coupling, presented in Figure 6.3

```
# each engine will do a call to this
def get_vector(engine_files, subsystems):
    hashmap = {}
    for subsystem in subsystems:
        files_filtered = filter_files(engine_files, subsystem)
        calculated_metric = calculate_metric(files_filtered)
        hashmap[subsystem] = calculated_metric
    return sort_hashmap_by_value(hashmap, "descending")
```

Listing 6.1 Pseudo-code of Computing Coupling Between Objects (CBO)

[4] www.flourish.org/cinclude2dot/

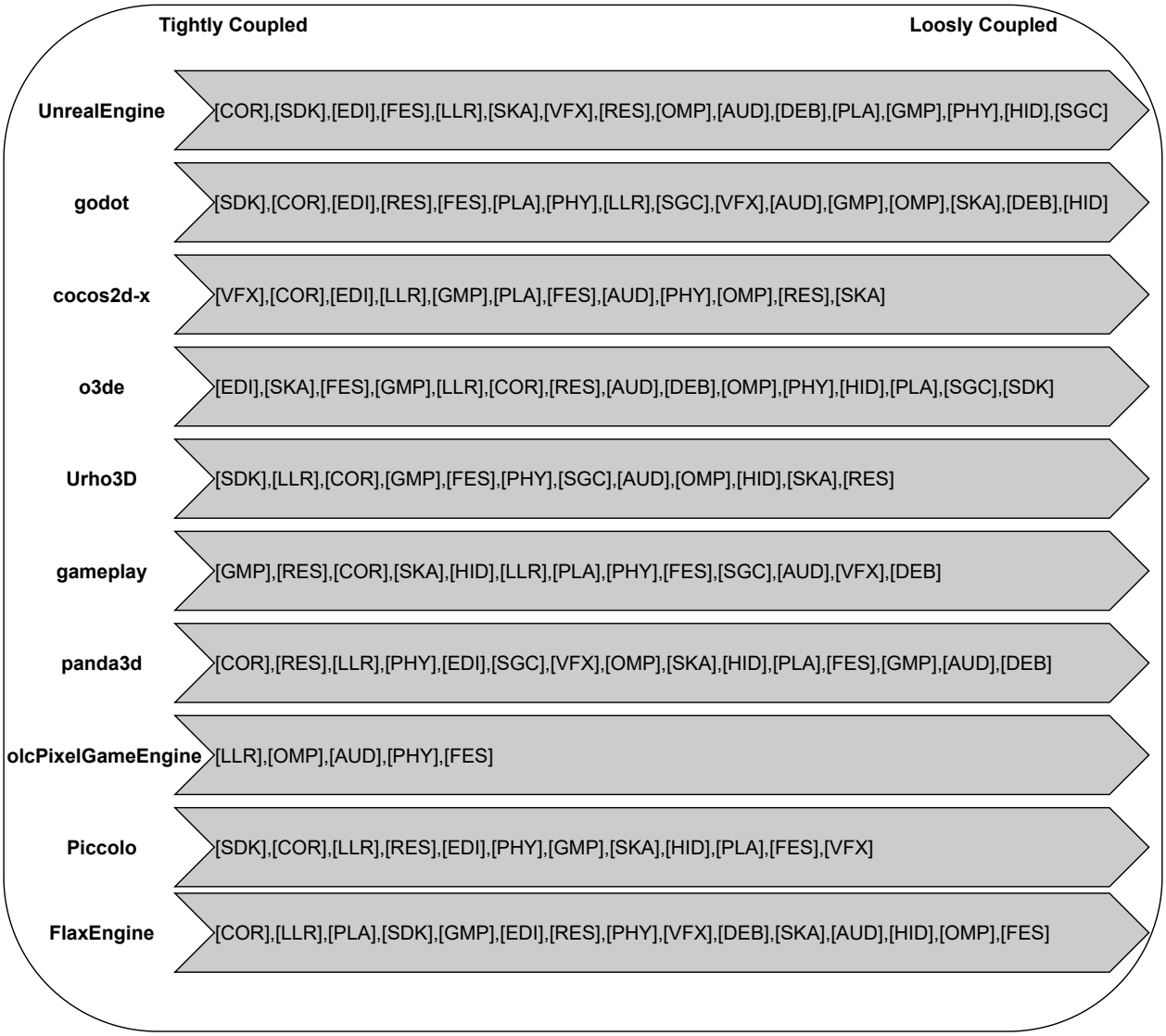


Figure 6.3 Game Engine Architectures of Subsystems Ordered by Coupling Degree

6.4.4 Consensus Model Generation

The final step of the approach is applying the selected consensus algorithms. We choose to apply the BioConcert and Kwiksort algorithms. The algorithms take as input the output from the previous step, which is a set of architectures of subsystems ranked by their degree of coupling. The algorithms should output one consensus architecture of subsystems.

We observe that not every engine includes all 16 identified subsystems, which in turn resulted in generating incomplete rankings; not all the elements exist in every ranking. To deal with incomplete rankings when applying consensus algorithms, we apply the unification technique to complete the rankings, which adds a bucket to the end of the rankings with missing

elements. For example, completing the ranking of Piccolo engine, adds a bucket with the missing subsystems as follows:

$$\begin{aligned} Piccolo \;=\; & [[SDK],[COR],[LLR],[RES],[EDI], \\ & [PHY],[GMP],[SKA],[HID],[PLA], \\ & [FES],[VFX],[SGC,AUD,OMP,DEB]] \end{aligned}$$

After completing the rankings, we apply the BioConcert and Kwiksort algorithms to the set of complete rankings. While results from both algorithms were very similar, the Kwiksort algorithm generated a result with a smaller generalised Kemeny score. Given that an optimal consensus ranking is the one with the smallest possible generalised Kemeny score, we adopt the result of the Kwiksort algorithm as our COSA for game engines.

6.5 Results & Discussions

In this Section, we present and discuss the result of applying the consensus algorithm (Kwik-Sort) to a set of subsystems of 10 game engine architectures that are ranked according to their coupling degree (tight to loose). The discussion revolves around two axes; first, we discuss commonalities between architectures and the most essential subsystems; second, we explore the most coupled subsystems and the underlying cause for their tight coupling.

6.5.1 Commonalities and Consensus Architecture of Subsystems

When comparing how similar game engine architectures are in terms of subsystems (refer to Figure 6.3), evidently most engine architectures, excluding olcPixelGameEngine, are composed of nearly the same subsystems. All subsystems exist in two engines, fifteen appeared in three engines, while twelve of the subsystems are part of the remaining four engine architectures. This confirms the success of our architecture recovery method and that we were able to identify 80% of the subsystems in all architectures.

According to Gregory, while details of architectures and implementation differ from engine to engine, all game engines must eventually include a set of main subsystems, such as rendering engine, physics engine, audio system, *etc.* [2]. Thus, this is an evident explanation for the commonality in engine architectures.

OlcPixelGameEngine, on the other hand, contains only five subsystems. The absence of more major subsystems occurs because olcPixelGameEngine is not a complete engine since it was developed by the YouTube channel OneLoneCoder for the purpose of teaching game engine

programming. From the top 10 selected open-source game engines, this is the only project that will not be developed further into a complete engine.

Figure 6.4 presents the result of applying the Kwiksort algorithm; a fundamental set of game engine subsystems ordered in a consensus fashion in accordance with their degree of coupling. Comparing the result of applying the consensus algorithm to the ground truth data, the consensus result identified all 16 subsystems as essential subsystems, and developers should therefore consider them in the architecture design when developing a modern game engine. In light of the fact that the majority of the subsystems were shared by the majority of the engine architectures, this not only confirms the validity of the consensus architecture of subsystems, but also ensures the importance of each subsystem as it plays a distinct role in the architecture. Hence, the proposed approach, COSA, fulfils our objectives of determining architectural commonalities between game engines and providing engine developers with a consensus architecture of a set of subsystems.

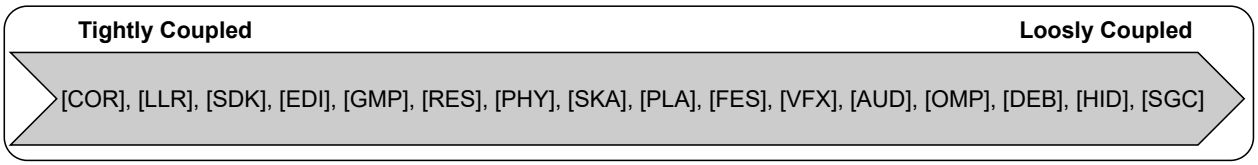


Figure 6.4 The Consensus Result of Applying the Kwiksort Algorithm

As mentioned in Subsection 6.4.2, we detected files that belong to unidentified subsystems in the ground truth data. A list of the discovered subsystems, their locations (game engines), and a relevant subsystem from the ground truth data are described in Table 6.3. We conclude that Gregory's engine architecture is not inclusive, and it can be expanded to comprise more subsystems provided by modern game engines. In spite of the discovery of new subsystems, these subsystems can not be regarded as essential since they are found in relatively few engines.

6.5.2 Degree of Coupling

The result of applying the consensus algorithm presents a consensus of the most coupled subsystems across all game engine architectures (Figure 6.4). The four most coupled subsystems are core systems, low-level renderer, 3rd-party SDKs, and world editor. We now discuss the underlying reasons behind the tight coupling and draw examples from the engines' files.

Core Systems and 3rd-Party SDKs As described by Gregory, these subsystems are responsible for low-level operations such as memory allocation, file I/O, system calls, as

Discovered Subsystems	**Game Engines**	**Relevant Subsystem from Ground Truth**
Code editor, Multi-user synchronization, Project creation and "cooking", CLI	UnrealEngine, o3de, panda3d	EDI
Cache, source control	UnrealEngine	RES
Cvars, graphs (data structure), Video subtitling and timecoding, Analytics, Media streaming	FlaxEngine, godot, o3de, panda3d, UnrealEngine	COR
Code hot reloading, visual scripting, assembler/compiler	FlaxEngine, godot, UnrealEngine	GMP
Virtual production (video post-production)	UnrealEngine	VFX
Screenshot capture	FlaxEngine	LLR
Foliage simulation	FlaxEngine, UnrealEngine	PHY
VR, AR, XR	godot, UnrealEngine	
Advertisement	UnrealEngine	
Cryptography	UnrealEngine, FlaxEngine	
Database	UnrealEngine, Urho3d, o3de	
Virtualization	UnrealEngine	
Cloud services integration	o3de	

Table 6.3 Newly Discovered Subsystems

well as communication with graphic and audio APIs. Therefore, they serve as support for all other high-level subsystems such as audio, low-level renderer and visual effects. This, in turn, means that files belonging to this subsystem are included by many other files. In Unreal Engine, for example, the most coupled core system file is *CoreMinimal.h*, and it is included by 14051 files while including only 151 files.

Low-Level Renderer It came as no surprise to us that the renderer subsystem is identified as one of the most coupled. It is responsible for producing 2D or 3D animated graphics we see on screen in all games. From the game objects to the UI of the world editor to everything needs to be drawn and continuously updated.

World Editor We observed that the editor has a high degree of coupling because it provides a visual interface to many other subsystems. Therefore, its files include many files from other subsystems, and they are included by these subsystems as well. Observing the files names in the Godot editor subsystem, we certainly notice that this subsystem serves many other subsystems within the engine. Examples of these files are: *animation_tree_editor_plugin.cpp*, *audio_stream_editor_plugin.cpp*, *particles_2d_editor_plugin.h*, *visual_script.cpp*.

In terms of most coupled game engines, we observe that UnrealEngine, panda3d and Urho3d are the three most coupled game engines. We noticed a correlation between the game engine coupling and metrics such as the number of files and forks+stars on GitHub. While this does not imply causation, this may be evidence that as engines and teams working on them grow in size, so does coupling.

6.6 Threats To Validity

Construct Validity

Subsystem Identification: Our subsystem identification in the ground truth data was based on Gregory's definition of "Runtime Engine Architecture". The list of 16 subsystems is not exhaustive. However, other than Gregory's book, there is no academic or technical research on game engine architecture. In the future, we intend to thoroughly examine, in collaboration with game engine developers, the source code of game engines to verify the accuracy of this list and possibly include more essential subsystems.

Clustering Bias: One of the authors manually clustered the game engine repository files and directories into subsystems. Despite the fact that the author's judgement was based on documentation attached to the repository, the author's judgement could be biassed.

Multi-Subsystems: During the file clustering process, we encountered files that could belong to more than one subsystem. The author relied on professional experience and engine documentation to cluster these files into the most corresponding subsystem. There is also a risk that the author's judgement could be biassed.

Unidentified Subsystems: During the file clustering process, we discovered a few files that might belong to subsystems that are not part of Gregory's engine architecture. We either discarded these files or clustered them into subsystems with comparable functionalities. As discussed in "*Subsystem Identification*", we intend in the future to extensively study game engines to detect undiscovered or newly created subsystems.

External Validity

Generalisation: We are confident that our approach can be generalised and applied to any other systems, as well as a wider range of game engines. In this work, we investigated 10 open-source game engines. Our selection of game engines might not represent all segments of the market. Popular engines like Unity and Source were left out of our investigation because they are not open-source and therefore analysing their source code is impossible. We reduced this threat by selecting general-purpose engines that serve all genres of games. Additionally, we recognise that most game engine development is closed-source. Therefore, the results may

not apply to all game engines, but ought to be valid for open-source game engines.

Reliability: To increase the reliability of our findings, we made all collected data and scripts available online in a public repository [145]. This allows other researchers to replicate and enhance our findings.

6.7 Conclusion

Video games are a prevalent form of entertainment and must provide gamers with new experiences that they have never had before. As a result, game development became more technically complex. Game engines are a key tool for building high-quality games. However, when building a game engine, developers struggle with the lack of knowledge about engine architecture in general, which engine subsystems to incorporate into the architecture, and managing architecture complexity to meet quality standards.

This chapter provided an overview of the commonalities and differences between game engine architectures in terms of their subsystems. Additionally, it defined a consensus of architecture and the degree of coupling among its subsystems to serve as a foundation for fair comparisons and discussions among game engine developers. Accordingly, the main objective of the approach was to provide developers with a high-level comparison between game engine architectures, help developers decide what subsystems to include in the architecture when building an engine, and present them with the most coupled subsystem so they can focus their development work on these subsystems to improve maintainability and reusability.

We presented an approach, COnsensus Software Architecture (COSA), that provides a ranking of game engine architecture subsystems based on their degree of coupling. We described COSA and applied it to 10 game engines. To identify what subsystems compose an engine, we performed an architecture recovery on the selected open-source game engines by manually analysing their repositories and clustering files into predefined subsystems. We built an included graph from the extracted subsystems to calculate their coupling degree and ranked each engine subsystem by its degree of coupling. To generate the ranking, we applied the consensus algorithm (KwikSort) to the ranked lists of architectures. We finally investigated the recovered architectures and compared their commonalities.

Our results showed that game engine architectures share many common subsystems. In fact, nearly all investigated game engines include the same subsystems in their architectures (COR, RES, FES, PHY, LLR, AUD, GMP, SKA). Additionally, our ranking concluded that all 16 predefined subsystems are essential and should be taken into consideration when building an advanced game engine. Besides, we showed that the most coupled subsystems

are core systems, 3rd-party libraries, world editors, and low-level renderers. As a result, we demonstrated that the consensus-based recommendation technique is capable of processing architecture data and generating recommendations for game engine architecture design issues.

CHAPTER 7 CONCLUSION AND FUTURE WORK

The development of any kind of software involves a variety of tasks that generate large amounts of data. Developers must constantly analyse and extract relevant information for solving software issues, making decisions, or performing their tasks. However, the growing volume of data makes it difficult to navigate and extract relevant information from software data. The reason for this is that software data is heterogeneous, evolves rapidly, and is constantly modified. Consequently, recommendation systems for software engineering (RSSEs) are needed to ease data searching and exploration as well as deliver valuable information to help developers perform specific tasks and increase productivity.

Recommendation systems for software engineering attracted considerable research attention to address a wide range of application-specific tasks. These recommendation systems rely on many techniques, from machine learning to data mining. However, these techniques can be subject to certain limitations. For example, requiring large datasets as input data or requiring developers' interaction with the application before making recommendations. Besides, available RSs have never been investigated and evaluated for their generalisability in producing recommendations from various types of data for various software applications.

In this thesis, we claim that the consensus algorithm can make recommendations that support resolving different software development tasks without the need for large input datasets. We proposed a consensus-based recommendation technique that generates recommendations by applying a consensus algorithm. The proposed technique recommends items in the form of a consensus ranking.

Hence, we restate the thesis statement as follows:

We proposed a software engineering recommendation technique based on the consensus algorithm that we applied and validated on various data types and resolved various software engineering-related issues in a variety of applications.

7.1 Contributions

We evaluated the applicability and usefulness of the consensus algorithms in supporting software developers by employing the consensus-based recommendation technique to address software engineering issues in three different applications.

Recommending Mobile App Reviews: We developed Review Prioritiser (RP) to provide developers with a consensus ranking of prioritised clusters of mobile app user reviews. We evaluated the usefulness and meaningfulness of the generated consensus rankings on four Android apps. We compared the rankings against reviews ranked by app developers manually. The comparison showed that there is a strong correlation between the two (average Kendall rank correlation coefficient = 0.516). We also invited app developers to evaluate the results qualitatively, and we surveyed their opinion on the technique. The survey responses demonstrated a high level of interest in the consensus-based prioritisation and recommendation for mobile app reviews.

Recommending Interaction Traces for Software Navigation: We proposed Consensus Task Interaction Trace Recommender (CITR) that recommends relevant file(s)-to-edit to assist developers carrying out the same or similar tasks on multiple software instances. We demonstrated the accuracy of the recommendations with average precision of 72%, recall of 61%, and F-measure of 60%. We conducted an observational comparative experiment with 30 developers to validate how well the proposed technique can affect developers' navigation behaviour. A qualitative analysis of the experiment revealed that CITR recommendations can guide new developers to exhibit a structured navigation behaviour that can increase their productivity. Finally, we compared CITR against MI and found that CITR yields higher accuracy and relevance recommendations than MI with average F-measure value of 60% and 20% respectively.

Recommending Game Engine Architecture Subsystems: We built COnsensus Software Architecture (COSA), that recommends a ranking of consensus fundamental game engine architecture subsystems that can support developer when designing an engine architecture. The ranking was determined by calculating the coupling between objects (CBO) metric for each subsystem. We evaluated the model by comparing it with predefined ground truth data. The results of the evaluation indicate that all identified subsystems from the architecture recovery are fundamental and should be considered when designing an engine architecture. Moreover, the ranking revealed that the most coupled subsystems are core systems, low-level rendering, third-party SDKs, and world editor.

7.2 Discussion

We presented in Section 3.1.4 a number of consensus algorithms. Brancotte *et al.* [52] investigated and compared the performance of these algorithms, and concluded that ExactAlgo-

rithm, BioConsert, and KwikSort are the most outperforming consensus algorithms. Therefore, we applied these algorithms in our research studies. Factors such as size of datasets, types of data, numbers of items in each ranking, and natures of ties, all affect the generalised Kemeny score's outcome and are likely to generate different results. To produce the highest quality recommendations possible, we ran each algorithm on the dataset in each study, compared the results, and chose the one with the lowest generalised Kemeny score. When we compared the recommendation results generated by each algorithm in all of our research studies, we observed that the results were very similar, with slightly different Kemeny scores. Hence, in each study, we decided to use the algorithm that generates results with the lowest generalised Kemeny score; ExactAlgorithm in Chapter 4, BioConsert in Chapter 5, and KwikSort in Chapter 6.

One of the limitations of some of the recommendation techniques, such as content-based techniques, is their inability to make recommendations without sufficient knowledge about the features of each item included in the dataset. In contrast to these techniques, the consensus algorithm does not require any metadata that describes the input items to generate recommendations. Furthermore, unlike machine learning techniques that rely on large datasets to train models and generate accurate recommendations, the consensus algorithm can scale with the amount of available data. We showed throughout our studies that the three approaches are able to provide high-quality recommendations without requiring massive input data. For example, in the study of AR-Miner [9], the authors relayed on exporting user reviews from popular mobile applications to ensure the availability of a large dataset. They exported close to $200K$ user reviews from four applications to partition the data into training and test sets. In our approach of prioritising mobile app user reviews (Chapter 4), we demonstrated the success of the consensus algorithm at producing an accurate (strong Kendall rank correlation coefficient) prioritised set of user reviews with database size of nearly 700 reviews.

Some recommendation approaches are based on interactive input techniques, which rely on input data from the users to create association between the input and existing data before making recommendations. We showed through our studies that all the proposed consensus-based approaches do not require any input form the user to generate recommendations and instead rely solely on input dataset. For example, our approach Consensus Interaction Trace Recommender (CITR) (Chapter 5) does not wait for developers to start interacting with the source-code before recommending relevant elements that are similar to those with which the developers have interacted. Throughout our research, we proposed approaches that are based on the consensus-based recommendation technique. We applied and tested the approaches on three different types of data to help software developers with completing various software engineering tasks on different software applications. Consequently, our research studies vali-

date and prove the generalisability of the proposed technique and its capability at providing recommendations that are close to the developers' needs for many applications.

7.3 Future Research

In Section 3.2, we presented the fundamental steps of building a software engineering recommendation system based on observations made from our research as well as a review of existing RSSEs studies. There is a lack of a research study that provides comprehensive knowledge about each step. To fill this gap, we plan to conduct an extensive literature review to obtain an in-depth explanation of how to detect and frame a software problem from various perspectives; a description of all types of data, their sources and methods of data extraction; an overview of all data cleaning and preprocessing techniques, specific to each type of data; a definition of all recommendation approaches, their techniques, and the factors that make a specific technique suitable for a particular situation; lastly, a customised evaluation design that can suite any recommendation technique. Our goal is to provide an in-depth view of recommendation systems.

In Chapter 4, we used an existing tool, CLAP [70], to categorise and cluster the mobile app reviews. Following a manual analysis of the tool categorisation and clustering results, we observed that the tool mis-categorises and mis-clusters some of the reviews. To ensure that we recommend accurate prioritisation from applying the consensus algorithm, we plan to propose an approach that bases the categorization and clustering on deep learning techniques, which have shown great success in both supervised and unsupervised data clustering, but were not investigated for clustering user reviews. We intend to apply and compare the results of the most promising deep learning techniques, such as Long Short Term Memory (LSTM), Convolutional Neural Network (CNN), or Recurrent Neural Networks (RNN). We also want to design an all-in-one approach that can automatically clean and preprocess reviews, categorise, cluster, and recommend a prioritised ranking of user reviews.

We intend to also expand our research in other directions: boarden the research by carrying out studies on industry-tailored software applications and real industry data, involving larger numbers of developers, with varying experience levels, in experimenting and evaluating the approaches, investigating and comparing the quality of recommendations of the consensus algorithms on a variety of dataset sizes, ranging from medium to large, including more systems with different programming languages in the evaluation process, automating data preprocessing for better quality results, and finally, building a complete consensus-based recommendation system that can be available and used by developers to support their tasks in any type of software applications.

REFERENCES

[1] D. Goldberg *et al.*, "Using collaborative filtering to weave an information tapestry," *Communications of the ACM*, vol. 35, no. 12, pp. 61–70, 1992.

[2] J. Gregory, *Game engine architecture*, 3rd ed. Boca Raton: Taylor & Francis, CRC Press, 2018.

[3] M. Robillard, R. Walker, and T. Zimmermann, "Recommendation systems for software engineering," *IEEE software*, vol. 27, no. 4, pp. 80–86, 2009.

[4] U. Pakdeetrakulwong, P. Wongthongtham, and W. V. Siricharoen, "Recommendation systems for software engineering: A survey from software development life cycle phase perspective," in *The 9th International Conference for Internet Technology and Secured Transactions (ICITST-2014)*. IEEE, 2014, pp. 137–142.

[5] A. Felfernig *et al.*, "Basic approaches in recommendation systems," *Recommendation Systems in Software Engineering*, pp. 15–37, 2014.

[6] T. Zimmermann *et al.*, "Mining version histories to guide software changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, 2005.

[7] F. Palma *et al.*, "Recommendation system for design patterns in software development: An dpr overview," in *2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE)*. IEEE, 2012, pp. 1–5.

[8] M. Andric, W. Hall, and L. Carr, "Assisting artifact retrieval in software engineering projects," in *Proceedings of the 2004 ACM symposium on Document engineering*, 2004, pp. 48–50.

[9] N. Chen *et al.*, "Ar-miner: mining informative reviews for developers from mobile app marketplace," in *Proceedings of the 36th international conference on software engineering*, 2014, pp. 767–778.

[10] P. Resnick *et al.*, "Grouplens: An open architecture for collaborative filtering of netnews," in *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, 1994, pp. 175–186.

[11] W. Hill *et al.*, "Recommending and evaluating choices in a virtual community of use," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 1995, pp. 194–201.

[12] U. Shardanand and P. Maes, "Social information filtering: Algorithms for automating "word of mouth"," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 1995, pp. 210–217.

[13] M. Shao *et al.*, "Computational drug repurposing based on a recommendation system and drug–drug functional pathway similarity," *Molecules*, vol. 27, no. 4, p. 1404, 2022.

[14] J. G. Mohebzada, G. Ruhe, and A. Eberlein, "Systematic mapping of recommendation systems for requirements engineering," in *2012 International Conference on Software and System Process (ICSSP)*. IEEE, 2012, pp. 200–209.

[15] R. Holmes *et al.*, "Automatically recommending triage decisions for pragmatic reuse tasks," in *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 397–408.

[16] T. Kobayashi, N. Kato, and K. Agusa, "Interaction histories mining for software change guide," in *2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE)*. IEEE, 2012, pp. 73–77.

[17] M. Ichii *et al.*, "Software component recommendation using collaborative filtering," in *2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*. IEEE, 2009, pp. 17–20.

[18] F. O. Isinkaye, Y. O. Folajimi, and B. A. Ojokoh, "Recommendation systems: Principles, methods and evaluation," *Egyptian informatics journal*, vol. 16, no. 3, pp. 261–273, 2015.

[19] B. Ashok *et al.*, "Debugadvisor: A recommender system for debugging," in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2009, pp. 373–382.

[20] N. Lopez and A. Van Der Hoek, "The code orb: supporting contextualized coding via at-a-glance views (nier track)," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 824–827.

[21] A. Bacchelli, L. Ponzanelli, and M. Lanza, "Harnessing stack overflow for the ide," in *2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE)*. IEEE, 2012, pp. 26–30.

[22] J. Cordeiro, B. Antunes, and P. Gomes, "Context-based recommendation to support problem solving in software development," in *2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE)*. IEEE, 2012, pp. 85–89.

[23] R. D. Burke, K. J. Hammond, and B. C. Young, "Knowledge-based navigation of complex information spaces," in *Proceedings of the national conference on artificial intelligence*, vol. 462, 1996, p. 468.

[24] L. O. Colombo-Mendoza *et al.*, "Recommetz: A context-aware knowledge-based mobile recommender system for movie showtimes," *Expert Systems with Applications*, vol. 42, no. 3, pp. 1202–1222, 2015.

[25] F. Ricci, L. Rokach, and B. Shapira, "Introduction to recommender systems handbook," in *Recommender systems handbook*. Springer, 2010, pp. 1–35.

[26] D. Pagano and W. Maalej, "User feedback in the appstore: An empirical study," in *2013 21st IEEE international requirements engineering conference (RE)*. IEEE, 2013, pp. 125–134.

[27] L. V. G. Carreño and K. Winbladh, "Analysis of user comments: an approach for software requirements evolution," in *2013 35th international conference on software engineering (ICSE)*. IEEE, 2013, pp. 582–591.

[28] S. Proksch, V. Bauer, and G. C. Murphy, "How to build a recommendation system for software engineering," *Software Engineering: International Summer Schools, LASER 2013-2014, Elba, Italy, Revised Tutorial Lectures 10*, pp. 1–42, 2015.

[29] S. García, J. Luengo, and F. Herrera, *Data preprocessing in data mining*. Springer, 2015.

[30] M. Robillard *et al.*, *Recommendation Systems in Software Engineering*. Springer, 2014.

[31] L. Etaiwi *et al.*, "Order in chaos: prioritizing mobile app reviews using consensus algorithms," in *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2020, pp. 912–920.

[32] ——, "Consensus task interaction trace recommender to guide developers' software navigation," *Empirical Software Engineering*, under review 2023.

[33] ——, "Consensus in game engine architectures: an overview of subsystem coupling in game engines," *17th European Conference on Software Architecture (ECSA)*, To Be Submitted 2023.

[34] F. Ricci *et al.*, *Recommender systems handbook*. Springer, 2011.

[35] H.-J. Happel and W. Maalej, "Potentials and challenges of recommendation systems for software development," in *Proceedings of the 2008 international workshop on Recommendation systems for software engineering*, 2008, pp. 11–15.

[36] M. Santosh Kumar and K. Balakrishnan, "Development of a model recommender system for agriculture using apriori algorithm," in *Cognitive Informatics and Soft Computing: Proceeding of CISC 2017*. Springer, 2019, pp. 153–163.

[37] R. Burke, "The wasabi personal shopper: A case-based recommender system," in *AAAI/IAAI*, 1999, pp. 844–849.

[38] J. Lu, "A personalized e-learning material recommender system," in *International conference on information technology and applications*. Macquarie Scientific Publishing, 2004.

[39] D. Cubranic and G. Murphy, "Hipikat: recommending pertinent software development artifacts," in *25th International Conference on Software Engineering, 2003. Proceedings.*, 2003, pp. 408–418.

[40] H. P. Young and A. Levenglick, "A consistent extension of condorcet's election principle," *SIAM Journal on applied Mathematics*, vol. 35, no. 2, pp. 285–300, 1978.

[41] J. Bartholdi, C. A. Tovey, and M. A. Trick, "Voting schemes for which it can be difficult to tell who won the election," *Social Choice and welfare*, vol. 6, pp. 157–165, 1989.

[42] J. Sese and S. Morishita, "Rank aggregation method for biological databases," *Genome Informatics*, vol. 12, pp. 506–507, 2001.

[43] S. Cohen-Boulakia, A. Denise, and S. Hamel, "Using medians to generate consensus rankings for biological data," in *Scientific and Statistical Database Management: 23rd International Conference, SSDBM 2011, Portland, OR, USA, July 20-22, 2011. Proceedings 23*. Springer, 2011, pp. 73–90.

[44] R. P. DeConde *et al.*, "Combining results of microarray experiments: a rank aggregation approach," *Statistical applications in genetics and molecular biology*, vol. 5, no. 1, 2006.

[45] B. Miller *et al.*, "The wisdom of crowds in rank ordering problems," in *9th International conference on cognitive modeling*. Citeseer, 2009.

[46] A. Ali and M. Meilă, "Experiments with kemeny ranking: What works when?" *Mathematical Social Sciences*, vol. 64, no. 1, pp. 28–40, 2012.

[47] A. Ruiz-Padillo *et al.*, "Social choice functions: A tool for ranking variables involved in action plans against road noise," *Journal of environmental management*, vol. 178, pp. 1–10, 2016.

[48] J. P. Baskin and S. Krishnamurthi, "Preference aggregation in group recommender systems for committee decision-making," in *Proceedings of the third ACM conference on Recommender systems*, 2009, pp. 337–340.

[49] C. Dwork *et al.*, "Rank aggregation methods for the web," in *Proceedings of the 10th international conference on World Wide Web*, 2001, pp. 613–622.

[50] X. Munoz, "Fair ranking in dancesport competitions," *Department of Computer Science, University of Otago*, vol. 39, 2003.

[51] R. Milosz, "Étude algorithmique et combinatoire de la méthode de kemeny-young et du consensus de classements," Ph.D. dissertation, Université de Montréal, Montréal, QC, Canada, 2019.

[52] B. Brancotte *et al.*, "Rank aggregation with ties: Experiments and analysis," *Proc. VLDB Endow.*, vol. 8, no. 11, p. 1202–1213, jul 2015.

[53] P. Andrieu, "Passage à l'échelle, propriétés et qualité des algorithmes de classements consensuels pour les données biologiques massives," Ph.D. dissertation, Université Paris-Saclay, Gif-sur-Yvette, France, 2021.

[54] N. Ailon, M. Charikar, and A. Newman, "Aggregating inconsistent information: Ranking and clustering," *J. ACM*, vol. 55, no. 5, nov 2008.

[55] J. C. de Borda, "Memoir on the elections in the poll, 1781," *History of the Royal Academy of Sciences, Paris*, 1953.

[56] R. Fagin, R. Kumar, and D. Sivakumar, "Efficient similarity search and classification via rank aggregation," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003, pp. 301–312.

[57] C. Fox, A. Levitin, and T. Redman, "The notion of data and its quality dimensions," *Information processing & management*, vol. 30, no. 1, pp. 9–19, 1994.

[58] S. Ramírez-Gallego *et al.*, "A survey on data preprocessing for data stream mining: Current status and future directions," *Neurocomputing*, vol. 239, pp. 39–57, 2017.

[59] V. Agarwal, "Research on data preprocessing and categorization technique for smartphone review analysis," *International Journal of Computer Applications*, vol. 131, no. 4, pp. 30–36, 2015.

[60] I. Avazpour *et al.*, "Dimensions and Metrics for Evaluating Recommendation Systems," in *Recommendation Systems in Software Engineering*, M. P. Robillard *et al.*, Eds. Springer Berlin Heidelberg, 2014, pp. 245–273.

[61] A. Dogtiev, "App stores list 2019," https://www.businessofapps.com/guide/app-stores-list/, September 2019.

[62] A. Al-Subaihin *et al.*, "App store mining and analysis," in *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, 2015, pp. 1–2.

[63] "Wordpress app on google play store," https://play.google.com/store/apps/details?id=org.wordpress.android.

[64] S. Panichella *et al.*, "Ardoc: App reviews development oriented classifier," in *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, 2016, pp. 1023–1027.

[65] ——, "How can i improve my app? classifying user reviews for software maintenance and evolution," in *2015 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2015, pp. 281–290.

[66] C. Iacob and R. Harrison, "Retrieving and analyzing mobile apps feature requests from online reviews," in *2013 10th working conference on mining software repositories (MSR)*. IEEE, 2013, pp. 41–44.

[67] B. Fu *et al.*, "Why people hate your app: Making sense of user feedback in a mobile app store," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2013, pp. 1276–1284.

[68] F. Palomba *et al.*, "User reviews matter! tracking crowdsourced reviews to support evolution of successful apps," in *2015 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2015, pp. 291–300.

[69] M. V. Phong *et al.*, "Mining user opinions in mobile app reviews: A keyword-based approach (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 749–759.

[70] S. Scalabrino *et al.*, "Listening to the crowd for the release planning of mobile apps," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 68–86, 2017.

[71] A. Ciurumelea *et al.*, "Analyzing reviews and code of mobile apps for better release planning," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 91–102.

[72] J. Gebauer, Y. Tang, and C. Baimai, "User requirements of mobile technology: results from a content analysis of user reviews," *Information Systems and e-Business Management*, vol. 6, pp. 361–384, 2008.

[73] P. Laurent, J. Cleland-Huang, and C. Duan, "Towards automated requirements triage," in *15th IEEE International Requirements Engineering Conference (RE 2007)*. IEEE, 2007, pp. 131–140.

[74] P. Avesani *et al.*, "Facing scalability issues in requirements prioritization with machine learning techniques," in *13th IEEE International Conference on Requirements Engineering (RE'05)*. IEEE, 2005, pp. 297–305.

[75] R. Beg, Q. Abbas, and R. P. Verma, "An approach for requirement prioritization using b-tree," in *2008 First International Conference on Emerging Trends in Engineering and Technology*. IEEE, 2008, pp. 1216–1221.

[76] S. Keertipati, B. T. R. Savarimuthu, and S. A. Licorish, "Approaches for prioritizing feature improvements extracted from app reviews," in *Proceedings of the 20th international conference on evaluation and assessment in software engineering*, 2016, pp. 1–6.

[77] C. Gao *et al.*, "Paid: Prioritizing app issues for developers by tracking user reviews over versions," in *2015 IEEE 26th international symposium on software reliability engineering (ISSRE)*. IEEE, 2015, pp. 35–45.

[78] "Replication package for rp," https://doi.org/10.5281/zenodo.3748768.

[79] M. G. Kendall, "Rank correlation methods." *Griffin*, 1948.

[80] J. Cohen, *Statistical power analysis for the behavioral sciences*. Academic press, 2013.

[81] CRM, "17 crm statistics: Growth, revenue, adoption rates & more facts," https://crm.org/crmland/crm-statistics, 2022.

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

[83] I. Hammouda *et al.*, *Open Source Systems: Long-Term Sustainability*. Springer, 2012.

[84] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: A study of developer work habits," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 492–501.

[85] A. J. Ko *et al.*, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Trans. Softw. Eng.*, vol. 32, no. 12, p. 971–987, dec 2006.

[86] S. Lee *et al.*, "The impact of view histories on edit recommendations," *IEEE Transactions on Software Engineering*, vol. 41, no. 3, pp. 314–330, 2015.

[87] R. Ramsauer, D. Lohmann, and W. Mauerer, "Observing custom software modifications: A quantitative approach of tracking the evolution of patch stacks," in *Proceedings of the 12th International Symposium on Open Collaboration*, ser. OpenSym '16. New York, NY, USA: Association for Computing Machinery, 2016.

[88] A. Ying *et al.*, "Predicting source code changes by mining change history," *IEEE Transactions on Software Engineering*, vol. 30, no. 9, pp. 574–586, 2004.

[89] J. Singer, R. Elves, and M.-A. Storey, "Navtracks: supporting navigation in software maintenance," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, 2005, pp. 325–334.

[90] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '06/FSE-14. New York, NY, USA: Association for Computing Machinery, 2006, p. 1–11.

[91] Z. Soh *et al.*, "On the effect of program exploration on maintenance tasks," in *2013 20th Working Conference on Reverse Engineering (WCRE)*, 2013, pp. 391–400.

[92] A. Sahm and W. Maalej, "Switch! recommending artifacts needed next based on personal and shared context," in *Software Engineering 2010 - Workshopband (inkl. Doktorandensymposium),Fachtagung des GI-Fachbereichs Softwaretechnik, 22.-26.02.2010, Paderborn*, ser. LNI, G. Engels *et al.*, Eds., vol. P-160. GI, 2010, pp. 473–484.

[93] R. Robbes, D. Pollet, and M. Lanza, "Replaying ide interactions to evaluate and improve change prediction approaches," *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pp. 161–170, 2010.

[94] S. Lee and S. Kang, "Clustering and recommending collections of code relevant to tasks," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, 2011, pp. 536–539.

[95] R. DeLine, M. Czerwinski, and G. Robertson, "Easing program comprehension by sharing navigation data," in *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, 2005, pp. 241–248.

[96] A. T. Ying and M. P. Robillard, "The influence of the task on programmer behaviour," in *2011 IEEE 19th International Conference on Program Comprehension*, 2011, pp. 31–40.

[97] H. Sanchez, R. Robbes, and V. M. Gonzalez, "An empirical study of work fragmentation in software evolution tasks," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 251–260.

[98] L. Zou, M. W. Godfrey, and A. E. Hassan, "Detecting interaction coupling from task interaction histories," in *15th IEEE International Conference on Program Comprehension (ICPC '07)*, 2007, pp. 135–144.

[99] C. Parnin and S. Rugaber, "Resumption strategies for interrupted programming tasks," in *2009 IEEE 17th International Conference on Program Comprehension*, 2009, pp. 80–89.

[100] W. Teitelman and L. Masinter, "The interlisp programming environment," *Computer*, vol. 14, no. 4, pp. 25–33, 1981.

[101] D. Rothlisberger *et al.*, "Supporting task-oriented navigation in ides with configurable heatmaps," in *2009 IEEE 17th International Conference on Program Comprehension*, 2009, pp. 253–257.

[102] M. P. Robillard and B. Dagenais, "Recommending change clusters to support software investigation: an empirical study," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 22, no. 3, pp. 143–164, 2010.

[103] S. Lee and S. Kang, "Clustering navigation sequences to create contexts for guiding code navigation," *J. Syst. Softw.*, vol. 86, no. 8, p. 2154–2165, aug 2013.

[104] R. Robbes and M. Lanza, "Improving code completion with program history," *Automated Software Engineering*, vol. 17, no. 2, pp. 181–212, 2010.

[105] Z. Wan, G. C. Murphy, and X. Xia, "Predicting code context models for software development tasks," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 809–820.

[106] M. P. Robillard, "Topology analysis of software dependencies," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 17, no. 4, pp. 1–36, 2008.

[107] M. Robillard, W. Coelho, and G. Murphy, "How effective developers investigate source code: an exploratory study," *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 889–903, 2004.

[108] J. Sillito, G. C. Murphy, and K. De Volder, "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, 2006, p. 23–34.

[109] J. Wang *et al.*, "An exploratory study of feature location process: Distinct phases, recurring patterns, and elementary actions," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, 2011, pp. 213–222.

[110] J. Starke, C. Luce, and J. Sillito, "Searching and skimming: An exploratory study," in *2009 IEEE International Conference on Software Maintenance*, 2009, pp. 157–166.

[111] S. Chattopadhyay *et al.*, "Latent patterns in activities: A field study of how developers manage context," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 373–383.

[112] Z. Soh *et al.*, "Noise in mylyn interaction traces and its impact on developers and recommendation systems," *Empirical Software Engineering*, vol. 23, no. 2, pp. 645–692, 2018.

[113] T. Fritz *et al.*, "Developers' code context models for change tasks," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 7–18.

[114] S. Amann, S. Proksch, and S. Nadi, "Feedbag: An interaction tracker for visual studio," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, 2016, pp. 1–3.

[115] R. Minelli *et al.*, "Quantifying program comprehension with interaction data," in *2014 14th International Conference on Quality Software*, 2014, pp. 276–285.

[116] "Replication package for citr," https://www.ptidej.net/downloads/replications/emse22a/.

[117] Z. Soh *et al.*, "Towards understanding how developers spend their effort during maintenance activities," in *2013 20th Working Conference on Reverse Engineering (WCRE)*, 2013, pp. 152–161.

[118] I. Majid and M. P. Robillard, "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, 2005, pp. 70–74.

[119] L. Bao *et al.*, "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, 2017, pp. 170–181.

[120] J. Fleming, "Down the hyper-spatial tube: Spacewar and the birth of digital game culture." https://http://www.gamasutra.com, 2007.

[121] Newzoo, "Global games market report," https://newzoo.com/products/reports/global-games-market-report, 2022.

[122] C. Politowski *et al.*, "Are game engines software frameworks? a three-perspective study," *Journal of Systems and Software*, vol. 171, p. 110846, 2021.

[123] A. Rollings and D. Morris, *Game architecture and design*. Paraglyph Press, 1999.

[124] C. Marin, M. Chover, and J. M. Sotoca, "Prototyping a game engine architecture as a multi-agent system," in *Computer Science Research Notes*. Západočeská univerzita,

2019. [Online]. Available: http://wscg.zcu.cz/wscg2019/2019-papers/!!_CSRN-2802-4.pdf

[125] D. Maggiorini *et al.*, "Smash: A distributed game engine architecture," in *2016 IEEE Symposium on Computers and Communication (ISCC)*. IEEE, 2016, pp. 196–201.

[126] E. F. Anderson *et al.*, "The case for research in game engine architecture," in *Proceedings of the 2008 Conference on Future Play: Research, Play, Share*, 2008, pp. 228–231.

[127] J. Garcia, I. Ivkovic, and N. Medvidovic, "A comparative analysis of software architecture recovery techniques," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Silicon Valley, CA, USA: IEEE, Nov. 2013, pp. 486–496. [Online]. Available: http://ieeexplore.ieee.org/document/6693106/

[128] H. A. Müller *et al.*, "A reverse-engineering approach to subsystem structure identification," *Journal of Software Maintenance: Research and Practice*, vol. 5, no. 4, pp. 181–204, 1993.

[129] Y. Wang *et al.*, "Improved hierarchical clustering algorithm for software architecture recovery," in *2010 international conference on intelligent computing and cognitive informatics*. IEEE, 2010, pp. 247–250.

[130] I. T. Bowman, R. C. Holt, and N. V. Brewster, "Linux as a case study: its extracted software architecture," in *Proceedings of the 21st international conference on Software engineering - ICSE '99*. Los Angeles, California, United States: ACM Press, 1999, pp. 555–563. [Online]. Available: http://portal.acm.org/citation.cfm?doid=302405.302691

[131] A. Saydemir, M. E. Simitcioglu, and H. Sozer, "On the use of evolutionary coupling for software architecture recovery," in *2021 15th Turkish National Software Engineering Symposium (UYMS)*. IEEE, 2021, pp. 1–6.

[132] M. Shaw and D. Garlan, *Software architecture: perspectives on an emerging discipline*. Upper Saddle River, N.J: Prentice Hall, 1996.

[133] T. Yang *et al.*, "Systematic review on next-generation web-based software architecture clustering models," *Computer Communications*, vol. 167, pp. 63–74, Feb. 2021. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0140366420320284

[134] J. C. Duenas, W. L. de Oliveira, and J. A. de la Puente, "Architecture recovery for software evolution," in *Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering*. IEEE, 1998, pp. 113–119.

[135] K. Wong *et al.*, "Structural redocumentation: A case study," *IEEE Software*, vol. 12, no. 1, pp. 46–54, 1995.

[136] H. A. Müller *et al.*, "A reverse-engineering approach to subsystem structure identification," *Journal of Software Maintenance: Research and Practice*, vol. 5, no. 4, pp. 181–204, 1993.

[137] Y. Wang *et al.*, "Improved Hierarchical Clustering Algorithm for Software Architecture Recovery," in *2010 International Conference on Intelligent Computing and Cognitive Informatics*. Kuala Lumpur, Malaysia: IEEE, Jun. 2010, pp. 247–250. [Online]. Available: http://ieeexplore.ieee.org/document/5565989/

[138] M. Risi, G. Scanniello, and G. Tortora, "Using fold-in and fold-out in the architecture recovery of software systems," *Formal Aspects of Computing*, vol. 24, no. 3, pp. 307–330, May 2012. [Online]. Available: https://dl.acm.org/doi/10.1007/s00165-011-0199-y

[139] S. Ducasse and D. Pollet, "Software architecture reconstruction: A process-oriented taxonomy," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 573–591, 2009.

[140] A. Saydemir, M. E. Simitcioglu, and H. Sozer, "On the Use of Evolutionary Coupling for Software Architecture Recovery," in *2021 15th Turkish National Software Engineering Symposium (UYMS)*. Izmir, Turkey: IEEE, Nov. 2021, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/9659761/

[141] A. I. Wang and N. Nordmark, "Software architectures and the creative processes in game development," in *International Conference on Entertainment Computing*. Springer, 2015, pp. 272–285.

[142] D. H. Eberly, *3D game engine design: a practical approach to real-time computer graphics*, 2nd ed. Amsterdam ; Boston: Elsevier/Morgan Kaufmann, 2007.

[143] E. Lengyel, *Foundations of game engine development*. Lincoln, California: Terathon Software LLC, 2016.

[144] E. F. Anderson *et al.*, "The Case for Research in Game Engine Architecture," in *Proceedings of the 2008 Conference on Future Play: Research, Play, Share*, ser. Future Play '08. New York, NY, USA: Association for Computing Machinery, 2008, pp. 228–231, event-place: Toronto, Ontario, Canada. [Online]. Available: https://doi.org/10.1145/1496984.1497031

[145] "Replication package for cosa," https://zenodo.org/record/7239232#.Y1RdfdLMIUE.

[146] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Systems Journal*, vol. 13, no. 2, pp. 115–139, 1974.

[147] L. C. Briand, J. W. Daly, and J. K. Wust, "A unified framework for coupling measurement in object-oriented systems," *IEEE Transactions on software Engineering*, vol. 25, no. 1, pp. 91–121, 1999.

[148] S. R. Chidamber and C. F. Kemerer, "Towards a metrics suite for object oriented design," in *Conference proceedings on Object-oriented programming systems, languages, and applications*, 1991, pp. 197–211.